N : xxxxxxxxxxx/IN

REPUBLIQUE ALGERIENNE DEMOCRATIQUE ET POPULAIRE

MINISTERE DE L'ENSEIGNEMENT SUPERIEUR ET DE LA RECHERCHE SCIENTIFIQUE

UNIVERSITE DES SCIENCES ET DE LA TECHNOLOGIE

«HOUARI BOUMEDIENNE»

Faculté d'Electronique et Informatique

MEMOIRE

Présenté pour l'obtention du diplôme de MAGISTER

En INFORMATIQUE

Spécialité : INFORMATIQUE MOBILE

PAR :

ZERAOULIA Née BELGUERCHE Nadia

Sujet

GROUPE K EXCLUSION MUTUELLE DANS LES RESEAUX MOBILES AD HOC

Soutenu publiquement le : 07/07/2011 devant le jury composé de :

Mr. M. Ahmed Nacer, Professeur, USTHB	Président
Mr. M. Benchaïba, Maître de Conférences /A, USTHB	Directeur de mémoire
Mme. S. Moussaoui, Maître de Conférences A, USTHB	Examinatrice
Mr. D. Tandjaoui, Maître de Recherche B, CERIST	Invité

DEDICACES

A mes parents…

A mon mari et mon fils Khalil

A toute ma famille

Sommaire

Introduction générale **1**

Chapitre 1 : Généralités sur les réseaux mobiles Ad hoc

1. Introduction **4**
2. Les environnements mobiles 5
    2.1 Définition 5
    2.2 Représentation d'un réseau mobile ad hoc 6
    2.3 Caractéristiques des réseaux mobiles ad hoc 6
    2.4Les applications possibles des réseaux mobiles ad hoc 7
    2.5Le routage dans les réseaux mobiles ad hoc 8
3. Classification des protocoles de routage 9
4. Conclusion 10

Chapitre 2 : Exclusion mutuelle de groupe (EMG)

1. Introduction **11**
2. Exclusion Mutuelle (EM) **11**
    2.1. Classification desalgorithmes d'exclusion mutuelle **12**
    2.2. Lesalgorithmes courants d'exclusion mutuelle **13**
        2.2.1. Algorithme de LAMPORT (1978) 13
        2.2.2. Algorithme de RICART et AGRAWALA (1981) 14
        2.2.3. Algorithme de CARVALHO et ROUCAIROL (1983) 14
        2.2.4. Algorithme de RICART et AGRAWALA(1983) 15
        2.2.5. Algorithme de SUZUKI et KASAMI (1985) 15
        2.2.6. Algorithme de NAIMI et TRAHEL (1987) 16
        2.2.7. Algorithme de MAEKAWA(1985) 17
    2.3. Discussion 18
3. Exclusion mutuelle de groupe (EMG) 18
    3.1. Les algorithmes courants d'exclusion mutuelle de groupe 19
        3.1.1. Algorithme de JOUNG (1998) 19
        3.1.2. Algorithme de WU et JOUNG (1999) 20
        3.1.3. Algorithme de RANGANATH et NEERAJ (1999) 23
        3.1.4. Algorithme de JOFFRO, SEBASTIEN, DATTA et PETIT (2003) 24
        3.1.5. Algorithme de CANTARELL, DATTA, PETIT et VILLAIN(2001) 26
    3.2. Exclusion mutuelle de groupe dans les réseaux mobiles ad hoc 27
        3.2.1. Algorithme de JIANG (2002) 27
        3.2.2. Algorithme de THIARE, NAIMI et GUEROUI (2006) 31
    3.3. Conclusion 31

Chapitre 3 : Groupe K Exclusion Mutuelle (Gk-EM)

1. Introduction 33
2. Groupe k exclusion mutuelle dans les réseaux mobiles ad hoc 34

3. Algorithmes courants de groupe k exclusion mutuelle 34
3.1 Algorithme de JIANG (2003).. 35
Figure1.1. Les liens de transmission dans les réseaux mobiles ad hoc…………………... 5 3.2 Algorith
me de THIARE, NAIMI et GUEROUI (2009) 40
4. Conclusion.. 40

Chapitre 4 : Le protocole de Groupe k exclusion mutuelle

1. Introduction 41
2. Approche globale du protocole 42
3. Principe de fonctionnement... 43
3.1 Circulation des jetons 43
3.2 Recherche du chemin vers le détenteur du jeton récolteur par le nœud détenteur du jeton distributeur... 44
3.3 Demande d'une session 45
3.4 Recherche de chemin vers le détenteur du jeton récolteur par un nœud demandeur 46
3.5 Libération de la section critique 47
3.6 Recherche du chemin vers le nœud détenteur du jeton distributeur................................. 48
3.7 Choix de la prochaine session 49
3.8 Perte des jetons 50
4. Hypothèses sur les nœuds et sur le réseau 51
5. Présentation du fonctionnement du protocole 52
5.1 Structures de données locales à chaque nœud.. 52
5.2 Les messages 53
5.3 Les différents événements 55
6. Discussion et conclusion 62

Chapitre 6 : Simulation du protocole

1. Introduction 65
2. Le simulateur NS2 66
2.1 Historique et présentation... 66
2.2 Fonctionnement 66
3. Implémentation du protocole de groupe k exclusion mutuelle 67
3.1 Méthode d'implémentation... 67
3.2 Environnement de la simulation 67
3.3 Mesures réalisées... 69
4. Résultats et interprétations 70
5. Discussion et conclusion 76
Conclusion générale 78
Références 80

Figure1.2. Changement de topologie dans les réseaux mobiles ad hoc…………………......
Figure1.3. La communication entre stations dans les réseaux mobiles ad hoc…………… 8
Figure 1.4. Classification des protocoles de routages…………………………………….. 9
Figure 2.1. Mécanisme de construction de l'arbre………………………………………... 16
Figure 2.2. Etats d'un philosophe et ses transitions……………………………………….. 18
Figure 3.1. Deux possibilités pour implémenter les primitives distribuées………………. 34
Figure 3.2. Exemple d'illustration du fonctionnement de l'algorithme…………………... 39
Figure 4.1 : Circulation des jetons………………………………………………………... 44
Figure 4.2 : Recherche du chemin vers le détenteur du jeton récolteur…………………….. 45
Figure 4.3. Demande d'une session………………………………………………………. 46
Figure 4.4. Recherche du chemin vers le nœud détenteur du jeton récolteur…………….. 47
Figure 4.5. Libération d'une session……………………………………………………… 48
Figure 4.6. Recherche du détenteur du jeton distributeur………………………………… 49
Figure5.1. Les graphes représentant le taux de sous-utilisation d'une session…………… 70
Figure 5.2. Les graphes représentant le taux de messages Release( ) non aboutis……….. 71
Figure 5.3. Les graphes représentant le taux de redemandes d'une session………………. 72
Figure 5.4. Les graphes représentant le nombre de messages redondants par SC………... 73
Figure 5.5. Les graphes représentant le nbMSG/SC……………………………………..... 73
Figure 5.6. Les graphes représentant les délais de synchronisation………………………. 75
Figure 5.7. Les graphes représentant Le taux de satisfaction……………………………... 76

Introduction générale

Durant ces dernières années, les utilisateurs de l'informatique moderne se sont orientés de plus en plus vers une politique de partage de l'information. En conséquence, les technologies de télécommunication n'ont cessés de progresser. D'autre part, nous avons assisté à une sensible démocratisation de la communication sans fil. Cette expansion a permis d'assurer un meilleur confort (mobilité) aux utilisateurs potentiels (gestionnaires) de l'informatique et des télécommunications, aux clientèles de la téléphonie et à d'autres utilisateurs.
Les interfaces de communication sans fil ont permis le développement des réseaux mobiles. Ces derniers peuvent être classés en deux catégories:

- Les réseaux cellulaires, dans lesquels les unités mobiles, pour communiquer entre elles passent par des stations fixes plus communément appelées "Stations de base". Ces réseaux définissent les réseaux mobiles avec infrastructures.
- Les réseaux mobiles ad hoc, dans lesquels les unités mobiles ne font appel à aucune infrastructure ou tout autre mécanisme centralisé. La communication est réalisée directement entre les unités mobiles.

Différentes contraintes caractérisent les réseaux mobiles ad hoc. La communication sans fil et la mobilité des nœuds imposent des restrictions quant à l'autonomie des unités mobiles. En effet, les nœuds formant un réseau mobile ad hoc soufrent de contraintes liées à l'énergie et à la largeur de la bande passante de communication. Hormis ces précédents problèmes, qui sont plus liées au matériel, la nature mobile des nœuds a créé d'autres : les formations/cassures de liens dont les conséquences sont : Les changements fréquents de la topologie, les différents partitionnements et fusionnements … etc.

Comme dans les systèmes distribués, le problème d'allocation de ressources est posé dans l'environnement mobile ad hoc. En effet, les nœuds mobiles partagent des ressources communes. Les processus qui constituent le système peuvent demander la même ressource en même temps. Si la ressource nécessite des accès exclusifs, un arbitrage est exigé afin d'allouer cette ressource équitablement. Ceci est le problème d'exclusion mutuelle. Dans ce même cadre, un autre problème a été introduit. Celui-ci permet de partager plusieurs ressources de même nature mais une seule est allouée à un moment donné à un nombre quelconque de demandeurs. Ce problème est communément nommé l'exclusion mutuelle de groupe. Dans certains cas, il s'avère que le nombre d'accès à une même ressource doit être limité à k, pour cela un nouveau problème a vu le jour. Ce problème s'énonce comme le Groupe K exclusion mutuelle. Ce dernier a été introduit par Jiang [Jia03] et a été modestement étudié dans la littérature. Pour ce faire, Jiang a fait une illustration du problème par l'exemple du CD-Jukebox. Dans cet exemple, plusieurs utilisateurs travaillent sur un projet qui dispose d'une très grande base de données enregistrée sur un périphérique de mémoire secondaire tel qu'un CD juke-box. Lorsqu'un utilisateur veut accéder à un objet de données, ce dernier sera chargé dans le buffer du périphérique. Afin d'augmenter les performances, les données chargées ne seront pas effacées jusqu'à ce qu'un autre objet

soit chargé et donc les utilisateurs voulant accéder à ces données, y accéderont d'une manière concurrente. Les autres utilisateurs voulant accéder à d'autres données devront attendre que le buffer se libère. De manière générale, les utilisateurs qui ont les mêmes centres d'intérêts peuvent les partager entre eux d'une manière concurrente tandis que les autres en sont exclus. Donc, si le CD-Jukebox peut être consulté par n'importe quel nombre de nœuds concurremment, alors la gestion d'accès au CD-Jukebox devient le problème de l'exclusion mutuelle de groupe (EMG). Autrement, si le nombre d'accès au CD-Jukebox est limité à k (la capacité d'accès concourante au CD-Jukebox est k), le problème devient celui de groupe k exclusion mutuelle (Gk-EM).
A notre connaissance, peu de travaux ont été effectués dans les réseaux mobiles ad hoc quant à l'exclusion mutuelle et notablement moins pour l'exclusion mutuelle de groupe et le groupe k exclusion mutuelle. Les algorithmes trouvés dans la littérature dédiés à ce type de réseau s'inspirent des algorithmes déjà existants dans les systèmes distribués moyennant une adaptation au nouvel environnement caractérisé par l'absence d'infrastructures et de contrôle centralisé, la mobilité des nœuds et des capacités limités des unités mobiles. Dans ce même environnement, l'utilisation de jeton est de rigueur car il permet une diminution considérable des messages générés par rapport aux systèmes à permission. C'est dans cette esprit que nous avons abordé le problème de *'groupe k exclusion mutuelle dans les réseaux mobiles ad hoc'* ayant pour objectif de concevoir un algorithme qui tente de suggérer des solutions aux problèmes posés au moindre coût.
L'algorithme proposé se base sur l'approche à jeton et est orienté recherche de jeton. Il utilise une topologie plate du fait que le maintien d'une structure logique dans ce type de réseau est difficile et couteuse notamment dans le cas de fortes mobilités des nœuds. Il se base sur la connaissance locale et ne se base pas sur la couche de routage. Par ailleurs, la solution prend en considération un certain nombre de caractéristiques d'un réseau mobile ad hoc entre autre la mobilité (les formations et ruptures de liens).

Dans notre algorithme, deux types de jetons sont utilisés, le jeton distributeur dont le rôle de son porteur est l'initiation d'une ressource (session), la gestion d'accès des nœuds à celle-ci et la fermeture de cette session. Le second est le jeton récolteur dont le rôle de son porteur est de récolter les demandes des différents nœuds afin de pouvoir les satisfaire par la suite. Lorsqu'une ressource est élue pour être utilisée, l'un des nœuds demandeurs de celle-ci sera destinataire du jeton récolteur. Dès que ce nœud reçoit ce jeton, il conclut que la prochaine session qui sera ouverte est celle qu'il demande. D`un autre côté, un nœud ne pourra accéder à la session qu'il demande que s'il détient les deux jetons récolteur et distributeur (, dans ce cas le nœud est qualifié d'initiateur) ou s'il est invité par le nœud initiateur de la session courante. De ce fait, lorsqu'un nœud reçoit le jeton récolteur, il attendra la réception du jeton distributeur pour pouvoir initier la session qu'il demande et inviter d'autres nœuds demandeurs de la même session à accéder à leur session.

Par ailleurs, dans la solution proposée, le partitionnement ne peut pas avoir lieu. Cependant, elle prend en charge l'isolement temporaire (ce dernier est limité dans le temps et ne peut pas causer de duplication de jetons) et la panne (sans causer de partitionnement dans le réseau) des nœuds. La panne d'un nœud détenteur de l'un ou des deux jetons cause implicitement une perte de l'un ou des deux jeton(s). Dans notre solution, la perte de jetons est détectée par un nœud demandeur d'une ressource, qui après un certain temps ne reçoit ni les jetons ni invitation. Dans ce cas, le nœud fait un certain nombre de tentatives avant de soupçonner la perte des deux jetons dans le cas où il n'a reçu aucun des jetons, ou bien la perte du jeton distributeur s'il a reçu le jeton récolteur mais pas le jeton distributeur. Pour ce faire, il lance une recherche des jetons dans le réseau. Cette recherche est réalisée à travers un message spécifique qui sera diffusé dans tout le réseau. Celui-ci permettra d'une part de vérifier l'existence du jeton récolteur dans le réseau mais aussi l'existence du jeton distributeur. Dans le cas où la recherche indique que les deux jetons sont perdus, le nœud ayant le plus petit identificateur crée une autre paire de jetons (cette

méthode est similaire à certains mécanismes d'élection). Autrement, c'est le détenteur du jeton existant qui sera responsable de créer le jeton perdu.
Ce document est structuré en cinq chapitres regroupés en trois parties. La première partie est dédiée à la présentation de la taxonomie des travaux connus dans la littérature comme suit :

- Le premier chapitre donne des généralités sur les réseaux mobiles ad hoc. Les chapitres deux et trois sont dédiés aux problèmes d'exclusion mutuelle, de l'exclusion mutuelle de groupe et de groupe k exclusion mutuelle qui sont étroitement liés.

- La deuxième partie constituée du chapitre quatre, traite la solution que nous proposons pour le problème de groupe k exclusion mutuelle dans l'environnement mobile ad hoc. Elle décrie le principe général qui fixe les grands choix. Ensuite, une présentation événementielle du protocole a été succincte suivie d'une discussion sur certains aspects du protocole. Le texte du protocole est donné en annexe.

- La troisième partie constituée du chapitre cinq présente une étude de simulation : Il est question d'évaluer les performances de notre algorithme dans un contexte prédéfini. Le chapitre commence par présenter le simulateur NS2 qui représente la plateforme de ce travail, les différents paramètres pris en compte dans les mesures effectuées, les critères de performances évalués ainsi que les différents résultats obtenus avec une interprétation qui met l'accent sur le déroulement de l'algorithme et tente d'expliquer certains résultats par rapport aux conditions de simulation et de journal des événements générés par le programme.

Chapitre 1 :

Généralités sur les réseaux mobiles Ad hoc

1. Introduction

L'essor des technologies sans fil, offre aujourd'hui de nouvelles perspectives dans le domaine des télécommunications. L'évolution récente des moyens de la communication sans fil a permis la manipulation de l'information à travers des unités de calculs portables qui ont des caractéristiques particulières (une faible capacité de stockage, une source d'énergie autonome...) et accèdent au réseau à travers une interface de communication sans fil. En comparaison avec l'ancien environnement (l'environnement statique), le nouvel environnement résultant appelé environnement mobile, permet aux unités de calcul, une libre mobilité et il ne pose aucune restriction sur la localisation des usagers. La mobilité (ou le nomadisme) et le nouveau mode de communication utilisé, engendrent de nouvelles caractéristiques propres à l'environnement mobile : une fréquente déconnexion, un débit de communication et des ressources modestes, et des sources d'énergie limitées.

Les environnements mobiles offrent une grande flexibilité d'emploi. En particulier, ils permettent la mise en réseau des sites dont le câblage serait trop onéreux à réaliser dans leur totalité, voire même impossible (par exemple en présence d'une composante mobile).

Les réseaux mobiles sans fil, peuvent être répartis en deux classes : les réseaux avec infrastructure qui utilisent généralement le modèle de la communication cellulaire, et les réseaux sans infrastructure ou les réseaux *ad hoc*. Plusieurs systèmes utilisent déjà le modèle cellulaire et connaissent une très forte expansion à l'heure actuelle (les réseaux *GSM* par exemple) mais requièrent une importante infrastructure logistique et matérielle fixe.

La contrepartie des réseaux cellulaires sont les réseaux mobiles ad hoc. Un réseau ad hoc peut être défini comme une collection d'entités mobiles interconnectées par une technologie sans fil formant un réseau temporaire sans l'aide de toute administration ou de tout support fixe. Aucune supposition ou limitation n'est faite sur la taille du réseau, cela veut dire qu'il est possible que le réseau ait une taille très énorme.

Dans un réseau ad hoc, les hôtes mobiles doivent former, d'une manière ad hoc, une sorte d'architecture globale qui peut être utilisée comme infrastructure du système. Les applications

des réseaux ad hoc sont nombreuses, on cite l'exemple classique de leurs applications de secours et les missions d'expérience.

Du fait que le rayon de propagation des transmissions des hôtes soit limité, et afin que le réseau ad hoc reste connecté, (c'est-à-dire toute unité mobile peut atteindre toute autre), il se peut qu'un hôte mobile se trouve dans l'obligation de demander de l'aide à un autre hôte pour pouvoir communiquer avec son correspondant. Il se peut donc que l'hôte destination soit hors de la portée de communication de l'hôte source, ce qui nécessite l'emploi d'un routage interne par des nœuds intermédiaires afin de faire acheminer les paquets de messages à la bonne destination.

La gestion de l'acheminement de données ou le routage, consiste à assurer une stratégie qui garantit, à n'importe quel moment, la connexion entre n'importe quelle paire de nœuds appartenant au réseau. La stratégie de routage doit prendre en considération les changements de la topologie ainsi que les autres caractéristiques du réseau ad hoc (bande passante, nombre de liens, ressources du réseau, etc.). En outre, la méthode adoptée dans le routage, doit offrir le meilleur acheminement des données en respect des différentes métriques de coûts utilisées.

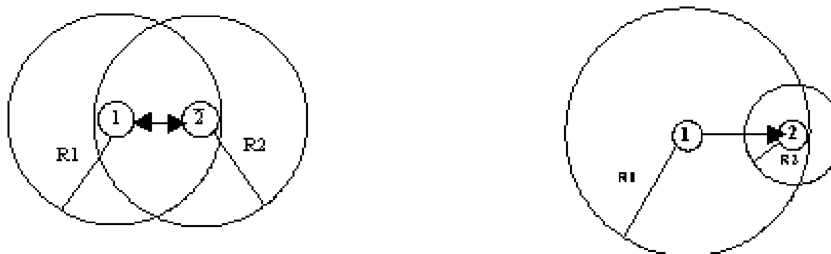
2. Les Réseaux Mobiles ad hoc

2.1 Définition

Un réseau mobile ad hoc, appelé généralement MANET (Mobile Ad hoc NETwork), consiste en une grande population, relativement dense, d'unités mobiles qui se déplacent dans un territoire quelconque et dont le seul moyen de communication est l'utilisation des interfaces sans fil, sans l'aide d'une infrastructure préexistante ou administration centralisée.

Afin de recevoir et d'émettre, les unités mobiles sont équipées d'émetteurs et de récepteurs tels que les cartes Wifi auxquelles le débit peut atteindre 54Mbps.

A un instant donné, selon la topologie du réseau (le positionnement des nœuds mobiles, la configuration des antennes d'émission et de réception, la puissance du signal ainsi que les interférences entre les canaux), une certaine connectivité existe entre les unités mobiles. Ce dernier change avec le temps en fonction du mouvement des nœuds ou d'autres changements de paramètres de configuration. La figure1.1 montre comment la mobilité et la portée du champ de communication influent sur la nature des liens de transmission.



(a) Lien bidirectionnel (b) Lien unidirectionnel

R1 : portée du nœud 1.

R2 : portée du nœud 2.

Figure1.1: *Les liens de transmission dans les réseaux mobiles ad hoc.*

2.2 Représentation d'un réseau mobile ad hoc

Un réseau mobile ad hoc est représenté sous forme d'un graphe orienté ou non orienté selon la nature des liens qui existent entre les différentes unités mobiles, G(t)= {M(t), L(t)} où :

- M(t) représente l'ensemble d'unités mobiles (nœuds) à l'instant t.
- L(t) représente l'ensemble des liens qui existent entre les différents nœuds à l'instant t.

Le mouvement des nœuds induit le changement fréquent de topologie donc des connections et des déconnections fréquentes.

2.3 Caractéristiques des réseaux mobiles ad hoc

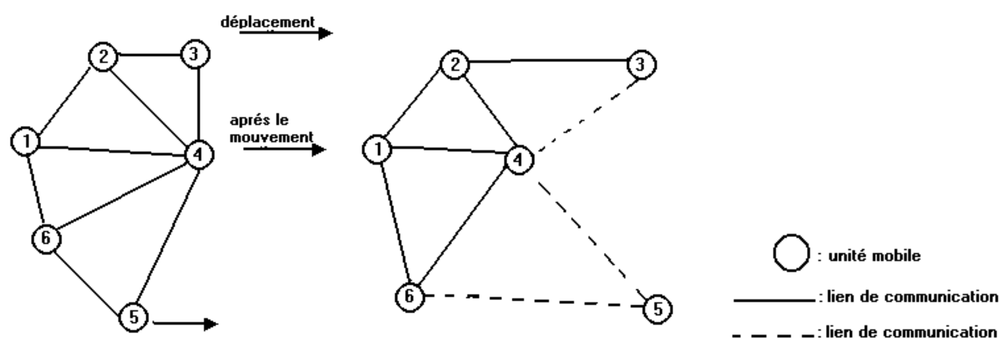
Parmi les principales caractéristiques des réseaux mobiles ad hoc, on distingue : l'absence d'infrastructure, topologie dynamique, contrainte sur la bande passante, contrainte d'énergie et sécurité limité.

- ✓ *Absence d'infrastructure*

Un réseau mobile ad hoc est caractérisé par l'absence d'infrastructure préexistante et toute administration centralisée car dans les réseaux mobiles ad hoc, les unités mobiles communiquent entre elles sans passer par aucune antenne relais ou tout dispositif fixe. De ce fait, cette caractéristique est considérée comme fondamentale (les données ne passent que par les unités mobiles).

- ✓ *Topologie dynamique*

Le mouvement des unités mobiles étant aléatoire, des changements de connectivité surviennent d'une manière imprévisible (figure1.2). En effet, un lien de communication existant entre deux stations peut se briser à n'importe quel moment, pour peu que les stations s'éloignent l'une de l'autre. Ce mouvement peut induire aussi à la création de nouveaux liens de communication, de cette manière, on peut remplacer les routes brisées par de nouvelles. De ce fait, cette caractéristique peut être vue comme une contrainte car le processus de maintenance des liens de communication peut s'avérer très lourd.



(a) L'ancienne topologie (b) La nouvelle topologie

Figure1.2 : *Changement de topologie dans les réseaux mobiles ad hoc.*

- *Contrainte sur la bande passante*

Les réseaux sans fil se basent sur le partage des médiums de communication, mais la bande passante réservée à ce type de réseaux est relativement modeste. Ce qui implique une utilisation de protocoles permettant un partage efficace de cette ressource.

- *Contrainte d'énergie*

Les unités mobiles sont alimentées à travers des sources d'énergie autonomes telle que la batterie, ce qui réduit le temps de disponibilité des unités mobiles mais l'utilisation de nouvelles technologies (processeur à fréquence variable par exemple) permet l'optimisation de la consommation d'énergie.

- *Sécurité limitée*

Les réseaux mobiles ad hoc sont considérés comme étant très vulnérables en matière d'attaque, vu les contraintes précédentes ainsi les méthodes de sécurité (cryptage…etc) sont réduites ce qui augmente le risque d'attaque ou de piratage. Les pirates informatiques peuvent accéder d'une manière directe à la donnée en utilisant des antennes pirates (car les données circulent par voie hertzienne) ou en obligeant une station à consommer une bonne partie de ses ressources d'énergie en l'inondant de requêtes inutiles.

2.4 Les Applications possibles des réseaux mobiles ad hoc

Les réseaux mobiles ad hoc ont un très grand potentiel d'applications dans les différents domaines. Nous pouvons citer :

- *La communication tactique*

Ce sont des applications qui concernent le domaine militaire vu la facilité du mouvement comme: le partage d'informations de position sur la cible ou sur l'ennemi, facilitation de la synchronisation, coordination et guidage des groupes sans passer par des stations fixes ce qui à petite échelle minimise le risque d'interception des messages.

- *Opérations de secours*

Ce réseau peut être utilisé aussi lors d'une opération de sauvetage suite aux catastrophes naturelles (incendies, inondations,…etc.). Ils peuvent satisfaire et résoudre le problème de communication là où on ne peut installer un réseau filaire qu'après un très long délai.

- *Les systèmes de conférences*

Les réseaux mobiles ad hoc facilitent le partage et l'échange d'informations entre les participants d'une conférence.

- *Les sites de patrimoines*

Dans les sites archéologiques, musés (anciens musés), châteaux, palais ou tout autre lieu semblable où l'installation des réseaux filaires est interdite à cause du risque de nuisance sur de tels environnements, les réseaux mobiles ad hoc peuvent être utiles.

- *En privé*

On peut échanger des données librement et faire des connections en Peer-to-Peer [site 05].

2.5. Le routage dans les réseaux mobiles ad hoc

2.5.1 Définition

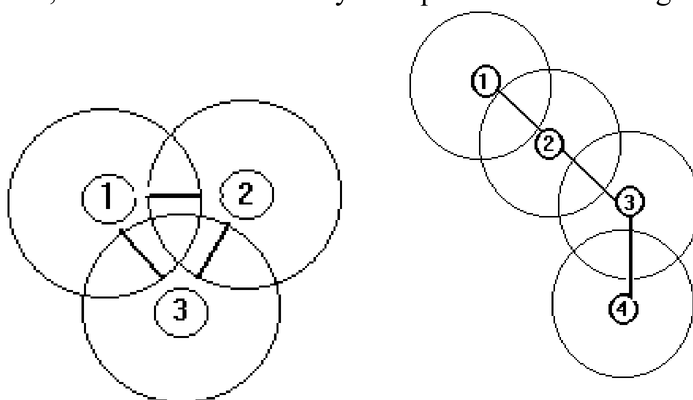
Le routage est une méthode d'acheminement des informations entre une station source et une autre destination à travers un réseau de communication donné. Selon les caractéristiques du réseau (capacité des liens, complexité, …etc.), l'acheminement des paquets à travers ce dernier doit être optimisé au sens d'un certain critère de performance. Donc, l'objectif de toute méthode de routage est d'assurer l'acheminement de données au moindre coût.

2.5.2 Le Problème de routage dans les réseaux mobiles ad hoc

Comme nous l'avons déjà défini, un réseau mobile ad hoc est caractérisé par l'absence de toute infrastructure préexistante à l'inverse des réseaux de télécommunication classiques .Un réseau mobile ad hoc doit s'organiser automatiquement d'une manière rapide afin de s'adapter aux conditions de propagation, au trafic et aux différents mouvements pouvant intervenir au sein des unités mobiles.

Afin d'assurer une connectivité du réseau avec l'absence d'infrastructure fixe et la mobilité des stations, chaque nœud doit participer au routage pour permettre à un nœud qui n'est pas en mesure d'atteindre directement sa destination de transmettre ses paquets. Ainsi, chaque nœud joue le rôle de station et de routeur. Dans la communication entre station mobiles, deux cas se présentent (voir la figure 1.3) :

- le premier cas (figure 1.3(a)) est celui où les stations sont dites voisines directes, ie: chaque station est dans les champs respectifs des autres. Ainsi, la communication est directe donc ne nécessite pas de recourt au routage.
- le second cas (figure 1.3(b)) est celui où les deux stations qui veulent communiquer ne sont pas voisines. Donc pour acheminer l'information, cette dernière doit passer par une ou plusieurs stations qui jouent le rôle de relais. De cette manière, on aura créé une route composée de plusieurs sauts entre les stations (multi hop-route). Dans les réseaux mobiles ad hoc, cette route est créée dynamiquement vu le changement fréquent de la topologie.



(a) Les nœuds communiquent d'une manière directe

(b) Pour que 1 communique avec 4 les données doivent passer par 2 et 3.

Figure1.3 La communication entre stations dans les réseaux mobiles ad hoc.

Des algorithmes de routage fonctionnant dans des environnements statiques tel que état de lien (Link state) [TAN96], Vecteur_distance (Distance-vector) [TAN96] ne sont pas très efficaces dans des environnements mobiles. C'est pour cette raison que plusieurs protocoles spécifiques aux réseaux mobiles ad hoc ont été proposés par différentes équipes de recherches.

3. Classification des protocoles de routage

La stratégie (ou protocole) de routage est utilisée dans le but de découvrir les routes qui existent entre les nœuds. Le but principal d'une telle stratégie est l'établissement de routes qui soient correctes et efficaces entre une paire quelconque d'unités, ce qui assure l'échange des messages de manière continue. Vu les limitations des réseaux mobiles ad hoc, la construction de routes doit être faite avec un minimum de contrôle et de consommation de bande passante. Suivant la manière de création et de maintenance de routes lors de l'acheminement de donnés, les protocoles de routages peuvent être séparés en deux catégories, les protocoles ***proactifs*** (tel que : [PER94]) et les protocoles ***réactifs*** (tel que : [JOH01], [PER98], [COR95], [KO98])**.** Les protocoles proactifs établissent des routes à l'avance en se basant sur l'échange périodique des tables de routages. Ces protocoles différents dans le nombre de tables maintenues et la manière de réaliser les mises à jour. L'inconvénient de ces protocoles est qu'ils propagent et maintiennent les informations de routages indépendamment du besoin des nœuds. Les protocoles réactifs cherchent les routes à la demande. Lorsque le réseau a besoin d'une route, un processus de découverte globale de route est déclenché. Plusieurs approches peuvent être appliquées ; la majorité des algorithmes utilisés sont basés sur le mécanisme d'apprentissage en arrière (backward learning). Le nœud source, qui est à la recherche d'un chemin, diffuse par inondation une requête dans le réseau. Une fois que la destination est atteinte, elle peut envoyer une réponse en utilisant le chemin tracé par la requête. Il existe d'autres protocoles de routages appelés ***hybrides*** (tel que : [PEA96, ZYG97]). Ces protocoles incluent plusieurs aspects des protocoles de routages proactifs et réactifs. Ils fournissent en général un routage hiérarchique où les nœuds sont groupés en clusters. Un protocole réactif assure le routage au sein du cluster et un protocole proactif est utilisé pour la communication inter cluster.

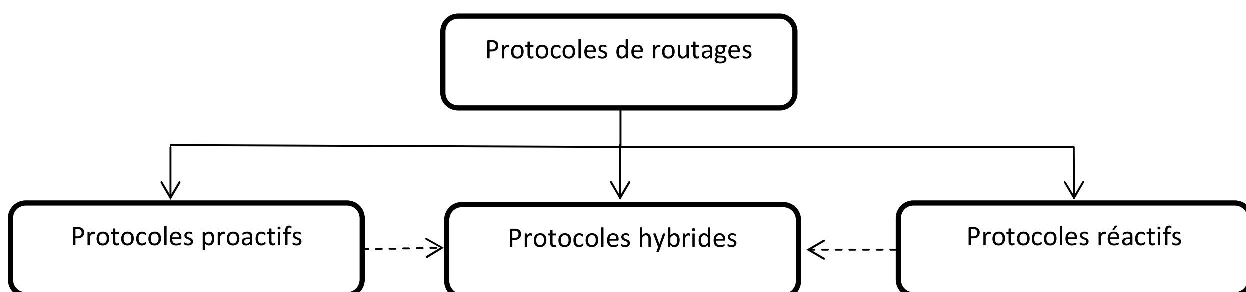


Figure 1.4 : Classification des protocoles de routages.

4. Conclusion

Comme nous l'avons vu dans cette partie, les réseaux mobiles ad hoc sont très prometteurs en matière de développement dans le domaine de l'informatique mobile. Seulement la réalisation de tels réseaux doit prendre en considération beaucoup de contraintes, donc la complexité de réalisation de tels réseaux est due en grande partie aux caractéristiques de ces réseaux telles que : Topologie dynamique, contraintes sur la bande passante, contrainte d'énergie et la sécurité limitée. Ce qui implique que la réalisation de tout algorithme distribué visant à satisfaire le besoin des utilisateurs tout en optimisant les services offerts par ces réseaux est très complexe car on est obligé de prendre en charge de nouvelles contraintes en plus de celles qui existent déjà dans les réseaux statiques.

Chapitre 2 :

Exclusion mutuelle de groupe (EMG)

1. Introduction

L'exclusion mutuelle de groupe est une généralisation intéressante du problème de l'exclusion mutuelle. Ce problème a été présenté par Joung [Jou98], et quelques algorithmes pour le problème ont été proposés en incorporant des algorithmes d'exclusion mutuelle. L'exclusion mutuelle de groupe se produit naturellement dans une situation où une ressource peut être partagée par des processus du même groupe. Mais pas par des processus de différents groupes. Dans ce qui suit, nous allons dans un premier temps s'intéresser à l'exclusion mutuelle et ce afin de pouvoir introduire dans un deuxième temps l'exclusion mutuelle de groupe étant donné leur lien étroit.

2. Exclusion mutuelle (EM)

Le problème d'allocation en exclusivité d'une seule ressource (Exclusion Mutuelle) est un paradigme des problèmes de compétition dans les systèmes. Il s'agit d'un problème de compétition dont l'énoncé est très simple : une entité, appelée ressource non partageable ou critique, ne peut être octroyée à un instant donné qu'à un seul processus parmi N processus du système. Autrement dit, dans un tel problème, c'est l'aspect de concurrence qui domine lorsque plusieurs processus tentent en même temps d'accéder à une même ressource. En effet, l'utilisation de la ressource critique simultanément par un ensemble de processus peut créer une situation d'incohérence. D'où la nécessité du développement des algorithmes de contrôle ayant comme fonctions principales : synchroniser les différents processus d'un système distribué pour l'accès à la même ressource, et éviter toute incohérence. Ce sont les algorithmes distribués d'exclusion mutuelle.

Deux approches peuvent être utilisées pour implémenter l'EM dans les systèmes distribués. L'approche centralisée, dans laquelle un seul nœud fonctionne comme coordinateur central: c'est lui qui accorde la permission d'entrée en SC. L'approche distribuée, dans laquelle la décision est faite à travers tout le système d'où la complexité de cette approche (car il n'y a pas de mémoire partagée, ni d'horloge physique commune et les délais de transmission des messages sont imprévisibles).

2.1.Classification des algorithmes d'exclusion mutuelle

Les algorithmes d'exclusion mutuelle peuvent être classés selon différents critères. De ces critères on distingue :

- le modèle d'exclusion mutuelle : est-ce que l'algorithme se base sur l'approche jeton ou sur l'approche permission?
- L'échelle de l'exclusion mutuelle : est-ce qu'un nœud voulant accéder à la SC, va devoir communiquer avec tous les nœuds du système ou seulement avec une partie d'entre eux ?

Selon le modèle, les différentes catégories sont comme suit :

A) *Les algorithmes utilisant le jeton*

Ils sont matérialisés par un objet spécial" jeton" ou "privilège", il est unique dans tout le système. Un nœud voulant accéder à la SC ne peut le faire que s'il possède le jeton, ce dernier lui donne ce privilège. Le jeton est un message spécial et unique ce qui implique que seul le processus qui le possède, peut accéder à la SC et à un moment donné le jeton doit être possédé par un seul nœud au plus ce qui garantit l'exclusion mutuelle. Le nœud possédant le jeton choisit le nœud privilégié suivant. Le jeton circule entre les nœuds du système selon différents mécanismes. De ces mécanismes, on distingue:

- Si les nœuds sont organisés selon une topologie logique en anneau, le jeton circule en faisant le tour de tous les nœuds et chaque nœud voulant accéder à la SC pourra le faire dès la réception du jeton. Si au contraire il ne veut pas le faire, il fait passer le jeton au prochain nœud (sur l'anneau). Si les liens entre les nœuds sont unidirectionnels, le problème de la famine est évité. Dans le cas où la charge des demandes serait faible, cette méthode est très coûteuse en terme de messages échangés (car le nombre de nœuds voulant accéder à la SC est très limité. Donc, le jeton va continuer à circuler inutilement dans le réseau).
- Une autre méthode, consiste à ce que le nœud voulant accéder à la SC demande l'acquisition du jeton à celui qui le détient. Ce dernier, à la fin d'exécution de la SC, choisit un des nœuds demandeurs pour lui envoyer le jeton. Si aucun nœud ne désire entrer en SC, il garde le jeton au repos. Le problème majeur de cette méthode réside dans la localisation du porteur du jeton pour minimiser les échanges de messages générés par les nœuds voulant accéder à leur SC.

  Quelques problèmes peuvent survenir dans l'approche à jeton. Les plus courants sont la perte de jeton (qui provoque un inter blocage), ou la duplication du jeton (l'exclusion mutuelle n'est plus assurée). D'où la nécessité d'intégration de mécanismes de régénération et de contrôle d'unicité du jeton, qui peuvent être très complexes à mettre en œuvre.

B) *Les algorithmes utilisant l'approche à permission*

Le droit d'accès à la SC est formalisé par la réception d'une permission de la part d'un ensemble de nœuds du système. Chaque nœud voulant entrer en SC, demande la permission d'y accéder aux autres nœuds. Les nœuds qui ne veulent pas accéder en SC, émettent leurs permissions à ceux qui veulent y accéder. Chaque nœud doit donner sa permission à un seul

nœud à un instant et un ordre de priorité doit être mis en œuvre entre les nœuds en concurrence, pour qu'un seul d'entre eux puisse entrer en SC.

Un des problèmes de cette méthode est de trouver le nombre minimal de permissions que doit réunir le nœud voulant accéder à la SC afin de réduire le nombre de messages nécessaires à l'invocation de cette dernière.

2.2.Algorithmes courants d'exclusion mutuelle

Dans ce qui suit, nous allons présenter quelques algorithmes courants d'exclusion mutuelle. Ces derniers ont été présentés dans le but de simplifier la compréhension des protocoles d'exclusion mutuelle de groupe et de Groupe k exclusion mutuelle de groupe (qui font référence aux algorithmes d'exclusion mutuelle présentés) qui feront l'objet de la prochaine partie. Ces algorithmes se basent sur les deux approches (permission et jeton), en mettant en évidence les structures de données qu'ils utilisent, leurs modes de fonctionnement, ainsi que le nombre de messages nécessaires pour accéder à la SC. Dans tous les algorithmes qui suivent, N désigne le nombre de nœuds dans le système.

2.2.1. Algorithme de LAMPORT (1978)

L'algorithme développé dans [LAM78] est basé sur l'approche à permission, et utilise le concept d'horloge logique pour garder l'ordre des requêtes. Cet algorithme utilise une horloge logique Ci du nœud i sachant que chaque nœud possède une file d'attente propre à lui. Cette file d'attente est un tableau où chaque élément *j* contient le type du dernier message reçu par le nœud j et l'estampille de ce message. Cet algorithme utilise trois messages : **Requête, Libération** et **Acquittement**.

Quand un processus i veut entrer à la SC, il envoie le message ***REQUEST*** (C_i ,i) à tous les autres nœuds du système, met ce message dans sa file d'attente et se met en attente. Quand le processus j reçoit le message ***REQUEST*** du nœud i, il met à jour son horloge logique C_j (tel que: $C_j > C_i$ où C_i représente l'estampille associée à l'évènement d'émission de ce message par le nœud i et C_j représente l'estampille associée à l'évènement de réception de ce message par le nœud j), empile la demande de i dans sa propre file d'attente et émet un message ***REPLY***(C_j ,j) à i. Si i reçoit une réponse des (N-1) autres nœuds et que leurs estampilles sont plus grandes que la sienne (celle de sa demande), et qu'il est en tête de file, alors i pourra accéder à la SC. Quand un nœud i sort de sa SC, il supprime la requête (C_i ,i) de sa file d'attente, et émet un message ***RELEASE***(C_i ,i) aux (N-1) autres nœuds du système. Quand un nœud j reçoit le message ***RELEASE***(C_i ,i),il supprime lui aussi la requête (C_i ,i) de sa file .

Quand deux nœuds i, j veulent accéder à la SC, et émettent un message ***REQUEST*** avec la même estampille, celui qui possède le plus petit identificateur passera en premier.

Cet algorithme suit une politique FIFO et ne traite pas les défaillances des liens de communication et des nœuds. Le nombre de messages échangés pour une entrée à la SC est de

3*(N-1) {c'est à dire: (N-1) messages ***REQUEST*** + (N-1) messages ***REPLY*** + (N-1) messages ***RELEASE***}.

2.2.2. Algorithme de RICART et AGRAWALA (1981)

L'algorithme développé dans [RA81] est basé sur l'approche à permission, et utilise le concept du numéro de séquence (NS).

Quand un nœud i veut accéder à la SC, il génère un numéro de séquence (***NS***) et émet un message ***REQUEST*** vers les (N-1) autres nœuds du système, puis se met à attendre. Le nœud i ne pourra accéder à la SC que s'il reçoit les (N-1) messages ***REPLY*** des autres nœuds. Quand un nœud j reçoit une demande de i, j émet un message ***REPLY*** s'il n'est pas demandeur de la SC, autrement si j est moins prioritaire que i alors j émet un message ***REPLY*** sinon la demande sera détenue par j jusqu'à ce qu'il sorte de sa SC. Quand un nœud k quitte sa SC, il émet un message ***REPLY*** à tous les nœuds dont la requête a été détenue.

Cet algorithme requiert 2*(N-1) messages pour accéder à la SC (c'est à dire: (N-1) messages ***REQUEST*** + (N-1) messages ***REPLY***) et suit une politique FIFO.

2.2.3. Algorithme de CARVALHO et ROUCAIROL (1983)

L'algorithme développé dans [CR83] est basé sur l'approche à permission. Il est une variante de l'algorithme de RICART et AGRAWALA [RA81], mais essaie de minimiser le nombre de permissions dont un nœud doit avoir pour accéder à la SC.

Cet algorithme utilise un tableau de booléens ***Permission_Valid[1…N-1] (PV[1…N-1])***, pour indiquer si la permission d'un nœud k est toujours valide, initialement à faux.

Le fonctionnement de cet algorithme est semblable au fonctionnement de l'algorithme de RICART et AGRAWALA[RA81], à un détail près, c'est quand un nœud i reçoit la permission d'entrer en SC de la part d'un nœud j, i met à jour la variable ***Permission_Valid[j]*** (c'est à dire: ***Permission_Valid[j]*** := Vrai), cette mise à jour signifie que la prochaine fois que i voudra accéder à la SC, il n'aura pas besoin de demander la permission à j tant que celui-ci n'a pas émis une nouvelle demande d'accès à la SC (comme si sa permission est toujours acquise par i), en d'autres termes i ne demandera la permission qu'aux nœuds qui ont émis des demandes d'accès à la SC.

- La mise à jour de la variable ***Permission_Valid[j]***, devient obsolète après la réception d'un message ***REQUEST*** émis par j.
- Si aucun nœud ne demande l'accès à la SC, le dernier nœud l'ayant quitté pourra de nouveau y accéder directement sans demander de permission.
- Le nombre de messages échangés pour accéder à la SC varie entre 0 et 2*(N-1).

2.2.4. Algorithme de RICART et AGRAWALA(1983)

L'algorithme développé dans [RA83] est basé sur l'approche à jeton, il utilise le concept de numéro de séquence (NS).

Quand un nœud i veut accéder à la SC: s'il possède le jeton, il y accède directement. Dans le cas contraire, il incrémente son NS, émet un message ***REQUEST*** aux (N-1) autres nœuds du système. Quand un nœud j reçoit le message ***REQUEST*** de i, il met à jour son NS, si le nœud j ne possède pas le jeton, alors il enregistre la demande de i. Sinon, il cherche dans une liste logique circulaire le premier nœud qui possède la demande la plus ancienne, et émet le jeton. Quand un nœud k quitte la SC, il met à jour son NS pour indiquer que sa demande vient d'être satisfaite. Si aucun nœud ne demande le jeton, celui qui le possède le maintien au repos.

Le jeton dans sa traversée suit un parcours logique en anneau circulaire, dans un ordre ascendant des identificateurs des nœuds. Ce qui implique que ce n'est pas forcément la plus ancienne demande qui va être satisfaite en premier.

L'algorithme requiert N messages pour accéder à la SC, et 0 messages si le nœud possède le jeton.

2.2.5. Algorithme de SUZUKI et KASAMI (1985)

Cet algorithme [SK85], utilise une topologie complètement maillée, se base sur l'approche à jeton et est une amélioration de l'algorithme de RICART et AGRAWALA [RA83]. Le principe de base de cet algorithme est de transférer le privilège d'entrer en section critique en utilisant un message unique représentant ce privilège d'accès. Un nœud désirant obtenir le privilège envoie un message de requête à tous les autres nœuds. Le nœud recevant le message privilège est autorisé à entrer dans sa section critique autant de fois qu'il le désire et aussi longtemps que le privilège ne lui a pas été sollicité.
Un message de requête en provenance d'un nœud j a la forme ***REQUEST***(j,n) où j est l'identificateur du nœud et n (n=1,2,…) est le numéro de séquence indiquant que le nœud j est en train de faire sa $(n+1)^{ième}$ invocation de la section critique. Chaque nœud manipule un tableau *RN* de taille N (N étant le nombre de nœuds dans le système) pour enregistrer le plus grand numéro de séquence reçu de chaque nœud dans la case correspondante. Lorsque le message ***REQUEST***(j,n) arrive à un nœud i, celui-ci calcule *RN*[j] := max(*RN*[j],n). Le nœud i utilise *RN*[i] pour générer son propre numéro de séquence.

La forme du message de privilège est ***PRIVILEGE***(Q,LN) ; où, Q est la file des requêtes reçues et *LN* est un tableau de taille N tel que *LN*[j] est le numéro de séquence de la dernière requête satisfaite du processus j. Lorsque le nœud i fini d'exécuter sa section critique, le tableau *LN* reçu avec le dernier privilège de i est modifié tel que *LN*[i] := *RN*[i] pour indiquer que la requête courante vient d'être satisfaite. Ensuite, tous les identificateurs j satisfaisants la condition *RN*[j] = *LN*[j]+1 (le nœud j est demandeur) sont enfilés dans la queue Q s'il n'y sont pas déjà. Si la queue

Q est vide, le privilège est retenu jusqu'à l'arrivée d'une requête. Si la queue Q n'est pas vide, le message de privilège est envoyé au nœud qui se trouve au sommet de file.
L'algorithme de SUZUKI et KASAMI[SK85] nécessite N messages pour chaque entrée en section critique ou 0 messages si le nœud demandeur est détenteur du jeton. La communication est supposée fiable ; les délais de transfert sont finis mais imprévisibles et la préservation de l'ordre des messages n'est pas requise.

2.2.6. Algorithme de NAIMI et TRAHEL (1987)

L'algorithme développé dans [NT87] est basé sur l'approche à jeton et impose une structure logique au réseau, cette dernière est dynamique. Chaque nœud i maintient deux variables:

$LAST_i$: elle indique le nœud auquel un message ***REQUEST*** doit être envoyé (cette structure est nulle au niveau de la racine).

$NEXT_i$: elle indique le nœud auquel le jeton doit être acheminé après la libération de la SC.

Physiquement, les nœuds sont complètement interconnectés par un réseau fiable (pas de panne). Logiquement, les nœuds forment une structure d'arborescence dont la racine est le nœud qui détient le jeton. Cette structure est construite de la manière suivante:

Si un nœud i est détenteur du jeton, il met ***$LAST_i$*** à 0 et ***$NEXT_i$*** à 0 puis envoie un message ***init*(j)** à tous ses voisins directs. Sinon, il attend jusqu'à la réception d'un message ***init*(j)**, met ***$LAST_i$*** à j et ***$NEXT_i$*** à 0 puis envoie à tous ses voisins directs sauf j un message ***init*(i)**. Les messages ***init*(k)** qui sont reçus par la suite seront ignorés. La figure 2.1 représente le mécanisme de construction de l'arbre.

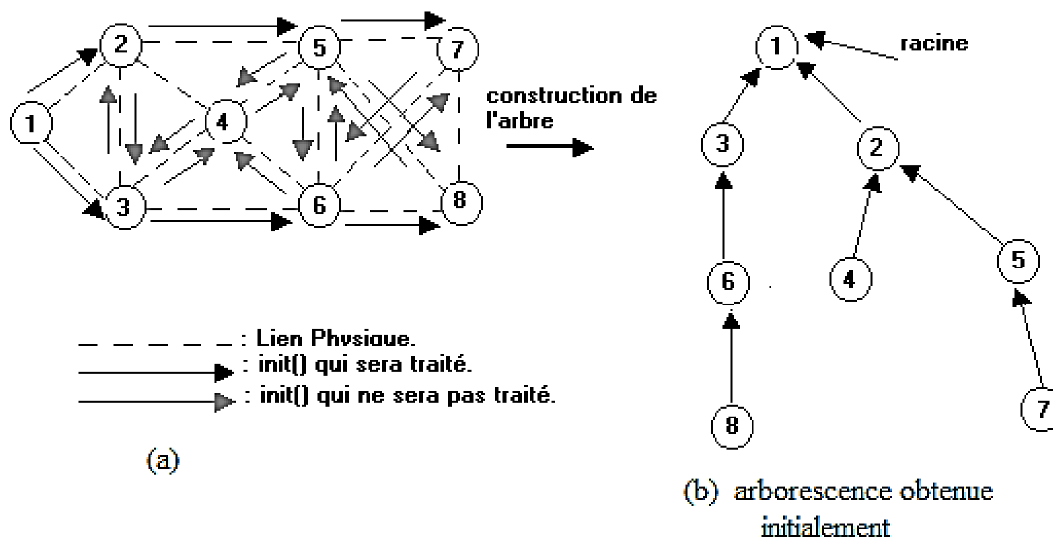


Figure 2.1: *Mécanisme de construction de l'arbre.*

Quand un nœud i veut accéder à la SC, il émet un message ***REQUEST(i)*** vers le nœud indiqué par la variable $LAST_i$ puis met $LAST_i$ à 0 (il se considère comme la nouvelle racine). Quand un nœud intermédiaire j reçoit un message ***REQUEST(i)***, il le transmet au nœud indiqué par $LAST_j$ et met $LAST_j$ à i. Maintenant, quand le nœud racine reçoit un message ***REQUEST(i)***, il met sa variable ***LAST*** à i et quand il quitte sa SC, il envoie le jeton à i; sinon, il met sa variable ***NEXT*** à i. Ainsi, s'il y a plusieurs nœuds demandeurs de SC, les variables ***NEXT*** forment une file d'attente implicite.

La complexité en messages de cet algorithme est en moyenne de l'ordre de log(N).

2.2.7. Algorithme de MAEKAWA (1985)

Cet algorithme [MAE85] est basé sur l'approche à permission, mais propose une amélioration visant la réduction du nombre de messages échangés entre les nœuds du réseau en associant à chaque nœud i un ensemble S_i (quorum).Ce dernier, étant un sous-ensemble des nœuds du réseau, le nœud i ne demande la permission qu'aux nœuds j ($j \in S_i - \{i\}$). Dans le but d'assurer l'exclusion mutuelle, toute paire d'ensemble S_i, S_j ($i \neq j$) doit avoir au moins un nœud qui apparaît dans les deux ensembles. Ce dernier aura le rôle d'arbitre.

Principe de fonctionnement

- Lorsqu'un site i souhaite exécuter sa section critique, il diffuse sa requête ***REQUEST*** à tous les membres de son groupe S_i. Il pourra accéder à la section critique lorsque tous les nœuds du quorum S_i auront répondu.

- Lorsqu'un site j reçoit une requête, il la place dans sa file d'attente locale et ordonne celle-ci selon le principe d'estampillage. S'il n'a pas accordé de permission, alors il répond à i en lui envoyant un message d'accord ***LOCKED***. Autrement, il compare les estampilles des requêtes des sites k et i selon leur ancienneté. Si l'estampille du site k est plus ancienne que celle du site i, alors il répond défavorablement à i en lui envoyant un message ***FAILED***. Dans le cas contraire, le site j envoie un message de demande d'information ***INQUIRE*** à k afin de récupérer son accord de permission.

- Lorsque le site k reçoit le message ***INQUIRE***, il renvoie un message ***RELINQUISH*** s'il n'a pas réuni toutes les permissions requises (i.e. : a reçu au moins un message ***FAILED)***. Le site k se voit contraint d'annuler le ***LOCKED*** parvenu du nœud émetteur du message ***INQUIRE***. Autrement, le message reçu est ignoré (ainsi, un inter blocage circulaire est évité).

- Lorsqu'un site reçoit un message de libération ***RELEASE*** d'un site i, il récupère sa permission en supprimant la requête i de sa file locale et réordonne celle-ci en déterminant lequel est le prochain site qui recevra sa permission.

MAEKAWA a montré que la complexité de message de cet algorithme est de O(q) où q est la taille maximale d'un quorum.

2.3. Discussion

Dans cette partie, nous avons présenté quelques algorithmes d'exclusion mutuelle qui se basent sur différentes approches ainsi que leurs principales caractéristiques, en termes de mode de fonctionnement et du nombre de messages à échanger, dans le but d'achever la section critique.

Le but de cette présentation est d'introduire le problème d'exclusion mutuelle afin de simplifier la compréhension de la prochaine partie qui concernera l'exclusion mutuelle de groupe. Ce dernier est une généralisation de l'exclusion mutuelle classique, du fait que dans l'exclusion mutuelle on ne permet qu'à un nœud d'utiliser une ressource à moment donné alors que dans l'exclusion mutuelle de groupe plusieurs nœuds peuvent accéder à la ressource concurremment. La prochaine partie sera réservée à la présentation du problème d'exclusion mutuelle de groupe.

3. Exclusion mutuelle de groupe (EMG)

Le problème d'exclusion mutuelle de groupe a été introduit par Joung [Jou98] sous le nom du problème des *philosophes parlant d'une même voix* (le nom original étant *the congenial talking philosophers*). Le problème concerne un ensemble de *n* philosophes qui passent leur temps à penser seul et à parler dans un forum. Les philosophes ne parlent pas tous de la même chose. Il existe au total *m* sujets de réflexion. Etant donné qu'il n'y a qu'une seule salle disponible, un philosophe peut entrer dans cette salle si et seulement si l'une des deux conditions suivantes est vérifiée :

• La salle est vide (dans ce cas il démarre le forum)
• Le philosophe est intéressé par le même forum que celui en cours dans la salle (dans ce cas, il rejoint le forum).

Ainsi, à tout instant, soit la salle est vide, soit elle contient des philosophes qui parlent de la même chose.

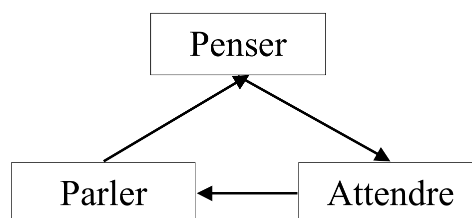


Figure 2.2 : Etats d'un philosophe et ses transitions.

Un forum est dit session, s'il y a au moins un philosophe dans ce dernier. Le problème posé est de concevoir un algorithme pour les philosophes qui répond aux exigences suivantes :

- ✓ Exclusion mutuelle : Si un philosophe est dans un forum, alors aucun autre philosophe ne peut être dans un forum différent simultanément.
- ✓ Attente bornée: Un philosophe essayant de s'occuper d'un forum réussira par la suite.

- ✓ L'accès concurrent : Si certains philosophes sont intéressés par une session et aucun philosophe n'est intéressé par un forum différent, alors les philosophes peuvent y accédés a la session concurremment.

Dans ce qui suit, nous allons présenter quelques algorithmes élaborés pour résoudre le problème d'exclusion mutuelle de groupe.

3.1. Les algorithmes courants d'exclusion mutuelle de groupe

Dans ce qui suit, nous allons présenter quelques algorithmes d'exclusion mutuelle de groupe se basant sur différentes approches (permission et jeton) et utilisant des topologies différentes.

3.1.1. Algorithme de Joung (1998)

Dans [Jou98], Joung présente deux algorithmes sur un graphe complet et sont basés sur une approche à permission. Chaque processus *p* possède un unique identificateur et peut communiquer avec tous ses voisins. Il modifie l'algorithme de Ricart et Agrawala pour résoudre le problème de l'exclusion mutuelle de groupe. Rappelons que dans cet algorithme, lorsqu'un nœud souhaite accéder à la section critique, il diffuse un message de demande à tous les autres processus et attend. Ce processus ne pourra accéder à la section critique que si tous les autres processus ont répondus.

L'idée principale de l'algorithme *RA*1 développé par Joung [Jou98] est qu'à chaque fois qu'un philosophe souhaite atteindre un forum *X*, il diffuse un message de demande à tous les autres philosophes. Il pourra accéder au lieu de discussion quand tous les philosophes auront répondus. Quand un philosophe reçoit un message de demande, soit il répond immédiatement au message s'il n'est intéressé par aucun forum où il est intéressé par un forum différent et sa priorité est inférieure à celle du message qu'il a reçu. Autrement, il détient la demande dans sa file jusqu'à ce qu'il ait fini le forum. Quand un philosophe quitte le forum, il libère toutes les demandes qu'il a retenu.

Dans l'algorithme *RA1*, on peut remarquer qu'un philosophe ne peut atteindre un forum que si sa priorité est supérieure à celles de tous les nœuds qui demandent un forum différent. Donc, même si un forum X est ouvert, il peut exister des philosophes qui demandent l'accès à X et qui ne peuvent pas le faire car les nœuds ayant une priorité supérieur et qui demande un forum différent ne répondent pas. Donc, il doit attendre que leurs demandes soient satisfaites.

Le nombre de messages par entrée en SC est 2*(N-1) ((N-1) messages ***REQUEST*** + (N-1) messages ***REPLY***). Donc, ce nombre est proportionnel à N, plus le nombre de nœuds dans le système augmente, plus le nombre de messages échangés pour accéder à la section critique augmente.

L'algorithme *RA*2 résout ce problème de la façon suivante : Tant qu'un philosophe P_i est dans une session X, s'il reçoit une demande d'un philosophe P_j pour la même session, alors P_i lui répond directement par un nouveau type de message (message ***START***), autorisant P_j à rentrer directement dans la session X. Donc le philosophe ayant la priorité la plus grande capture les autres processus. En revanche, les philosophe qui ont reçus la demande de P_j doivent être au

courant de son entrée à la session X et donc attendront seulement une réponse de P_i. De ce fait, quand P_i quitte la session, dans la réponse envoyée aux philosophes, P_i doit avertir qu'il a capturé P_j. Dans cet algorithme, une entrée en section critique nécessite entre n et $3(n\text{-}1)$ messages.

3.1.2. Algorithme de Wu et Joung (1999)

Dans [WJ99], Wu et Joung présentent deux algorithmes CTP-Ring1 et CTP-Ring2fonctionnant sur un anneau unidirectionnel. Chaque processus P_i possède un identifiant unique i et ne peut communiquer qu'avec son voisin p_i+1 (les opérations sur les identifiants sont *modulo n*).
De la même façon que l'algorithme de Joung sur le graphe complet, les auteurs modifient l'algorithme de Ricart et Agrawala pour résoudre le problème de l'exclusion mutuelle de groupe. L'algorithme présenté est adapté pour fonctionner sur un anneau unidirectionnel, il utilise les mêmes messages que l'algorithme sur le graphe complet.

A. Algorithme CTP-Ring1 (les philosophes qui discutent en accord)

L'idée principale de cet algorithme est que lorsqu'un philosophe souhaite atteindre un forum, il envoie sa demande au successeur. Lorsqu'un philosophe i reçoit une demande d'un philosophe j, si i est intéressé par le même forum, a une priorité inférieur à celle de j ou qu'il ne demande aucun forum, alors il envoie la demande de j au successeur, sinon i détient la demande de j jusqu'à ce qu'il sorte de la SC. Un philosophe ne peut accéder à la SC que s'il reçoit sa propre demande. Les détails de l'algorithme seront présentés ci-dessous. Dans cet algorithme chacun des philosophes ne peut être que dans l'un des trois états suivants :

Penser, signifiant qu'il n'est intéressé par aucun forum ;
Attente, signifiant qu'il attend un forum ;
Discuter, signifiant qu'il est dans un forum.

Chaque philosophe détient une variable SN qui représente son horloge logique locale selon le principe de Lamport [Lam78] et qui lui permet d'avoir une vue globale du réseau. Initialement, SN est égale à 0, à chaque fois qu'il reçoit un SN supérieur à son SN il le met à jour. De plus, quand un philosophe souhaite s'occuper d'un forum, il incrémente son SN de 1. Initialement, chaque philosophe est dans l'état penser.

La priorité d'un philosophe est assignée comme suit : Quand le philosophe envoie une demande $Req(<SN_j, j> ; X)$, il obtient $<SN_j, j>$ comme priorité et il la garde jusqu'à ce qu'il quitte le forum X. La priorité de p_i est supérieure à la priorité de p_j si et seulement si $SN_i>SN_j$ ou $SN_i=SN_j$ et $i<j$. L'auteur suppose que la priorité d'un philosophe p_i à l'état penser est $<i,\infty>$.Ainsi, un philosophe à l'état penser à toujours une priorité inférieure à celle d'un philosophe intéressé par un forum.

<u>*Principe de fonctionnement*</u>

- Quand un philosophe p_i souhaite s'occuper d'un forum X, il se met à l'état attente, il incrémente son SN de 1 et envoie un message de demande $Req(<SN_i, i>; X)$ à son successeur p_i+1, où le SN_i représente la nouvelle valeur de SN de p_i. Le philosophe p_i reste dans l'état *attente* jusqu'à ce qu'il reçois sa demande $Req(<SN_i, i> ; X)$. Dans ce cas,

p_i assiste au forum *X*.

- Quand p_i+1 reçoit un message de demande ***Req(<SN$_j$, j> ; X)*** de p_i, il envoie la demande à son successeur p_i+2 s'il est dans l'état *penser*, il est également intéressé par X, ou il est intéressé par un forum différent et a une priorité inférieure à celle de p_j. Autrement, il la détient dans sa file d'attente.
- Quand p_i quitte le forum, il libère toutes les demandes qui sont dans sa file d'attente et il revient à l'état *penser*.

Le nombre de messages nécessaires pour un philosophe afin d'atteindre le forum est N messages.

Dans CTP-Ring1, on peut remarquer qu'il y a une basse simultanéité qui est due au fait que, si deux philosophes p_i et p_j sont intéressés par le même forum et il y a un troisième philosophe p_k intéressé par un forum différent et a une priorité inférieure à celle de p_i et supérieure à celle de p_j, p_i et p_j ne peuvent pas s'occuper du même forum concurremment parce que le philosophe p_i de priorité inférieure doit attendre que p_k ait fini son forum avant qu'il puisse s'occuper d'un forum. Pour cela, une autre solution a été proposée pour le problème qui consiste à ce que p_j envoie un autre message pour "capturer" p_i (et les autres philosophes intéressés également par le même forum) de sorte que les philosophes capturés puissent s'occuper du forum en même temps que p_j. Cette solution est une variante de l'algorithme CTP Ring1 et est présentée ci-dessous.

B. Algorithme CTP Ring2

Afin d'améliorer l'algorithme CTP-Ring1 et de permettre plus de simultanéité, un nouvel état a été introduit. Cet état est l'état *vérification* qui permet de vérifier s'il y'a des philosophes qui demandent la même session que celle actuellement ouverte. Comme précédemment, la priorité d'un philosophe est $<SN_i, i>$ qui est établie lorsqu'il émet une demande ***Req(<SN$_i$, i> ; X).*** Cependant, cette priorité peut "être augmentée" (mais tout au plus une fois) avant que le philosophe s'occupe du forum *X*. La priorité est remise à une valeur minimale $<i,\infty>$ quand le philosophe revient à l'état *penser*.

Principe de fonctionnement

- Quand un philosophe p_i souhaite s'occuper d'un forum *X*, il se met à l'état *attente*, il incrémente son *SN* et envoie un message de demande ***Req(<i, SN$_i$>; X)*** à son successeur p_i+1.

- Quand un philosophe p_j reçoit la demande ***Req(<SN$_i$, i> ; X),*** il envoie le message à son successeur s'il est également intéressé par *X* ou a une priorité inférieure à $<i, SN_i>$. Autrement, le processus p_j détient la demande dans sa file d'attente jusqu'à ce qu'il ait fini le forum.

- Quand un philosophe p_i reçoit sa demande, il se met à l'état *vérification* et envoie un message de confirmation ***Confirm_C(<i, SN$_i$>,X)*** à son successeur p_i+1. Le but du message est de capturer les philosophes qui attendent *X* ainsi que pour s'assurer qu'aucun philosophe n'est encore dans un forum différent. Le philosophe p_i doit attendre jusqu'à ce qu'il reçoive le message qu'il a envoyé avant qu'il puisse assister à *X*.

- Quand un philosophe p_j reçoit ***Confirm_C(<i, SN$_i$>,X),*** il traite le message pareillement aux

messages de demandes. Le philosophe p_j envoie le message au successeur s'il est également intéressé par *X* ou a une priorité inférieure à $<i, SN_i>$. Autrement, le processus p_j détient le message. Notons que tous les messages (de demande et de confirmation) détenus par un philosophe sont libérés quand il finit un forum. De plus, si le philosophe p_j est également intéressé par *X*, il est dans l'état *attente* et a une priorité inférieure à $<i, SN_i>$ alors p_j est capturé par p_i. Dans ce cas, le processus p_j prend la priorité de p_i et met son état à *vérification* puis, envoie à son tour un message de confirmation ***Confirm(j, <i, SN$_i$>,X).*** À la différence du message de la confirmation de p_i, le message de confirmation de p_j à un seul objectif qui est de s'assurer qu'aucun philosophe n'est encore dans un forum différent. Par conséquent, quand un philosophe reçoit un message ***Confirm,*** il le traite pareillement à un message ***Confirm_C***, sauf que les messages ***Confirm*** ne permettent pas de capturer des philosophes. Ceci empêche le philosophe p_j de capturer p_i après que p_i ait assisté à *X* et ait fait une nouvelle demande pour assister à *X*, empêchant de ce fait p_i et p_j de se capturer à plusieurs reprises (qui pourrait également avoir comme conséquence une violation de l'exclusion mutuelle). Le philosophe p_i met son état à *parler* pour assister à *X* quand il reçoit son message de confirmation ***Confirm_C(<i, SN$_i$>,X)***. Après que p_i ait fini le forum *X,* il libère tous les messages qu'il a détenu à son successeur p_i+1 et revient à l'état *penser.* Dans ce cas, il est possible qu'il y'ait encore des philosophes qui ont été capturés par P_i dans *X* alors qu'un autre philosophe p_k, dont la priorité est supérieur à celle de ces philosophes, attend un forum différent *Y*. Par conséquent, lorsque P_i quitte *X* et il libère tous les messages tenus dans sa file d'attente, p_k va recevoir sa propre demande ***Req(<k, SN$_k$>,X)***. Le philosophe p_k ne peut pas directement accéder à *Y* puisqu'il doit lancer un message ***Confirm_C(<k, SN$_k$>,Y)*** qui lui permettra de s'assurer qu'aucun philosophe n'est dans une session différente ainsi que de capturer les philosophes intéressés par *Y*.

Discussion

On peut remarquer que dans l'algorithme CTP Ring1, le problème qu'on a cité pour l'algorithme proposé par Joung [Jou08] se pose. Du fait que lorsqu'un philosophe souhaite atteindre un forum, il envoie sa demande à son successeur. Lorsqu'un philosophe i reçoit une demande d'un philosophe j, i n'envoie la demande à son successeur que s'il est intéressé par le même forum, moins prioritaire ou ne demande pas l'accès à un forum. Donc, même si le forum que demande j est ouvert il ne pourra pas accéder si i est plus prioritaire et demande un forum différent. Ceci dit, un philosophe ne peut accéder à un forum que s'il est le plus prioritaire des philosophes qui demandent un forum différent. Par ailleurs, dans cet algorithme, le nombre de messages échangés par un philosophe pour atteindre un forum est N.

L'algorithme CTP Ring2, règle le problème qui a été posé pour CTP Ring1. Dans CTP Ring2, lorsqu'un philosophe reçoit sa demande, avant d'accéder au forum il envoie un deuxième message Confirm_C qui permettra à tous les philosophes qui demandent le même forum d'y accéder. De ce fait, lorsqu'un forum est ouvert tous les philosophes le demandant y accéderont simultanément. Le nombre de messages échangés par philosophe pour atteindre un forum dans cet algorithme est 2*N.

3.1.3. Algorithme de Ranganath et Neeraj (1999)

Cet algorithme [RN99] utilise le concept de groupe (quorum), se base sur l'approche à permission et est une extension de l'algorithme de Maekawa [MAE85].

Rappelons que dans l'algorithme de MEKAWA, lorsqu'un nœud souhaite accéder à la SC, il envoie un message de demande à tous les membres de son groupe et attend leurs réponses. Lorsqu'un nœud i reçoit un message de demande d'un nœud j, si i n'a pas encore donné son accord à un autre nœud alors il accorde la demande de j. Sinon et si la priorité de j est inférieure à la priorité du nœud auquel il a donné son accord, i répond à j par un message d'échec pour lui dire qu'il a déjà donné son accord. Autrement, le nœud i essaie de récupérer son accord en envoyant un message au nœud qui détient l'accord. Si le nœud interrogé à reçu au moins un message d'échec d'un autre nœud, alors il remet l'accord à i. Dans le cas contraire, il le garde jusqu'à ce qu'il quitte la SC. Lorsqu'un nœud quitte la section critique, il donne son accord à la demande la plus prioritaire dans sa file. Cette solution a été modifiée et adaptée au problème d'exclusion mutuelle de groupe. Après qu'un nœud ait verrouillé tous les membres de son groupe et avant d'accéder à la section critique, il invite les nœuds qui demandent le même forum que lui.

Principe de fonctionnement

- Lorsqu'un site *i* souhaite atteindre un forum, il diffuse sa requête ***REQUEST*** à tous les membres de son groupe S_i.
- Lorsqu'un site *i* sort de la section critique, il diffuse un message de libération ***RELEASE*** à tous les membres de son groupe S_i.
- Lorsqu'un site *j* reçoit une requête, il la place dans sa file d'attente locale et ordonne celle-ci selon le principe d'estampillage. S'il n'a pas accordé de permission, alors il répond à *i* en lui envoyant un message d'accord ***LOCKED***. Autrement (il a accordé sa permission à un site *k*), il compare les estampilles des requêtes des sites *k* et *i* selon leur ancienneté. Si l'estampille du site *k* est plus ancienne que celle du site *i*, alors il répond défavorablement à *i* en lui envoyant un message ***FAILED***. Dans le cas contraire, le site *j* envoie un message de demande d'information ***INQUIRE*** à *k* afin de récupérer son accord de permission.
- Quand un processus *i* reçoit un message ***LOCKED***, il stocke toutes les demandes qui ont été envoyées dans le message. Une fois que *i* a verrouillé tous ses membres de quorum (groupe), il envoie un message d'invitation ***INVITE*** aux processus qui ont fait ces demandes.
- Quand un processus *i* reçoit un message d'invitation ***INVITE***, il libère ses membres de quorum en leurs envoyant un message d'annulation ***CANCEL***.

- Un processus peut entrer dans un forum quand il verrouille tous ses membres de quorum ou quand il reçoit un message d'invitation ***INVITE*** d'un autre processus. Dans le premier cas, le processus entre dans le forum en tant que *leader*. Dans le deuxième, il entre comme *palpeur*.

- Quand un nœud reçoit un message annulation ***CANCEL*** d'un processus, enlève sa demande de la file d'attente, si elle est encore présente.

- Quand l'un des *palpeurs* quitte le forum, il envoie un message ***LEAVE*** au *leader*.

- Un leader maintient les verrous sur ses membres de quorum jusqu'à ce qu'il ait reçu un message ***LEAVE*** de tous ses palpeurs et ait lui-même quitté le forum. Il envoie alors un message ***RELEASE*** à ses membres de quorum.
- Lorsque le site k reçoit le message ***INQUIRE***, il renvoie un message ***RELINQUISH***. S'il n'a pas réuni toutes les permissions requises (i.e. : a reçu au moins un message ***FAILED***). Le site k se voit contraint d'annuler le ***LOCKED*** parvenu du nœud émetteur du message ***INQUIRE***. Autrement, le message reçu est ignoré (ainsi, un interblocage circulaire est évité).

- Lorsqu'un site reçoit un message de libération ***RELEASE*** d'un site i, il récupère sa permission en supprimant la requête i de sa file locale et réordonne celle-ci en déterminant le prochain site qui recevra sa permission.

Discussion

Dans cet algorithme le concept de quorum a été utilisé. La solution que proposent les auteurs permet à plusieurs processus de partager un forum, ce partage est le but de l'exclusion mutuelle de groupe. Lorsqu'un nœud atteint un forum, tous les membres de son groupe qui demandent le même forum que lui pourront y'accéder concurremment. Du fait que lorsqu'un nœud reçoit l'acquittement de tous les membres de son groupe, celui-ci invite les nœuds qui sont dans son groupe et qui demandent le même forum que lui. Par ailleurs, le nombre de messages échangés par nœud afin d'atteindre un forum est de l'ordre de O(q) où q est la taille maximale d'un quorum (le nombre de nœuds que contient le quorum).

3.1.4. Algorithme de JOFFRO, SEBASTIEN, DATTA et PETIT(2003)

Dans [JSDP03] les auteurs proposent deux algorithmes et utilisent une architecture en arbre. L'idée principale des algorithmes est que lorsqu'un nœud demande d'accéder à une session, celui-ci transmet la demande à la racine de l'arbre. Cette dernière diffuse l'information dans le réseau, le nœud pourra accéder lorsque tous les descendants auront répondus.

A. Algorithme EMG1

L'algorithme EMG1 résout le problème d'exclusion mutuelle de groupe en utilisant des phases. Les phases sont lancées par la racine de l'arbre en réponse aux demandes faites par des processus pour accéder à des sessions. Lorsqu'un processus *p* fait une demande, il enregistre *X* comme session demandée et envoie ce message de demande à son père. Le message est alors routé vers la racine à travers les processus qui sont entre le processus *p* et la racine.

Lorsque la racine reçoit le message de demande, elle lance (propage) la phase d'ouverture de la session *X*. Chaque processus recevant cette propagation enregistre *X* comme la session qui est actuellement ouverte. Tant qu'aucun processus ne demande d'accéder à une session différente de *X*, tous les processus souhaitant accéder à la session *X* pourront le faire concurremment autant de fois qu'ils le voudront et ceci se fera sans avoir à envoyer des messages additionnels puisque chaque processus sait que la session *X* est la session actuellement ouverte.

Quand un processus *q* souhaite accéder à une session $Y \neq X$, il envoie un message de demande de *Y* vers la racine. À la réception de ce message, la racine lance (propage) la deuxième phase qui est la phase de fermeture de la session *X*. Chaque processus atteint par la propagation de la phase de fermeture prend acte que la fermeture de la session est lancée jusqu'à ce qu'elle atteigne les feuilles de l'arbre. Les feuilles répondent alors par le message feedback. À la réception du feedback, si le processus n'est intéressé par aucune session, il renvoie le message à son père immédiatement. Autrement, si un processus *p* est en section critique quand il reçoit le message feedback, celui-ci garde le message jusqu'à ce qu'il quitte la section critique à ce moment, il transmettra le message à son père et il garde trace de la nouvelle session demandée. Quand la racine reçoit le feedback de tous ses fils, celle-ci sait qu'aucun processus n'est (ou peut entrer) dans la section critique. La racine lance alors l'ouverture de la session *Y*. Le processus ci-dessus est répété pour l'ouverture et la fermeture des sessions.

Dans la solution proposée, il est facile d'observer que les processus qui sont proches de la racine profitent plus de la session ouverte que les autres processus, du fait qu'ils sont les premiers qui reçoivent le message d'ouverture d'une session et les derniers qui reçoivent le message de fermeture d'une session.

B. Algorithme EMG2

L'ouverture d'une session dans EMG1 nécessite 3(n-1) + h (2(n-1), pour la fermeture d'une session, (n-1) pour l'ouverture de la session et h qui est la hauteur de l'arbre).Cet algorithme réduit le nombre de messages de 3 (n -1) + h à 4*h. L'idée principale employée par les auteurs pour réduire le nombre de messages est comme suit : Au lieu de diffuser le message d'ouverture et de fermeture dans le réseau entier (ce qui est fait dans l'algorithme EMG1), ils seront envoyés seulement vers les processus qui ont fait une demande.

Comme dans le premier algorithme, les auteurs supposent que la première demande de la session X est faite par le processus p, celle-ci est reçue par la couche application du processus. Le processus *p* insert le tuple <X, Ø> dans sa file d'attente (pour dire qu'il demande la session X). Lorsque *p* reçoit une autre demande de la même session X de l'un de ses descendants (par exemple *q*), *p* ajoutera *q* au tuple <X, Ø>, le tuple sera égale à <X, q>. Cette file est employée pour réduire le nombre de demandes d'ouverture et de fermeture de sessions. Chaque processus *p* enregistre la route des descendants qui ont fait des demandes d'une session (par exemple X) dans un ensemble (appelé NS). De ce fait, les messages d'ouverture et de fermeture seront envoyés "seulement" aux processus qui sont dans NS.

Lorsque *p* souhaite accéder à une session, il envoie le message ASK(X) à son père. Le message est alors expédié à la racine par l'intermédiaire des autres processus sur le chemin de *p* vers la

racine. Chaque processus sur ce chemin met à jour sa file d'attente. Quand la racine reçoit ASK(X), elle lance la session d'ouverture de la session X pour p, garde trace qu'une session est ouverte et envoie le message d'ouverture de session OS(X) aux descendants qui sont dans NS et qui ont demandés X. Lorsqu'un nœud reçoit le message OS(X) de son père, il applique les mêmes étapes que la racine et envoie le message aux processus qui demandent X et qui sont dans NS.

Lorsqu'un processus quitte la section critique, ce dernier ne connaît plus les descendants qui sont dans X (car il a supprimé l'entrée X de NS). Pour résoudre ce problème, chaque processus maintient un autre ensemble, OpenSet. Avant que *p* enlève le NS de sa file d'attente, il le copie dans OpenSet.

Lorsqu'un processus demande l'accès à une session $Y \neq X$, il lance la phase d'ouverture de Y comme expliqué précédemment. Lorsque la racine reçoit le message ASK(Y), elle lance la phase de fermeture de la session X et envoie le message CS aux processus qui sont dans OpenSet (semblable à l'envoi du message OS).

Lorsqu'un processus reçoit le message CS, il fait de même que la racine. Une fois que le message CS a atteint les nœuds feuilles de l'arbre, soit le nœud répond immédiatement par un message DONE s'il a quitté la section critique ou il attend jusqu'à ce qu'il ait fini avec la section critique. Lorsqu'un nœud a reçu un message DONNE de tous ses descendants qui ont demandés l'accès à X, il transmet à son tour ce message à son père. Lorsque la racine reçoit le message DONNE, la session X est fermée et donc le processus d'ouverture de Y peut être entamé (comme expliqué précédemment pour X).

Discussion

Comme expliqué précédemment, dans EMG1, le nombre de messages nécessaires pour atteindre un forum est de $3(n-1) + h$ (où 2(n-1) messages sont utilisés pour la fermeture d'une session, $(n-1)$ pour l'ouverture de la session et h est la hauteur de l'arbre).Ceci est dû au fait que lorsqu'un nœud souhaite atteindre une session, la racine propage l'information dans tout le réseau ce qui est couteux en nombre de messages. Pour cela, les auteurs ont proposé une deuxième variante qui réduit le nombre de messages échangés à $4*h$. Ce nombre a pu être réduit du fait qu'au lieu d'informer tout le réseau il informe seulement les nœuds concernés par la session ouverte.

3.1.5. Algorithme de CANTARELL, DATTA, PETIT et VILLAIN (2001)

L'algorithme développé dans [CDPV01] se base sur l'approche à jeton. Il est utilisé sur une topologie en anneau où circule un jeton.

Initialement, le premier processus p qui accède à la session X annonce aux autres processus que la session X est ouverte. Le processus p est considéré comme chef de la session courante. Si un autre processus demande une autre session Y, le processus p est responsable de la fermeture de la session X et de l'ouverture de la prochaine session Y et donc un nouveau chef est choisi pour la session Y.

L'algorithme EMG se compose de trois sections. La section d'entrée qui est exécutée lorsqu'un processus demande l'accès à une section critique. La section de sortie qui est exécutée quand un processus décide de quitter la section critique. La troisième section est utilisée pour la gestion du jeton.

Initialement, un des processus détient le jeton et le fait circuler dans le réseau. Quand un processus p reçoit le jeton et s'il demande la session X celui-là va être considéré comme chef de la session X et annonce à tous les autres processus que la session X est ouverte en leurs envoyant le message TOKEN(X, Ø).

Tant qu'aucun processus ne demande une session différente, le jeton circule dans l'anneau et reste dans le même état (TOKEN(X, Ø)).

À la réception du jeton, les processus stockent localement l'identificateur de la session actuellement ouverte(X). Ainsi, tous les processus demandant la session X pourront accéder concurremment. Si un autre processus q demande d'accéder à une session Y différente de X et que aucun autre processus ne demande une session Z différente de X et de Y, dans ce cas le message TOKEN circule avec le premier champ égale à X et le deuxième à vide. Lorsque q reçoit le jeton, il change le deuxième paramètre et il le met à Y, de ce fait les processus qui vont recevoir ce message comprendront que la session Y est demandée et qu'elle est la prochaine qui va s'ouvrir. Quand le chef de la session X reçoit le message TOKEN(X, Y) il met à jour ce message s'il a quitté la SC et envoie de nouveau le message TOKEN(Ø, Y). Quand un processus reçoit le message TOKEN(Ø, Y), s'il est dans la session X, il détient le message jusqu'à ce qu'il ait fini avec la session X. Sinon, il le renvoie à son successeur.

Quand le chef de la session X (p) reçoit le message TOKEN(Ø, Y), il met à jour ces variables et déclare que la session X est fermée et que la session Y est ouverte, le processus p envoie alors le message TOKEN(Y, Y) et un nouveau chef est choisi pour la session Y. Le même processus est utilisé pour l'ouverture et la fermeture des autres sessions.

Discussion

On peut remarquer que le fonctionnement de cet algorithme ressemble un peu à celui de l'algorithme précédent. La différence entre les deux est que dans le premier utilise une architecture de réseau en arbre qui se base sur l'approche à permission et dans le deuxième une architecture en anneau qui se base sur l'approche à jeton. Le nombre de messages nécessaires pour l'accès à une session est 2*N.

3.2. Exclusion mutuelle de groupe dans les réseaux mobiles ad hoc

3.2.1. Algorithme de jiang (2002)

Dans [Jian02], Jiang propose une solution pour le problème d'exclusion mutuelle de groupe dans les réseaux mobiles ad hoc. L'algorithme se base sur l'approche à jeton et est une adaptation

de l'algorithme *RL* [WWV01] qui utilise le concept du graphe acyclique direct ***(DAG)*** sur le réseau. Ce dernier, change dynamiquement sa topologie logique pour s'adapter aux changements physiques de la topologie des réseaux mobiles ad hoc. Un nœud n_0, initialement porteur du jeton, lance la procédure de construction du ***DAG***, met sa taille à 0 et émet un message d'initialisation vers ses voisins. Chacun de ses voisins recevant ce message incrémente sa taille, transmet le message d'initialisation vers ses voisins et inverse son lien pour pointer vers le nœud initiateur (ou vers le nœud qui lui a transmis le message d'initialisation et qui possède la plus petite taille). Ainsi, le nœud possédant la plus petite taille est porteur du jeton et devient un puits (i.e. : tous les liens se dirigent vers lui).
Comme dans [GB81], cet algorithme utilise aussi la technique d'inversement partielle(ou *Partial Reversal Technique)* pour maintenir la circulation du jeton orientée ***DAG*** avec une destination dynamique. Chaque nœud maintient une file d'attente qui contient les identificateurs des nœuds voisins qui ont émis un message ***REQUEST*** vers lui.

L'algorithme considère les propriétés suivantes sur les nœuds mobiles et le réseau :

1. Les nœuds ont des identificateurs uniques,
2. Les nœuds ne tombent pas en panne,
3. Les communications de liens sont bidirectionnelles et *FIFO*,
4. Chaque nœud est averti de l'ensemble des nœuds avec lequel il peut directement communiquer en fournissant des indications de formations ou de pannes de liens,
5. Les pannes de liens naissantes sont détectées, fournissant ainsi une communication fiable,
6. Le partitionnement du réseau ne se produit pas,

Dans l'algorithme, quand un nœud veut accéder à une ressource partagée, il envoie une requête le long d'un lien de communication. L'algorithme utilise une relation d'ordre total sur les nœuds de sorte que le nœud de plus petit ordre soit toujours celui qui possède le jeton. Chaque nœud choisit dynamiquement son nœud voisin de plus petit ordre comme son lien sortant du détenteur du jeton.

Les structures de données de cet algorithme est la suivante :

- *state :* indique si un nœud est dans l'un des états suivants : en attente : *ES*, en SC : *SC* ou bien au repos : *NSC*. Initialement, *status*=*NSC*.

- *N :* l'ensemble de tous les nœuds en contact physique avec le nœud *i*. Initialement, *N* contient tous les nœuds voisins de *i*.

- *height :* un triplet(h_1, h_2, i) représentant la taille (l'élévation) du nœud *i*. Ce triplet définit deux niveaux de références h_1 eth_2 (h_1,h_2 étant des entiers). Si par exemple, i possède la taille (0, 2, i) et qu'un autre nœud j possède une taille (0,3,j), alors le lien qui unit ces deux nœuds est dirigé de j vers i (de celui possédant la plus grande taille vers celui possédant la plus petite taille).

- *hVector :* un tableau de triplets qui contient les tailles des nœuds voisins de i. Initialement, nous avons :*hVector*[j]=$height_j$ pour tout j voisin de i. Si le lien entre *i* et *j* est *entrant* au nœud *i,* alors *hVector*[j] >$height_i$, si le lien est *sortant* du nœud *i,* alors *hVector*[j] <$height_i$.

- *Leader :* un drapeau mis à True si un nœud possède le jeton et mis à False autrement. Initialement, *Leader*=True si i = 0, et *Leader*=False sinon.

- *next :* indique la position du jeton par rapport à la vue de *i*. Quand un nœud *i* possède le jeton, *next* = *i*, sinon *next* est un nœud d'un lien *sortant* de i. Initialement, *next* = 0 si i = 0, et *next* est nœud voisin (celui qui a la plus petite taille) *sortant* sinon.

- *weight :* représente le poids d'un nœud dans le réseau. Initialement, weight=0.

- *Q :* une file d'attente contenant les requêtes des nœuds voisins de i. Les opérations dans *Q* incluent :*empiler* (), qui met en file d'attente un nœud seulement s'il n'est pas déjà dans *Q*,*dépiler* () extrait le nœud tête de file avec une sémantique usuelle *FIFO*, et *supprimer* (), qui supprime un nœud de *Q*, sans se soucier de sa position. Initialement, *Q* = ∅.

- *ReceivedLI :* tableau de booléens, indiquant si le message *LinkInfo* est reçu du nœud *j*, j ayant récemment reçu le jeton. Une quelconque taille de j reçue avec *receivedLI*[*j*] =false ne sera pas prise en compte par i. Initialement, $receivedLI_i[j]$=True pour tout j ∈*N*(voisin de i).

- *forming*[*j*] : tableau de booléens mis à True si une formation de lien vers le nœud *j* a été détectée et mis à False après la réception d'un message *LinkInfo* de la part du nœud *j*. Initialement, $forming_i[j]$=False pour tout j ∈*N*(voisin de i).

- *formHeight*[*j*] : un tableau de triplets ayant la valeur de la taille du nœud *j*, j ∈*N*, quand le premier lien vers ce dernier est formé. Initialement, $formHeight_i[j]=height_i$ pour tout j ∈*N*.

Cet algorithme répond à des évènements tels que : la réception d'un message *Request* ou d`un *RELEASE,* la rupture ou la création d'un lien,…etc. Chaque message émis porte la valeur actuelle de la taille du nœud émetteur.

- Quand un nœud i souhaite accéder à une ressource S, il empile son identificateur dans sa file d'attente *Q* et met *state* à ES. Si le nœud i ne détient pas le jeton et qu'il y a un seul élément dans *Q,* alors il émet un message *Request(j, S)* vers *next* (où *next* est le voisin dont la taille set la plus petite par rapport aux autres voisins). Sinon (le nœud i détient le jeton), il met *weight* (poids) à 0, enlève sa requête de sa file Q et met *state* à CS. Si i reçoit une demande *Request(j, S)* d'un voisin j intéressé par la même ressource que lui pendant qu'il exécute sa section critique, alors ce dernier lui envoie un message *SUBTOKEN(S, w)* avec w=1. A chaque fois que le nœud i envoie un message *SUBTOKEN(S, w)*, il incrémente *weight* de 1. Il est à noter que les requêtes reçues par i concernant la ressource S ne vont pas être enregistrées dans sa file.
- Quand un nœud i quitte la section critique (S), alors s'il n'est détenteur du jeton il se contentera d'envoyer un message *Release(w)* où $w=weight_i$ à *next* et se mettra dans l'état NCS. Sinon, s'il détient le jeton alors si *weight* = 0, (i.e que tous les nœuds qui utilisent la ressource S ont fini de le faire) alors, il met *state* à NCS et appel la procédure *GiveTokenToNext ( ).*

- Quand un nœud i reçoit un message *Request(j, R)* d'un nœud j, le nœud i ignore ce message si *ReceivedLI[j]*= False. Sinon, il met à jour *hVector[j],* empile j dans la file d'attente si le lien est entrant (i.e. : de j vers i). Si la file n'est pas vide et que $state_i$= NCS, et qu'il possède le jeton, i appelle la procédure *GiveTokenToNext ( ).*Sinon (i ne possède pas le jeton), alors il appelle la procédure *RaiseHeight ( )* s'il ne possède aucun lien sortant, ou bien émet un message *Request(j, R)* si j est le seul élément dans *Q*, ou bien si *Q* n'est pas vide et que le lien vers *next* a été inversé.

- Quand un nœud i reçoit un message Release(w), alors s'il ne détient pas le jeton il envoie le message reçu à un de ses voisins. Sinon, il met $weight_i$ à $weight_i$-*w*, si cette différence est nulle alors met *state* à NCS et appel la procédure *GiveTokenToNext ( ).*

- Quand un nœud i reçoit un message *SUBTOKEN(S, w),*alors il fractionne w en w_1,w_2,…,w_i,….w_q (tel que q = |Q|) et envoie les messages *SUBTOKEN(S,* w_1*), SUBTOKEN(S,* w_2*),…, SUBTOKEN(S,* w_q*)* aux q demandeurs dans la file de i. De plus, si la demande de i pour la ressource S est dans Q_i, alors il met state à CS et *weight* à w_i. Dans le cas où la file du nœud i ne contient que sa demande pour une ressource R, tel que R≠S alors le nœud i émet un message *Release(w)* à *next* avec w = $weight_i$. les demandes concernées par la ressource R seront supprimées de la file de i.

- Quand un nœud i reçoit le jeton de la part d'un nœud j, il met *Leader* à true, puis il réduit sa taille (i.e. : $height_i(h_1, h_2, i) := height_i(h_1, h_2 - 1, i)$) pour qu' elle soit inférieure à celle de j, puis il informe tous les nœuds qui sont sur ses liens sortants de sa nouvelle taille en envoyant un message *LinkInfo*, puis il fait appel à la procédure *GiveTokenToNext ( ).* Le nœud i informe aussi le nœud j de sa nouvelle taille.

- Quand un nœud i reçoit le message d'information de lien *LinkInfo*() d'un nœud j, si *ReceivedLI*[*j*] = True, la taille de j est mise à jour. Si *ReceivedLI*[*j*] = false, i vérifie si la taille de *j* dans le message est égale à celle qu'il possède quand le jeton a été envoyé vers j. Si c'est le cas *ReceivedLI*[*j*] est mis à True.
  Si *forming*[*j*] = True, i compare sa taille avec la valeur dans *formHeight*[*j*], si elles sont différentes, alors un message *LinkInfo*() est émis vers j, l'identificateur de j est ajouté aux voisins de i et *forming*[*j*] := False. Si j est un élément de *Q* et qu'il est sur un lien sortant de i, alors j sera supprimé de la file. Si le nœud i ne possède aucun lien sortant et n'est pas porteur du jeton, il appelle la procédure *RaiseHeight ( )* pour créer ses liens sortants. Sinon, si *Q* n'est pas vide et que le lien vers *next* a été inversé, i émet un message *Request* vers l'un de ses voisins sur ces liens sortants ayant la plus petite taille.

- Si un nœud i détecte qu'un lien vers un nœud j est rompu, il supprime j de ses voisins, met *ReceivedLI*[*j*] à vrai, supprime j de la file s'il se trouve. Si i ne possède pas le jeton et qu'il ne possède aucun lien sortant alors, il appelle la procédure *RaiseHeight ( ),* pour en créer un. Si i ne possède pas le jeton et que *Q* n'est pas vide et que le lien vers *next* est rompu, i choisit parmi ses voisins sur ses liens sortants celui qui a la plus petite taille et lui émet un message *Request.*

- Quand un nœud *i* détecte la formation d'un nouveau lien avec *j*, i émet un message *LinkInfo*() à *j*, met *forming*[*j*] à True et *formingHeight*[*j*] à *height*.

- La procédure *GiveTokenToNext(),* est appelée pour la transmission du jeton, le nœud i dépile le nœud tête de file et le met dans *next*, si ce dernier est égale à i, alors i peut accéder à la section critique : il met *state* à CS et pour chaque demandeur dans sa file, il envoie un message *SUBTOKEN(R, w)* (q message seront envoyés, tel que q = |Q|) avec *w*=1 et met *weight à weight+q.* Les demandes concernant la ressource R seront supprimées de la file du nœud i.
  Sinon (*next* ≠ i), i réduit *hVector[next]* afin de diriger toutes nouvelles requêtes vers ce nœud, met *Leader* à false et *ReceivedLI*[*next*] à false, puis émet le jeton vers *next*. Si la file d'attente de i n'est pas vide, alors i émet un message *Request* vers *next.*

- La procédure *RaiseHeight()* est appelée quand le nœud i perd tous ses liens sortants et ne possède pas le jeton. Cette procédure crée de nouveaux liens sortants (en utilisant la technique d'inversement partiel [GAF81]), en supprimant son ancienne taille et en créant un nouveau niveau de référence. Le nœud i informe tous ses voisins de sa nouvelle taille. Tous les nœuds de sa file qui ont des liens sortants seront supprimés. Si *Q* n'est pas vide, i émet un message *Request* vers l'un de ses voisins sur ses liens sortants ayant la plus petite taille.

3.2.2. Algorithme de THIARE, NAIMI et GUEROUI (2006)

Comme dans l'algorithme proposé par Jiang[Jia02], l'algorithme [TNG06] se base sur l'approche à jeton et est une adaptation de l'algorithme *RL* [WWV01] qui utilise le concept du graphe acyclique direct ***(DAG)*** sur le réseau.
Le principe de fonctionnement de cet algorithme est similaire à celui de Jiang [Jia02], sauf pour la notion de poids (weight). Dans cet algorithme, les auteurs proposent d'introduire un compteur NUM qui comptera les nœuds en cours d'exécution d'une ressource R. quand un nœud i reçoit le jeton, s'il est destinataire alors il envoie des messages *SUBTOKEN* aux demeures de sa file et incrémente Num. Dès qu'un nœud quitte la ressource, ce dernier envoie un message de libération de ressources au nœud détenteur du jeton. Quand tous les nœuds auront quittés la ressource (Num = 0), le nœud i fait passer le jeton au suivant et ainsi de suite.

3.2.3. Conclusion

Dans cette partie, nous avons présenté différents algorithmes d'exclusion mutuelle de groupe se basant sur les deux approches (permission, jeton) et utilisant différentes topologies logiques de communication. Dans cette présentation, nous avons essayé d'expliquer le principe de fonctionnement des différents algorithmes ainsi que leurs principales caractéristiques. Dans la littérature, on a pu trouver un certain nombre d'algorithmes qui résolvent le problème d'exclusion

mutuelle de groupe dans les systèmes distribués dont on a présenté une partie. En contrepartie, on a trouvé que deux algorithmes qui résolvent le même problème dans les réseaux mobiles ad hoc. Les deux algorithmes présentés se basent sur l'approche à jeton et sont tous les deux une adaptation de l'algorithme *RL* et maintiennent un graphe acyclique direct ***(DAG)*** sur les liens physiques du réseau dans toute l'exécution d'algorithme.

Ces algorithmes ont été présentés dans le but de préparer la prochaine partie qui concerne le problème de groupe k exclusion mutuelle. A la différence du problème d'exclusion mutuelle de groupe qui permet à tous les nœuds demandant une même ressource (le nombre étant illimité) d'y accéder concurremment tant qu'aucun nœud ne demande une ressource différente. Le problème de groupe k exclusion mutuelle limite le nombre d'accès simultanés à une ressource à k. Nous pouvons remarquer que les deux problèmes se rapprochent du fait que le groupe k exclusion mutuelle est une extension de l'exclusion mutuelle de groupe. Le prochain chapitre sera consacré au problème de groupe K exclusion mutuelle.

Chapitre 3 :

Groupe K Exclusion Mutuelle (GK-EM)

1. Introduction

Le problème de groupe k exclusion mutuelle (Gk-EM) est une extension du problème d'exclusion mutuelle de groupe (EMG).Dans ce dernier, le nombre de nœuds qui peuvent accéder concurremment à une ressource partagée n'est pas restreint. Si nous limitons l'accès concurrent à la ressource partagée par K nœuds, nous obtenons le problème de groupe k exclusion mutuelle (Gk-EM). Ce problème peut être illustré par l'exemple du CD-Jukebox. Dans cet exemple, plusieurs utilisateurs travaillant sur un projet et qui dispose d'une très grande base de données enregistrée sur un périphérique de mémoire secondaire tel qu'un CD juke-box. Lorsqu'un utilisateur veut accéder à un objet de données, ce dernier sera chargé dans le buffer du périphérique. Afin d'augmenter les performances, les données chargées ne seront pas effacées jusqu'à ce qu'un autre objet soit chargé. Donc, les utilisateurs voulant accéder à ces données, y accéderont d'une manière concurrente, les autres utilisateurs voulant accéder à d'autres données devront attendre que le buffer se libère. De manière générale, les utilisateurs qui ont les mêmes centres d'intérêt peuvent les partager entre eux d'une manière concurrente tandis que les autres en sont exclus. Donc si le CD-Jukebox peut être consulté par n'importe quel nombre de nœuds concurremment, alors la gestion d'accès au CD-Jukebox devient le problème de l'exclusion mutuelle de groupe EMG. Autrement, si le nombre d'accès au CD-Jukebox est limité à K accès, le problème devient comment gérer l'accès de k nœuds à la ressource partagée et donc un nouveau problème est introduit qui est le groupe k exclusion mutuelle (GK_ME).

Un algorithme de groupe k exclusion mutuelle doit satisfaire les conditions suivantes :

- Exclusion mutuelle (Sureté): au plus k nœuds peuvent exécuter simultanément une session donnée. De plus, si deux nœuds i, j sont entraine d'exécuter leur sessions alors $Session_i$=$Session_j$.

- Attente bornée (Vivacité): Si un nœud demande l'accès à une session, alors il accèdera finalement à sa session.

- k-accès concurrents (Efficacité): Si un certain nombre de nœuds demandent d'accéder à la même session tandis qu'aucun de ces nœuds ne demande une session différente, alors les nœuds pourront accéder à la session tant que le nombre de nœuds dans la ressource est inférieur à k.

Dans ce qui suit, nous allons introduire le problème de groupe k exclusion mutuelle dans les réseaux mobiles ad hoc, ensuite, nous présenterons quelques algorithmes de groupe k exclusion mutuelle.

2. Groupe k exclusion mutuelle dans les réseaux mobiles ad hoc

Les algorithmes distribués qui existent et fonctionnent dans les réseaux mobiles ad hoc, peuvent être divisés en deux classes :

- la classe des algorithmes distribués pour réseaux statiques qui fonctionnent sur une couche de routage. Celle-ci s'occupe de l'acheminement des messages sur le réseau sans fil ad hoc. Ces algorithmes cachent la nature mobile du réseau (Figure 3.1(a))
- la classe des algorithmes qui sont en contact direct avec la couche liaison du réseau, ils prennent en charge toutes les opérations d'acheminement des messages (découverte de la route, l'émission et la réception des messages, le maintien de la route, le traitement des ruptures des liens ,… etc.), ces algorithmes sont appelés **Mobility aware algorithms** (figure 3.1(b))

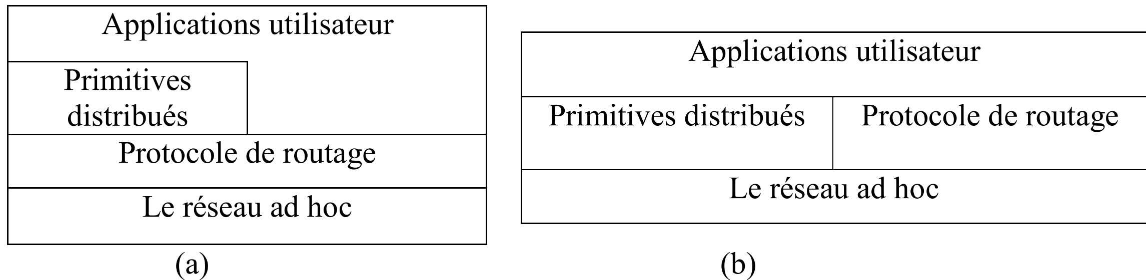


Figure 3.1 : Deux possibilités pour implémenter les primitives distribuées.

3. Algorithmes courants de groupe k exclusion mutuelle

Le problème de groupe k exclusion mutuelle est modestement abordé dans la littérature. Dans ce qui suit, nous présentons les solutions connues.

3.1 Algorithme de Jiang (2003)

Dans [Jia03], Jiang considère un système réparti qui se compose de n nœuds et m ressources partagées. Il suppose que chacun des nœuds est dans l'un des états suivants : soit le nœud ne demande pas la section critique (NCS), il demande d'entrer à la section critique (ES), il attend d'entrer à la section critique (WS), il est dans la section critique (CS) ou il est sorti de la

section critique (XS). Un nœud peut accéder à la ressource partagée seulement s'il est dans l'état section critique(CS).

Quand un nœud i souhaite accéder à une ressource partagée R_i, il change son état de NCS à ES et attend d'entrer dans CS. Un nœud se met dans l'état WS si le nombre de nœuds qui sont dans CS concurremment est supérieur à k. Quand le nœud i quitte la ressource partagée, il se déplace de CS à XS et finalement de XS à NCS.

Principe de fonctionnement

L'auteur suppose que l'algorithme s'exécute dans un système réparti composé de n nœuds (0, 1,...., n-1) et m ressources partagées (0, 1,.......m-1). Les nœuds du système forment un anneau de sorte qu'un nœud i ne peut envoyer des messages qu'à son successeur.

Les structures de données au niveau du nœud i :

- *TARGET* : est une variable locale qui contient la ressource que demande le nœud i. Initialement *TARGET* est égale à nulle.
- *STATE* : est une variable locale indiquant l'état du nœud, qui peut être *NCS*, *ES*, *WS*, *CS* ou *XS*. La valeur initiale de *STATE* est *NCS*.
- *PREEMPT* : est une variable locale indiquant si le nœud a pris la préemption (la priorité) d'un autre. La valeur initiale de *PREEMPT* est égale à faux.
- ***TOKEN(N, P, WN, WP, R, Q)*** : est un message jeton qui permet à un nœud d'accéder à la section critique. où R indique la ressource qui est actuellement consultée (R est nulle si aucune ressource n'est consultée), *Q* est une file d'attente des ressources demandées où Q.head renvoie le premier élément inséré dans Q, Q.dequeue() enlève Q.head de Q, et Q.enqueue(q) insère l'élément q en fin de la file Q, cette dernière est nulle si elle est vide. Les deux champs N et P représentent le nombre de nœuds qui accèdent à la ressource R. Le champ N compte le nombre de nœuds qui entrent dans CS sans la préemption (N représente "Non" préemption), alors que le champ P compte le nombre de nœuds qui entrent dans CS avec la préemption (P représente la "Préemption"). Les deux champs WN et WP représentent le nombre de nœuds qui attendent pour accéder à la ressource R quand il y a k nœuds dans la ressource R (W représente "attendre"). Le champ WN (resp. WP) compte le nombre de nœuds en attente qui accéderont plus tard à la ressource R sans (resp. avec) préemption. On dit qu'un nœud i est rentré à la section critique avec préemption s'il existe un nœud j tel que j demande une ressource différente de celle de i, j est plus prioritaire que i et i accède avant j(lorsque le nœud i avait reçu le jeton, la demande de j était dans Q).

Initialement, le message TOKEN(0, 0, 0, 0, nulle, nulle) est lancé dans le système.

- Quand un nœud i souhaite accéder à une ressource partagée X, il se met à l'état ES, et attend d'entrer en section critique.

- Quand un nœud i reçoit TOKEN(0, 0, 0, 0, nulle, nulle) (R=Nulle signifie qu'aucun nœud ne souhaite accéder à une ressource), le nœud j peut entrer en section critique et se met dans l'état CS pour accéder à la ressource partagée. Afin de permettre à d'autres nœuds d'entrer à

la ressource X concurremment, i envoie le message TOKEN(1, 0, 0, 0, X, nulle) à son successeur.

- Lorsqu'un nœud j demandeur de la ressource X, reçoit TOKEN(N, P, WN, WP, R, Q) avec (N+P) <k et TARGET=R=X, il incrémente N (j incrémente P au lieu de N si Q n'est pas vide), entre dans la section critique et envoie le message TOKEN. Cependant, si j reçoit le message TOKEN avec (N+P) ≥ k et TARGET=R=X, il ne peut pas accéder à la ressource immédiatement ce qui est dû à la limitation 'k-accès concurrent'. Dans ce cas, il se met dans l'état WS (attente de la ressource) et incrémente WN (j incrémente WP au lieu de WN si Q n'est pas vide) et envoie le jeton.

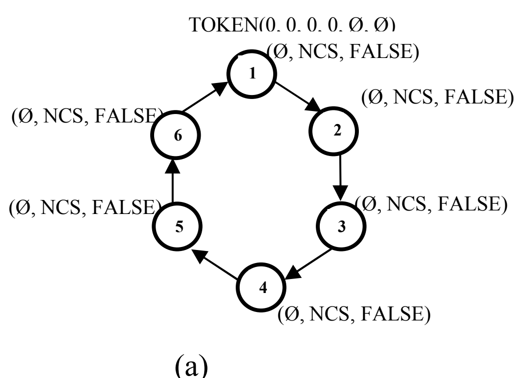
- Lorsqu'un nœud j reçoit TOKEN(N, P, WN, WP, R, Q) avec $TARGET_j \neq R$, il insert la ressource ($TARGET_i$) dans Q si elle ne l'est pas. Cependant, même quand TARGET=R, j doit l'insérer dans Q si elle ne l'est pas, Q≠Nulle et (N+WN)=0. Notons que Q≠Nulle signifie qu'il y a des nœuds qui demandent des ressources différentes de R et (N+WN)=0 veut dire que j ne peut pas entrer à CS par la préemption. Dans un tel cas, j doit attendre les nœuds qui demandent d'accéder à des ressources différentes dans Q.

- Quand un nœud quitte la section critique, il se met à l'état XS (sortie de la section) et attend la réception du jeton. Quand le message TOKEN(N, P, WN, WP, R, Q) est reçu, le nœud décrémente N (ou P si PREEMPT est à vrai) et vérifie si N+P+WN+WP=0, cela signifie que tous les nœuds qui souhaitaient accéder à R l'ont fait. Dans ce cas, le nœud met R à Q.head, exécute l'opération Q.dequeue () et envoie le jeton pour permettre aux nœuds d'accéder à la nouvelle ressource Q.head.

Illustration

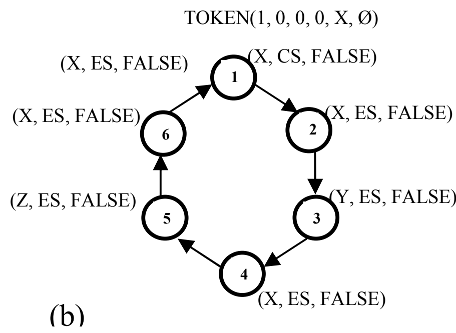
Ce principe peut être illustré par l'exemple suivant :
On propose un réseau de 6 nœuds, 3 ressources partagées (X, Y, Z) avec K égale à 3. Dans ce qui suit, pour chacun des nœuds on associera un triplet qui représentera respectivement les variables Target, State et Preempt.

A l'état initial, tous les nœuds sont dans l'état NCS, ils ne demandent aucune session et Preempt est à false comme le montre la figure 3.2.

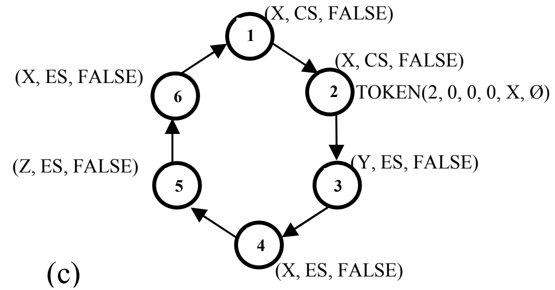


(a)

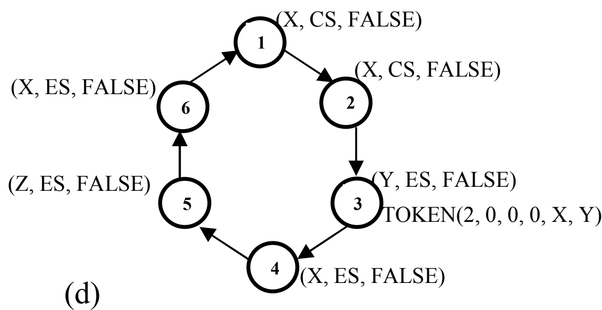
Si on prend le scénario suivant : Les nœuds 1et 2 demandent X, 3 demande Y, 4 demande X, 5 demande Z et 6 demande X. comme le nœud 1 détient initialement le jeton, il met son état à CS et fait circuler le message. Les autres nœuds vont mettre Target à la ressource qu'ils demandent et leurs états à ES pour dire qu'ils attendent d'entrer en section critique. Les figures de 3.2.(b) à 3.2.(n) représentent le scénario d'exécution de l'algorithme.



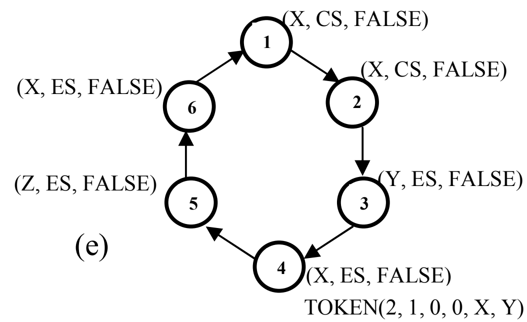
(b)



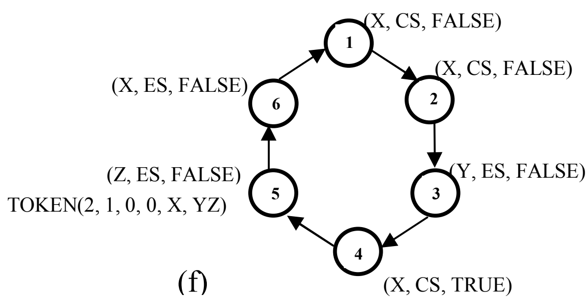
(c)



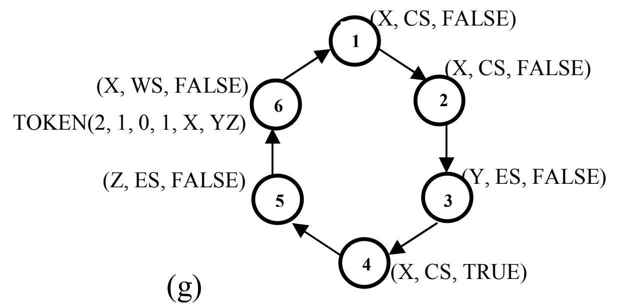
(d)



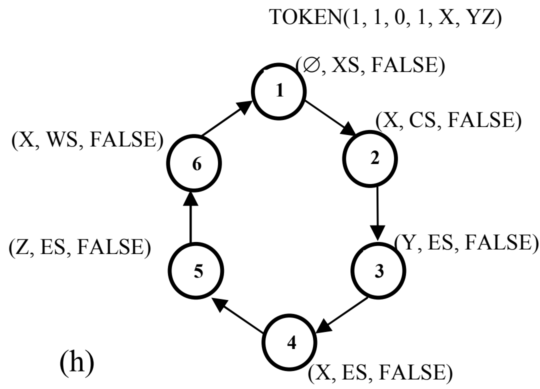
(e)

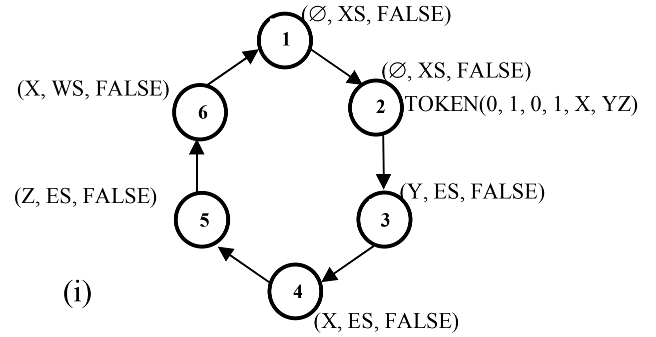


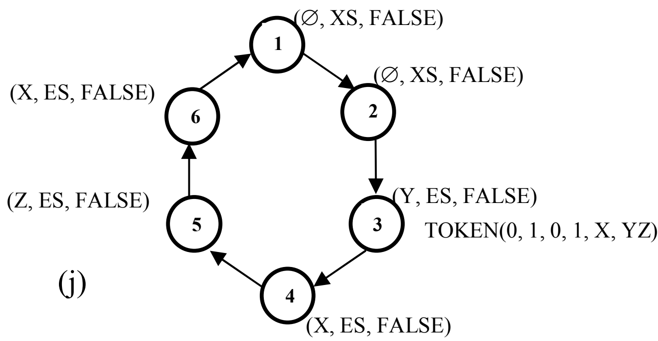
(f)

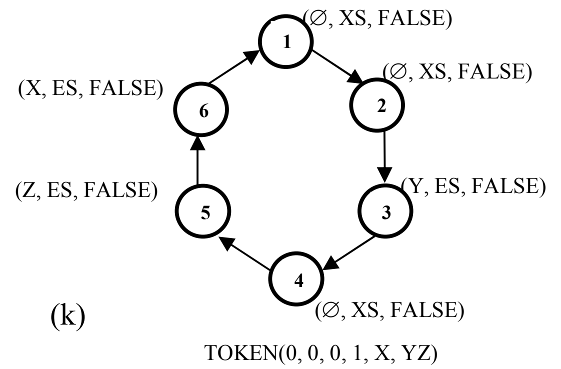


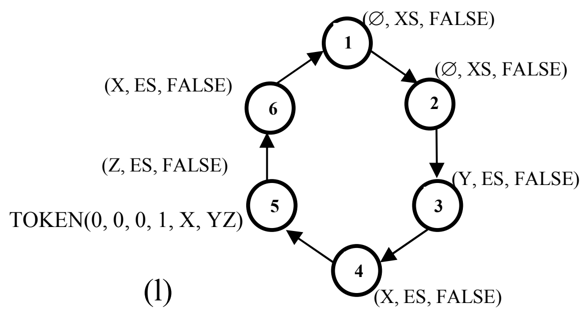
(g)

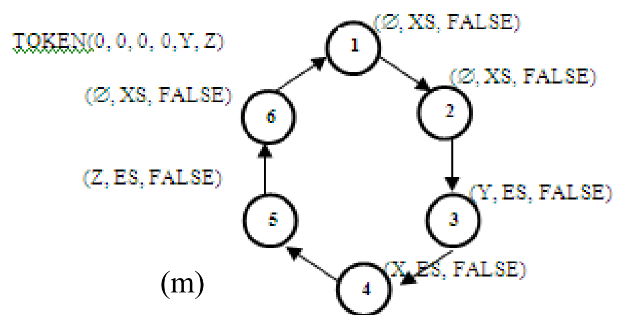

TOKEN(1, 1, 0, 1, X, YZ)
(∅, XS, FALSE)
(X, CS, FALSE)
(X, WS, FALSE)
(Z, ES, FALSE)
(Y, ES, FALSE)
(X, ES, FALSE)
(h)


(∅, XS, FALSE)
(X, WS, FALSE)
(∅, XS, FALSE)
TOKEN(0, 1, 0, 1, X, YZ)
(Z, ES, FALSE)
(Y, ES, FALSE)
(i)
(X, ES, FALSE)


(∅, XS, FALSE)
(∅, XS, FALSE)
(X, ES, FALSE)
(Z, ES, FALSE)
(Y, ES, FALSE)
TOKEN(0, 1, 0, 1, X, YZ)
(j)
(X, ES, FALSE)


(∅, XS, FALSE)
(∅, XS, FALSE)
(X, ES, FALSE)
(Z, ES, FALSE)
(Y, ES, FALSE)
(k)
(∅, XS, FALSE)
TOKEN(0, 0, 0, 1, X, YZ)


(∅, XS, FALSE)
(∅, XS, FALSE)
(X, ES, FALSE)
(Z, ES, FALSE)
(Y, ES, FALSE)
TOKEN(0, 0, 0, 1, X, YZ)
(l)
(X, ES, FALSE)


(∅, XS, FALSE)
TOKEN(0, 0, 0, 0, Y, Z)
(∅, XS, FALSE)
(∅, XS, FALSE)
(Z, ES, FALSE)
(Y, ES, FALSE)
(X, ES, FALSE)
(m)

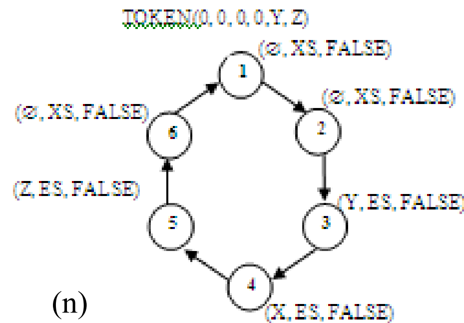


Figure 3.2 : Exemple d'illustration du fonctionnement de l'algorithme.

Lorsque le nœud 1 avait reçu le jeton, le champ R été à Ø (la section critique été vide), donc il a mis R à X il a incrémenté N, a envoyé le jeton au successeur puis il est rentré dans la session X.

Lorsque le nœud 2 avait reçu le jeton, R été égale à Target de plus N+P=1 <3 donc le nœud 2 à mis à jour ses variables ainsi que le jeton, il a envoyé le jeton au successeur et est rentré dans la session X.

Le nœud 3, en recevant le jeton avec R=X≠Target (la ressource utilisée actuellement diffère de celle qu'il demande), il a inséré sa demande dans la file Q et a envoyé le jeton au nœud 4.

Lorsque le jeton avait atteint le nœud 4, R été égale à X et égale à Target, N+P=2<3 mais la file Q n'était pas vide. Le nœud 4 est rentré par préemption à la session X, il a incrémenté P et a envoyé le jeton au nœud 5.

Lorsque le nœud 5 avait reçu le jeton R=X≠Target, le nœud a inséré sa demande dans la file Q et a envoyé le jeton au successeur.
Lorsque le jeton avait atteint le nœud 6, Target était égale à X=R mais N+P=4>3, il a incrémenté WP, il a mis son état à WS et a envoyé le jeton. Lorsque le nœud 1 avait reçu le jeton une deuxième fois, il avait quitté la session X, il décrémente N mais comme N+P+WP+WN≠0 (il y'a encore des processus dans X ou ils attendent d'accéder à X) le nœud 1 va seulement envoyer le jeton au successeur (il n'ouvre pas une nouvelle session). Le même processus est répété jusqu'à ce que tous les nœuds demandeurs arrivent à satisfaire leurs demandes

Discussion

L'algorithme que propose l'auteur, résout le problème de GK-EM en utilisant une topologie en anneau et en se basant sur l'approche à jeton. La solution qu'apporte l'auteur pour le problème GK-EM consiste à attendre le jeton, quand un nœud a le jeton et avant d'accéder à la section critique, il informe les autres nœuds afin de leurs permettre d'y accéder concurremment.

L'algorithme satisfait les trois propriétés de groupe K exclusion mutuelle : La propriété l'exclusion mutuelle : si deux nœuds i et j utilisent leurs ressources simultanément (R_i et R_j respectivement), alors $R_i=R_j$. Cette propriété est vérifiée car si un nœud i est dans la session X, R

va être égale à X. Donc, lorsque j va recevoir le jeton, s'il accède à X alors R_j=R_i=X. la propriété d'attente bornée : Cette propriété est vérifiée du fait que lorsque les nœuds quittent une session, le dernier d'entre eux met R à Q.head. Donc, chaque nœud réussira à accéder à la ressource qu'il demande. La troisième propriété : k-accès concurrent : lorsqu'un nœud reçoit le jeton avec Target = R, il pourra accéder tant que N+P≤K.

3.2 Algorithme de THIARE, NAIMI et GUEROUI (2009)

Le principe de fonctionnement de cet algorithme est similaire à l'algorithme qu'ont proposé les auteurs pour résoudre le problème d'exclusion mutuelle de groupe dans les réseaux mobiles ad hoc [TNG06]. Dans cet algorithme, les auteurs proposent d'introduire un compteur NUM qui comptera les nœuds en cours d'exécution d'une ressource R. quand un nœud i reçoit le jeton, s'il est destinataire alors il envoie des messages *SUBTOKEN* aux demandeurs de sa file sans dépasser le nombre k et incrémente Num. Dès qu'un nœud quitte la ressource, ce dernier envoie un message de libération de ressources au nœud détenteur du jeton. Quand tous les nœuds auront quittés la ressource (Num = 0), le nœud i fait passer le jeton au suivant et ainsi de suite.

4. Conclusion

Dans ce chapitre, nous avons défini le problème GK-EM et présenté quelques algorithmes. Nous avons dans un premier temps abordé l'algorithme de Jiang[Jia02] ainsi que ses principales caractéristiques, en terme de mode de fonctionnement. Cet algorithme se base sur une approche à jeton et utilise une topologie en anneau. Nous avons aussi présenté une solution au problème qui est dédiée aux réseaux mobile ad hoc. Cette dernière se base sur l'approche à jeton et est une adaptation de l'algorithme RL[WWV01].

L'objectif essentiel de ce travail est d'introduire le problème de groupe K exclusion mutuelle afin de pouvoir élaborer par la suite un protocole qui résout le problème groupe k exclusion mutuelle. Vu l'étroite relation entre ce problème et l'exclusion mutuelle ainsi que l'exclusion mutuelle de groupe, nous étions amené à présenter dans les parties précédentes ces deux problèmes.

Le prochain chapitre sera consacré à la présentation en détails de la solution qu'on propose pour résoudre le problème de groupe k exclusion mutuelle. Cette solution se base sur une topologie plate et sur une approche à jeton.

Chapitre 4 :

Le Protocole de groupe k exclusion mutuelle

1. Introduction

L'allocation de ressources est un problème fondamental dans les systèmes distribués. Dans ce cadre, nous nous intéressons au problème GK-EM (Group k Exclusion Mutuelle). Dans ce problème, un certain nombre de nœuds se concourent l'accès à un nombre fixe de ressources. Cependant, chacune des ressources ne peut être partagée qu'au plus par *k* nœuds simultanément et au plus une ressource est utilisée à un moment donné. De ce fait, le problème revient à gérer l'accès exclusif entre les ressources et l'accès simultané par un certain nombre de nœuds (au plus *k* nœuds) à une même ressource.

Les solutions aux problèmes classiques d'allocation de ressources (tels que : l'exclusion mutuelle et l'exclusion mutuelle de groupe) dans les systèmes distribués adoptent deux approches différentes, à savoir l'approche à jeton et l'approche à permission. On peut constater que l'adaptation de l'approche à permission à un réseau mobile ad hoc est très difficile à mettre en œuvre. Ceci est dû au fait, qu'elle repose soit sur la connaissance globale du réseau (approche à permission totale), soit sur des concepts mathématiques tels que les plans projectifs finis ou bien les quorums, qui mettent en évidence des ensembles de nœuds afin de représenter tout le réseau (approche à permission partielle). Le premier cas, est presque impossible à avoir vu que les arrivées et les départs des nœuds dans le réseau sont imprédictibles. Dans le second cas, vu la nature des réseaux mobiles ad hoc, la construction, la manipulation et la préservation de ces ensembles sont plus que difficiles. Donc, l'approche à jeton paraît comme la plus raisonnable à appliquer dans ce type de réseaux. Ce qui est le cas pour la plus part des solutions actuelles d'allocation de ressources dans le domaine des réseaux mobiles ad hoc.

La première solution au problème de GK-EM est due à Jiang [Jia02] et est définie pour les réseaux statiques. Sa solution repose sur une structure logique d'anneau imposée au réseau sur laquelle un jeton circule. L'adaptation de cette solution aux réseaux mobiles ad hoc nécessite l'entretien permanant de cette structure étant donnée les nouvelles caractéristiques introduites pour ce type de réseaux (telle que la mobilité). Par conséquent, cette solution ne peut s'adapter directement à un tel type de réseau.

La solution que nous proposons est dédiée aux réseaux mobiles ad hoc. Elle se base sur l'approche à jeton qui semble être l'approche la plus adéquate à un tel type de réseaux. Elle utilise une topologie plate du fait que le maintien d'une structure logique dans ce type de réseau est difficile et couteuse notamment dans le cas de fortes mobilités des nœuds. Elle se base sur la connaissance locale, ce qui favorise la scalabilité du réseau et ne se base pas sur la couche de routage. Par ailleurs, la solution prend en considération un certain nombre de caractéristiques d'un réseau mobile ad hoc entre autre la mobilité(les formations et ruptures de liens).

Le plan de ce chapitre est comme suit : Nous commencerons par donner une approche globale du protocole qui permettra de définir globalement quelques idées de bases sur notre protocole. Nous enchainerons par le principe de fonctionnement détaillé du protocole. Ensuite, nous présenterons

les structures ainsi que la description événementielle de la solution et nous terminerons par une conclusion.

2. Approche globale du protocole

Dans cette section, nous aborderons les éléments généraux de la solution. Le protocole que nous allons décrire dans ce chapitre se base sur l'approche à jeton et sur la connaissance locale, celle du voisinage immédiat et est indépendant de la couche de routage.

Quand un nœud souhaite utiliser une ressource (*session*), il se mettra à la recherche de l'acquisition du droit d'accès à celle-ci. Ce droit peut être l'obtention d'un jeton mais peut être aussi une invitation de la part d'un autre nœud qui a bénéficié d'un privilège d'accès à cette ressource, étant donné que les nœuds se concourent l'accès. Il ressort que le premier nœud qui accède à cette ressource (appelé *initiateur* de la session) est celui qui reçoit le jeton, les autres seront invités à celle-ci. Par conséquent, ce nœud initiateur gère l'ouverture, l'accès et la fermeture de cette session. Il doit donc garantir qu'au maximum *k* nœuds utilisent simultanément cette session.

Pour pouvoir inviter les différents nœuds intéressés par la même ressource, le nœud initiateur de la ressource doit avoir une connaissance des nœuds actuellement demandeurs des ressources. L'utilisation d'un même jeton pour en même temps accorder le droit d'accès et récolter les requêtes implique le transport de celles-ci avec leurs routes. L'inconvénient majeur est qu'on est amené à entretenir ces routes quand le jeton est en cours de route sans compter la masse d'information qui doit être transportée. Ce qui en résulte la nécessité d'introduction d'un mécanisme de récolte de requêtes. Il est donc intéressant d'introduire un autre jeton dont le rôle est de récolter les requêtes. Afin de partager la charge, celui-ci reste stationnaire pendant la récolte de requêtes. De ce fait, quand un nœud souhaite accéder à une ressource, sa demande doit atteindre le jeton qui se charge de récolter les requêtes. On peut distinguer alors l'existence de deux types de jetons dans le réseau. Le premier jeton permettra à un nœud d'initier une ressource et donc d'ouvrir une session. En d'autres termes, ce jeton donne le privilège à un nœud d'initier la session. Le second jeton permettra de récolter les différentes demandes. Par convention, le premier est appelé le jeton *distributeur* et le second, le jeton *récolteur*.

En effet, il est clair que les deux jetons ne fonctionnent pas indépendamment l'un de l'autre. Lorsqu'un nœud reçoit le jeton distributeur, il reçoit le droit d'initiation de la ressource et donc ce nœud sera l'initiateur de la session. Par conséquent, il devra inviter d'autres nœuds à utiliser cette ressource. Comme les demandes des nœuds sont inscrites dans le jeton récolteur, le nœud initiateur doit posséder ce jeton pour pouvoir inviter les autres nœuds. Ce qui implique que pour qu'un nœud puisse initier une session*, il devra disposer des deux jetons*. Une fois, le nœud a invité les demandeurs, il libérera le jeton récolteur vers l'un des nœuds demandeurs d'une autre ressource et il gardera le jeton distributeur jusqu'à la fermeture de cette session. Ainsi, un nœud obtenant le droit d'accès à une ressource suite à une invitation se contentera d'accéder à la session et d'envoyer un message de libération de ressource au nœud initiateur à la fin de l'utilisation de celle-ci.
Par conséquent, pour qu'un nœud puisse accéder à une ressource, il doit recevoir, soit les deux jetons récolteur et distributeur ; dans ce cas, le nœud est qualifié d'initiateur, soit il reçoit une invitation concernant la même ressource d'un nœud initiateur.

Suite à ces outils, plusieurs problèmes apparaissent. Parmi ces problèmes, on peut citer :

- Comment rechercher le jeton récolteur par un nœud demandeur de session, sachant que chacun des nœuds ne peut communiquer directement qu'avec ses voisins directs ?
- Quand un nœud demande une ressource, quels sont les nœuds qui doivent être informés?
- Comment inviter les demandeurs de la session en cours tout en évitant de dépasser le nombre *k* et en assurant que tout demandeur de cette session y accède au bout d'un temps fini?
- Comment acheminer le message de libération au nœud initiateur ?
- Comment savoir que tous les nœuds ont fini d'utiliser la ressource et conclure que la session est terminée ?
- Comment circulent les deux jetons et à quel moment?
- Comment sélectionner la prochaine session à ouvrir ?

La suite de ce chapitre a pour but de répondre à ces questions et à d'autres.

3. Principe de fonctionnement

Dans cette section, nous présentons le principe général de fonctionnement du protocole tout en répondant aux questions posées précédemment.

3.1 Circulation des jetons

Comme défini dans la section **2**, il existe deux types de jetons dans le réseau. Le jeton distributeur : Le rôle du porteur de ce dernier est l'initiation d'une session, la gestion d'accès des nœuds à celle-ci et la fermeture de cette session. Le second est le jeton récolteur : il se chargera de récolter les demandes des différents nœuds. Afin de satisfaire les requêtes des différents nœuds tout en évitant la famine, nous proposons d'estampiller les différentes requêtes, un numéro de séquence (*NS*) sera utilisé. Ce dernier est mis à jour à chaque fois qu'un nœud fait une demande ou reçoit une demande d'un autre nœud.
Lorsqu'une ressource est élue pour être utilisée, l'un des nœuds demandeurs de celle-ci sera destinataire du jeton récolteur. Dès que ce nœud reçoit ce jeton, il conclut que la prochaine session qui sera ouverte est celle qu'il demande. Puisque ce nœud ne pourra initier la session que s'il a les deux jetons, il attendra la réception du jeton distributeur pour pouvoir l'initier tout en continuant à attendre de nouvelles requêtes.
Dès que ce nœud reçoit le jeton distributeur, il pourra initier la session qu'il demande. Pour ce faire, le nœud met à jour le jeton récolteur en supprimant de ce dernier les requêtes satisfaites à partir du jeton distributeur, puis, il garde une copie dans sa file d'attente des demandes existantes dans le jeton récolteur concernant cette session. Dans un deuxième temps, il libère ce dernier en l'acheminant vers le prochain nœud qu'il choisit lui-même pour initier la prochaine session. Ensuite, il invite les demandeurs de cette session dont les requêtes existent au niveau local tout en gérant le nombre d'accès à la session. Dès la fermeture de la session en cours, le jeton distributeur va suivre la trace du jeton récolteur. Pour pouvoir suivre ce dernier, une trace de ce jeton doit être gardée au niveau de chaque nœud. A cet effet, à chaque fois qu'un nœud s'apprête à acheminer le jeton récolteur, il garde trace du nœud auquel ce jeton sera acheminé (par convention ce nœud sera le *next*).

Il est à noter que le choix de la prochaine session à ouvrir se fait sur la base des demandes existantes dans le jeton récolteur.
L'exemple de la figure 4.1 illustre la manière dont circulent les jetons.

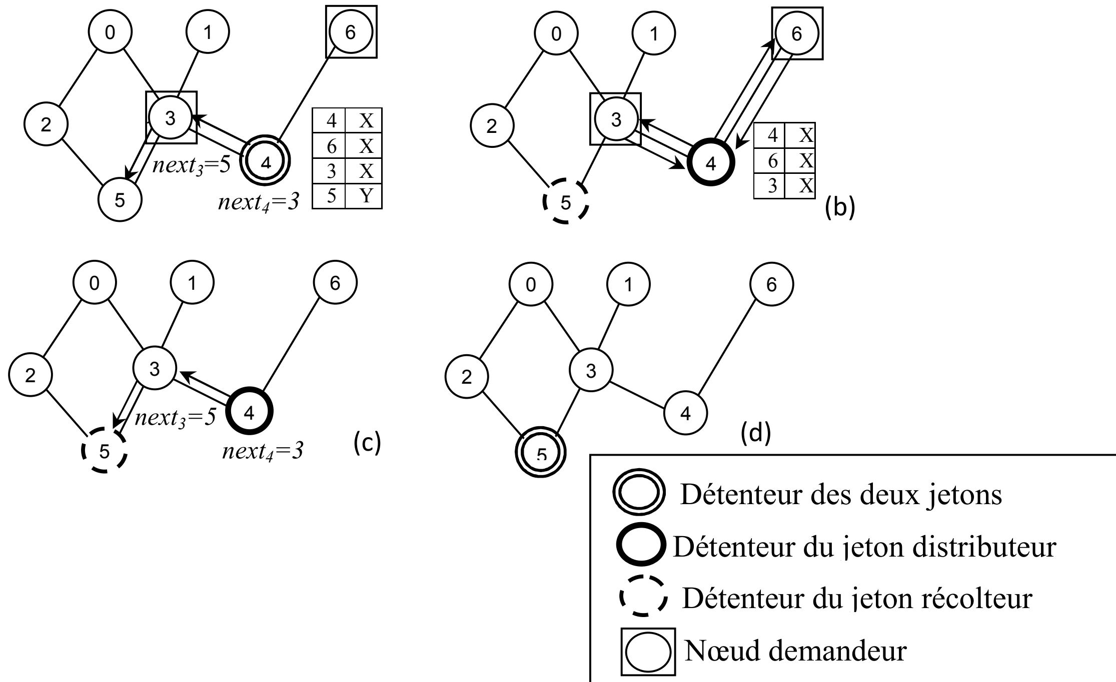


Figure 4.1 : Circulation des jetons.

Initialement (figure4.1.(a)), le nœud 4 détient les deux jetons récolteur et distributeur. Donc, ce nœud peut initier la session dont il est nécessairement demandeur. Ensuite, il met à jour le jeton récolteur en supprimant les requêtes satisfaites, garde trace des nœuds demandeurs de la même session dans sa file puis libère le jeton récolteur en l'acheminant vers le nœud 5 (le nœud 5 étant l'initiateur de la prochaine session élue) (figure 4.1.(b)) et invite les nœuds demandeurs de la même session (nœuds 6 et 3) à participer à l'utilisation de celle-ci. Compte au jeton distributeur le nœud va le garder jusqu'à fermeture de la session en cours. Dès que les nœuds invités par le nœud 4 (nœuds 6 et 3) auront quittés leur session et alors que le nœud 4 lui-même aura quitté sa session (la session sera fermée), le jeton distributeur suit la trace du jeton récolteur et donc il l'achemine vers le nœud 5 (figure 4.1.(c)). Une fois le nœud 5 acquière le jeton distributeur (figure 4.1.(d)), il pourra initier la session qu'il demande et inviter d'autres nœuds à participer à la session. Le jeton récolteur est envoyé à son prochain et ainsi de suite.

3.2 Recherche du chemin vers le détenteur du jeton récolteur par le nœud détenteur du jeton distributeur

Comme expliqué dans la section **3.1**, le jeton distributeur suit la trace du jeton récolteur. Dans certains cas, il s'avère que c'est impossible de suivre cette trace. Ceci est entre autres conséquence de la mobilité (changement de liens). Afin de parvenir à ce problème, un numéro de séquence sera transporté par le jeton récolteur (par convention on l'appellera *RNS*) et à chaque

fois qu'un nœud reçoit le jeton récolteur il garde trace de ce numéro de séquence. Si un nœud, devant transmettre un message à son *next*, détecte que ce dernier n'est plus son voisin alors il choisit aléatoirement l'un de ses voisins pour lui transmettre le jeton distributeur, en le recevant, ce nœud va comparer son *RNS* avec celui de l'émetteur, s'il est inférieur alors il conclut que l'émetteur a des informations plus récentes concernant le jeton récolteur et donc il lui retransmet le jeton distributeur pour qu'il puisse choisir un autre voisin et lui envoyer le jeton distributeur et ainsi de suite. Dans le cas où tous les voisins de ce nœud ont un RNS inférieur à celui du nœud émetteur alors il choisit parmi ses voisins celui qui a le RNS le plus grand et c'est à ce dernier de prendre en charge le jeton distributeur.

Reprenons l'exemple de la figure 4. 1. supposons par exemple qu'au moment d'acheminer le jeton distributeur à travers sa trace, le nœud 3 n'est plus voisin du nœud 4 (voir figure 4.1.(c)) comme le démontre la figure 4.2.
Dans la figure 4.1.(c) le *next* du nœud 4 est égale à 3 mais ces derniers ne sont plus voisins dans la figure 4.2.(a). Donc, le nœud 4 va choisir aléatoirement un de ses voisins pour lui acheminer le jeton distributeur tout en mettant à jour son next, dans ce cas c'est le nœud 6 qui est choisi. Le nœud 6 en recevant le jeton distributeur, il compare son *RNS* avec celui du nœud 4, il constate qu'il est inférieur alors il le remet à son émetteur qui est le nœud 4.

Détenteur des deux jetons
Détenteur du jeton distributeur
Détenteur du jeton récolteur

Figure 4.2 : Recherche du chemin vers le détenteur du jeton récolteur.

Le nœud 4 va choisir un autre voisin tout en mettant à jour son next, dans notre exemple c'est le nœud 1(figure 4.2.(b)). Ce dernier, en recevant le jeton, il constate que son RNS est supérieur à celui du nœud 4, ce dernier continue son acheminement vers l'un de ses nœuds voisins (le nœud 5), puisque son *next* = 3 n'est plus son voisin. Quand le nœud 5 reçoit le jeton distributeur, il initie sa ressource car il détient le jeton récolteur. Ce mécanisme consiste à une recherche arborescente améliorée.

3.3 Demande d'une session

Comme expliqué dans la section 5.2, lorsqu'un nœud demande une session, il suit la trace du jeton récolteur à travers un message de demande de session. La poursuite de ce dernier consiste à suivre son itinéraire tout en construisant la route qui mène à ce dernier. De ce fait, quand un

nœud demande une session, il achemine sa demande vers le nœud auquel le récolteur a été acheminé (le *next* du nœud) la dernière fois. De la même manière, à chaque fois qu'un nœud reçoit une demande d'un autre, s'il n'est pas détenteur du jeton récolteur, il achemine la requête au *next*. Autrement (i.e : détenteur du jeton récolteur), il enregistre la demande dans le jeton récolteur.
Il est à noter que seul le nœud détenteur du jeton récolteur traite la requête, les nœuds intermédiaires se contenteront d'acheminer la requête jusqu'à ce qu'elle atteint le détenteur du jeton récolteur.
La figure 4.3 donne un exemple de recherche du jeton récolteur. Le nœud 0 demande l'accès à une session X (figure 4.3.(a)), comme le prochain auquel la requête doit être acheminée est le nœud 3 (*next*=3), le nœud 0 achemine sa demande vers le nœud 3. Quand ce dernier reçoit la requête du nœud 0 (figure 4.3.(b)) et puisqu'il n'est pas détenteur du jeton récolteur, il achemine la demande vers le nœud 6 (*next* = 6). Dès que cette dernière atteint le nœud 6, elle sera insérée dans le jeton récolteur.

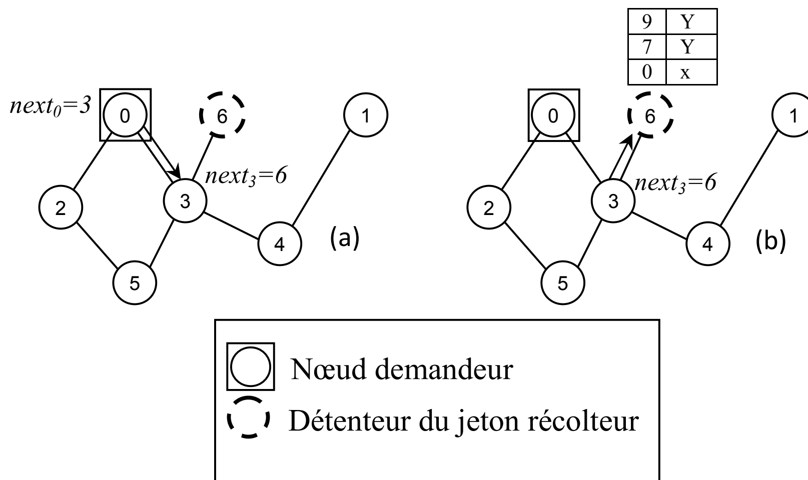


Figure 4.3 : Demande d'une session.

3.4 Recherche de chemin vers le détenteur du jeton récolteur par un nœud demandeur

Comme la mobilité est une caractéristique principale d'un réseau mobile ad hoc, il ressort que l'acheminement des différents messages, la mise à jour des routes (maintenir la meilleure route) ainsi que l'optimisation du nombre de messages circulant dans le réseau deviennent difficiles.
A cet effet, dans le cas où le *next* d'un nœud n'est plus son voisin (trace perdue), le nœud va chercher le chemin vers le nœud détenteur du jeton récolteur. Pour ce faire, il va envoyer la requête à l'un de ses nœuds voisins tout en construisant la route qui mène vers le détenteur du jeton récolteur. Dès que ce voisin reçoit la requête, il compare son *RNS* avec celui de l'émetteur. S'il est inférieur, alors il renvoie la demande au nœud émetteur et c'est à ce dernier de choisir un autre voisin pour lui envoyer la demande. Sinon (RNS du nœud récepteur est supérieur à celui du nœud émetteur), à son tour, il choisira un autre voisin pour lui envoyer la requête. Cette dernière

va être traitée de la même manière jusqu'à ce qu'elle atteint le détenteur du jeton récolteur. Le fonctionnement est illustré par la figure 4.4.

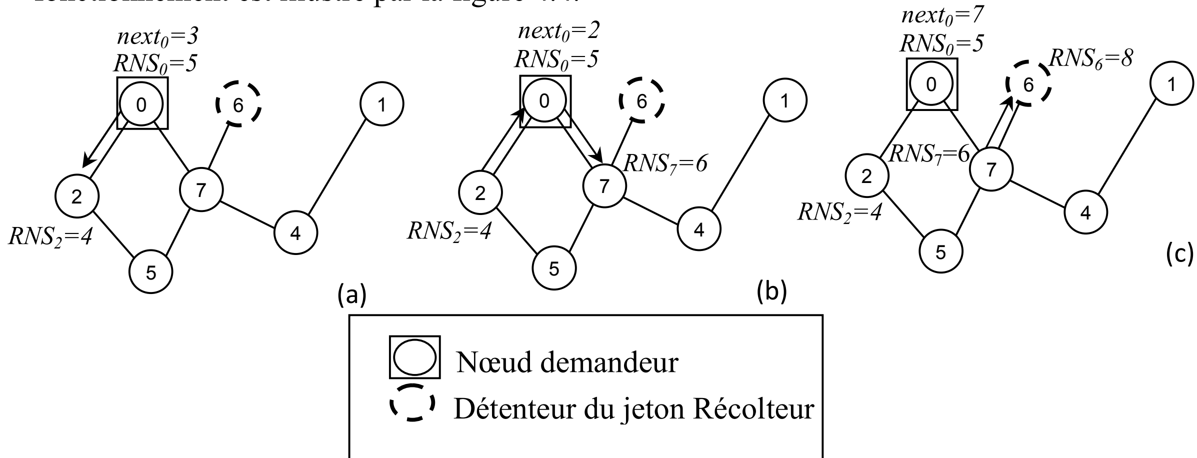


Figure 4.4 : Recherche du chemin vers le nœud détenteur du jeton récolteur.

Dans la figure 4.4.(a), le nœud 0 est demandeur, son *next* est égale à 3 mais ce dernier n'est plus son voisin (trace perdue). De ce fait, le nœud 0 choisit aléatoirement un de ses voisins (soit le nœud 2) pour lui envoyer sa demande. Lorsque le nœud 2 reçoit la demande du nœud 0 (figure 4.4.(b)), il constate que son *RNS* est inférieur à celui du nœud 0 et lui renvoie sa demande. Le nœud 0, en recevant sa demande, il conclut qu'il faut choisir un autre nœud voisin et donc envoie sa demande au nœud 7. Ce dernier, lorsqu'il reçoit la demande du nœud 0, l'achemine vers l'un de ses voisins (puisque le *RNS* du nœud 7 est supérieur à celui du nœud 0), dans l'exemple c'est le nœud 6. Lorsque ce dernier reçoit la demande (figure 4.4.(c)), il l'insert dans le jeton récolteur.

3.5 Libération de la section critique

Parmi les problèmes cruciaux dans la GK-EM, on peut citer ceux de la gestion de l'ouverture et la fermeture d'une session (le début de l'utilisation de la ressource et la fin de son utilisation par les nœuds concernés). Le fait d'être dans un environnement mobile, induit des changements de liens fréquents vu que la mobilité est l'une des caractéristiques principales d'un tel réseau. D'autre part, un nœud n'a des informations que sur son voisinage immédiat et comme ces informations changent fréquemment, la communication entre les nœuds devient difficile.
Quand un nœud quitte sa session, s'il n'est pas initiateur de celle-ci, il se contentera d'acheminer un message de libération au nœud initiateur et ce à travers la route avec laquelle il a été invité.
Lorsque le nœud initiateur aura reçu un message de libération à partir de tous les utilisateurs, il conclut que tous les nœuds ont fini d'utiliser la ressource. Si lui aussi a terminé d'utiliser celle-ci, il fait suivre le jeton distributeur la trace du jeton récolteur et ainsi permettre à la prochaine session d'être initiée.
Dans le cas où le nœud initiateur ne reçoit pas tous les messages de libération après un certain temps, il conclut qu'il y'a eu un échec dans l'acheminement d'un ou de plusieurs messages de libération et déclare que la session est fermée.
Dans la figure 4.5.(a), les nœuds 0, 5 et 6 sont en cours d'utilisation de la session X. Lorsque les nœuds auront fini d'utiliser la session, ils envoient un message de libération concernant la session X au nœud 4 à travers la route avec laquelle ils ont été invités.

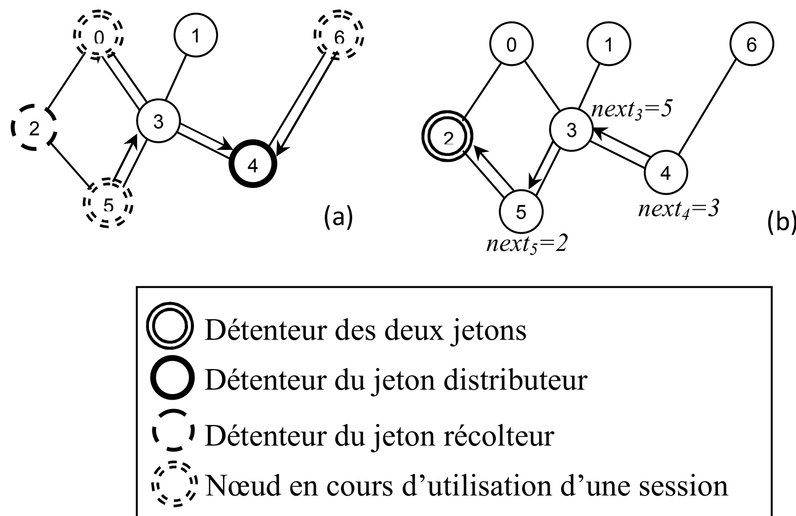


Figure 4.5 : Libération d'une session.

Quand le nœud 4 aura reçu les messages de libération à partir de tous les utilisateurs et lui-même aura fini d'utiliser la session X (figure 4.5.(b)), il conclut que la session est fermée et achemine le jeton distributeur au nœud 2 (nœud détenteur du jeton récolteur). Ce dernier en recevant le distributeur, il initie la session qu'il demande et libère le jeton récolteur en l'acheminant vers le prochain initiateur et ainsi de suite.

3.6 Recherche du chemin vers le nœud détenteur du jeton distributeur

Comme expliqué dans la section 3.5, quand un nœud quitte sa session, il achemine un message de libération de session au nœud initiateur (détenteur du jeton distributeur) à travers la route avec laquelle il a été invité. Dans le cas où la route est brisée, le nœud va tenter de retrouver le chemin vers le nœud initiateur. Pour ce faire, il vérifie si l'initiateur est l'un de ses voisins. Si c'est le cas, il lui envoie le message de libération directement. Sinon, il vérifie de la même manière, les autres nœuds de la route. Dans le cas échéant, il constate qu'il y'a eu échec dans l'acheminement du message de libération et ignore ce message.

Rappelons qu'à chaque ouverture d'une session un timer est déclenché. Dès la fin de ce dernier, la session sera fermée et le jeton distributeur sera acheminé vers le nœud élu pour initier la prochaine session élue. De ce fait, même si un message de libération est ignoré la session sera fermée.

Dans la figure 4.6.(a), les nœuds 0, 5, 6 sont en cours d'utilisation de leur session. Dès la fin de l'utilisation, les nœuds 5 et 6 acheminent les messages de libération à travers la route par laquelle ils ont été invités. Le nœud 0, en acheminant le message de libération, il constate que *next* de la route (nœud 3) n'est plus un voisin (voir figure 4.6.(b)). Pour ce faire, il vérifie si le dernier de la route (initiateur) n'est pas un de ses voisins, il constate qu'il ne l'est pas, il fait la même chose avec le nœud 7 et il constate que c'est un voisin et donc il met à jour la route et envoie le message au nœud 7 qui lui-même le transmet au nœud 4. Le nœud 4 en recevant le message de libération

du nœud 0 (figure 4.6.(c)), il ferme la session puisque il a déjà reçu les messages de libérations des autres nœuds utilisateurs de la session et fait suivre le jeton distributeur la trace du jeton récolteur.

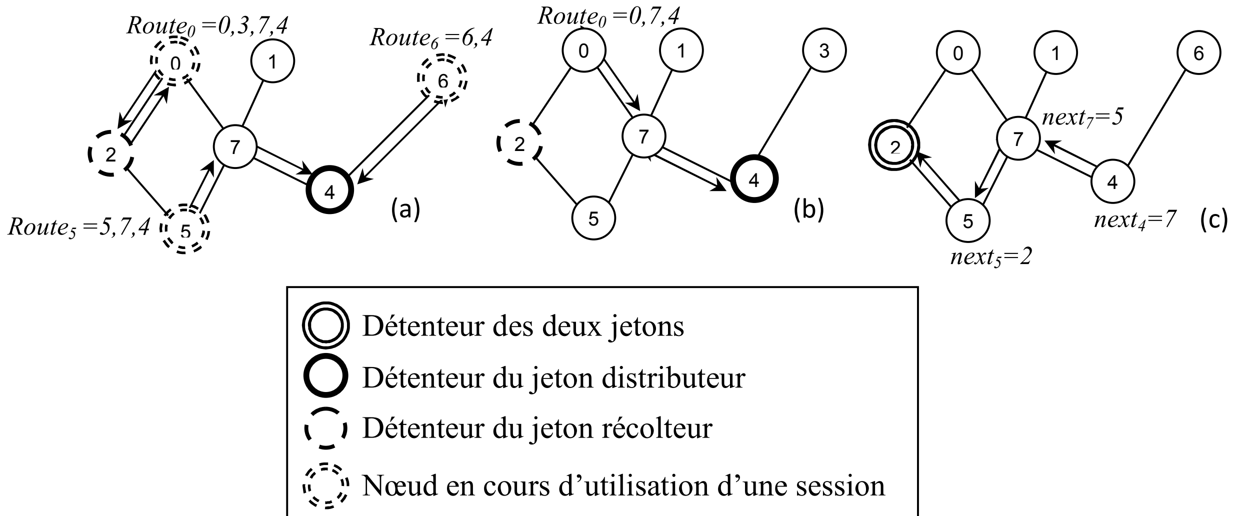


Figure 4.6 : Recherche du détenteur du jeton distributeur.

3.7 Choix de la prochaine session

Dans cette partie, il s'agit de répondre à deux questions fondamentales : la première est : A quel moment il est décidé le choix de la prochaine session à ouvrir ? La seconde est quelle sera la session qui va s'ouvrir ?

Supposons que le nœud i détient le jeton récolteur et le nœud j détient le jeton distributeur. Le fait que i a le jeton récolteur signifie que la prochaine session est déjà décidée. Dès que j décide de fermer la session en cours, il va faire suivre au jeton distributeur la trace du jeton récolteur. Une fois que les deux jetons se rencontrent au niveau du nœud i, celui-ci décide de la session future qui va s'ouvrir en envoyant le jeton récolteur vers le nœud concerné. Quant au nœud i, il s'occupe de la gestion de la session qui a été déjà décidée auparavant, il s'agit de la session dont il est forcément demandeur.

Pour le choix de la session prochaine à ouvrir, il y'a deux volets, la session elle-même mais aussi le nœud concerné par la gestion de cette session.

Dans notre solution, nous proposons de satisfaire les demandes des nœuds selon leur ancienneté mais aussi on prend en considération la distance qui sépare les nœuds demandeurs du nœud détenteur des jetons. De ce fait, pour chaque session demandée nous calculerons le couple moyenne d'âges des nœuds demandeurs d'une session X (E_X) et la distance moyenne en nombre de pas(D_X) qui séparent les demandeurs du nœud actuel détenteur du jeton récolteur pour le choix de la prochaine session. Donc, la session élue sera celle qui a le couple (E_X, D_X) le plus petit. Dans le cas où deux sessions ont les mêmes valeurs, le choix se fera selon le nombre moyen d'ouvertures de chacune des sessions. La session ayant la moyenne la plus inférieure sera élue. Quoique ce choix peut être subjectif dans le sens où si une session a été demandée plusieurs fois

alors qu'une autre ne l'a pas été, la deuxième sera prioritaire. Pour cela, nous rajoutons ce paramètre à la troisième position.
Un deuxième volet, consiste à choisir le prochain nœud initiateur. Une fois la prochaine session est élue, il faudra choisir parmi les nœuds demandeurs de celle-ci le nœud qui va initier cette session (ouverture de celle-ci, invitation des nœuds et fermeture de cette session). Dans notre contexte, le choix se fera par rapport à la distance qui sépare les nœuds demandeurs du détenteur du jeton récolteur : l'idéal est de choisir à partir des routes existantes dans le jeton récolteur le demandeur étant au centre par rapport aux autres demandeurs de cette session. Pour des contraintes algorithmiques, on a choisi le nœud le plus proche du nœud actuel détenteur du jeton récolteur. Une fois la session et le nœud sont élus le jeton récolteur sera acheminé vers ce dernier. D'autres critères de choix peuvent être adoptés, si l'on considère une fonction qui retourne ce choix.

3.8 Perte des jetons

Dans la solution proposée le partitionnement ne peut pas avoir lieu. Par ailleurs, elle prend en charge l'isolement temporaire (ce dernier est limité et ne peut pas causer de duplication de jetons) et la panne (sans causer de partitionnement dans le réseau) des nœuds. La panne d'un nœud détenteur de l'un ou des deux jetons cause implicitement une perte de l'un ou des deux jeton(s). De ce fait, trois cas de perte de jetons peuvent être détectées: perte du jeton récolteur, perte du jeton distributeur ou perte des deux jetons.
La perte de jetons est détectée par un nœud demandeur d'une ressource, qui après un certain temps ne reçoit ni les jetons ni invitation. Dans ce cas, le nœud fait une autre tentative avec le même numéro de séquence. Le fait d'utiliser le même numéro de séquence lui permet de garder sa priorité entre les différents nœuds demandeurs puisque la satisfaction des requêtes prend en considération l'ancienneté de ces dernières. Si toutefois, après un certain nombre de tentatives le nœud ne reçoit ni jetons ni invitation, il soupçonne la perte des deux jetons dans le cas où il n'a reçu aucun des jetons, ou bien la perte du jeton distributeur s'il a reçu le jeton récolteur mais pas le jeton distributeur (isolement momentané ou panne). Pour ce faire, il lance une recherche des jetons dans le réseau. Cette recherche consiste à générer un message de recherche qui sera diffusé dans tout le réseau qui permettra d'une part de vérifier l'existence du jeton récolteur dans le réseau mais aussi l'existence du jeton distributeur. Dans le cas où la recherche indique que les deux jetons sont perdus, le nœud ayant le plus petit identificateur crée une autre paire de jetons (cette méthode est similaire à certains mécanismes d'élection). Les jetons crées auront la même identité, celle du nœud créateur (cette identité sera représentée par le couple identificateur du nœud et un numéro de séquence propre au nœud et à la création. Ce dernier, permettra l'existence d'un unique identificateur pour chaque paire de jetons créés). Puis, il incrémente son numéro de séquence. Cette incrémentation garantie cette unicité. Autrement, si la recherche indique que les jetons ne sont pas perdus (parce que, et le nœud détenteur du jeton récolteur et le nœud détenteur du jeton distributeur ont reçu le message de recherche et auquel ils ont envoyé un message d'existence des jetons et ce pour lui permettre d'arrêter la diffusion du message de recherche), dans ce cas-là, le nœud attendra la réception des jetons ou d'une invitation d'un nœud intéressé par la même session. Dans le cas où la recherche indique que l'un des jetons est perdu mais que l'autre jeton existe, le nœud détenteur de ce dernier régénère le jeton perdu en attribuant aux deux jetons une nouvelle identité, celle-ci sera la même pour les deux jetons, celle du nœud porteur du jeton existant et incrémente son numéro de séquence.

Le retour d'un nœud suite à un isolement momentané est détecté à la création des liens, et donc à chaque création de lien, les deux nœuds le formant s'échangent les identités des jetons qui leurs sont associés. Dans le cas où les identités sont différentes l'un des deux nœuds doit mettre à jour les identités des jetons qui lui sont associées. Le problème posé et comment savoir quel est le nœud qui doit mettre à jour ces identités? Pour parvenir à ce problème nous proposons de garder l'identité des jetons qui précédent les jetons actuels au niveau de chaque nœud. Autrement dit, avant de créer une nouvelle paire de jetons suite à la perte de l'un des deux jetons ou des deux, nous gardons trace de leur identité. De ce fait, lors de la création de lien les deux nœuds comparent les nouvelles identités des jetons si elles sont différentes, et en plus la nouvelle identité des jetons de l'un des nœuds correspond à l'ancienne identité des jetons de l'autre nœud, dans ce cas, c'est l'identité des jetons du deuxième nœud formant le lien qui est gardée. Autrement, c'est l'identité des jetons du premier nœud qui est gardée.

4. Hypothèses sur les nœuds et sur le réseau

- Le réseau est composé d'un nombre fixe de nœuds (soit N), chaque nœud possède un identificateur unique.

- Les nœuds utilisent un support de communication sans fil, les liens sont bidirectionnels (i.e. les nœuds ont la même portée des canaux de communications) et fiables.

- Un nœud ne connaît que ses voisins immédiats. Le protocole ne repose ni sur la couche de routage, ni sur une structure logique établie préalablement.

- Les nœuds du réseau peuvent tomber en panne sans causer de partitionnements dans le réseau

- Les mouvements des nœuds peuvent engendrer des connexions et / ou des déconnexions de liens. Cependant la vitesse de déplacement de chaque nœud est limitée.

- Aucune partition dans le réseau ne peut apparaître.

- La couche de liaison permet aux nœuds de connaître leurs voisins à tout moment.

- Un nœud ne peut demander l'accès à une autre session que si la première a été satisfaite.

- La durée d'une session est limitée.

- L'isolement temporaire d'un nœud ne peut pas causer une duplication des jeton(s).

5. Présentation du fonctionnement du protocole

5.1. Structures de données locales à chaque nœud

Chaque nœud i possède les variables et les structures de données suivantes :

Structures de base

- *NS_i :* c'est l'estampille locale générée par le nœud i, initialement égale à 0. Sert à estampiller les requêtes.
- *$Request_NS_i$* : à chaque émission d'une demande d'entrée en SC, l'estampille de cette dernière est sauvegardée dans cette variable. Sa valeur sera utilisée dans le cas où le nœud i génère une nouvelle tentative de demande consécutive à une autre qui n'a pas été satisfaite. Ceci, afin de lui permettre de garder la même priorité entre les deux et d'éviter la famine. Initialement égale à 0.
- *$State_i$:* l'état du nœud i, à un moment donné, *IDLE* i.e. au repos, *Requesting* i.e. demandeur de SC ou bien *InCS* i.e. dans sa SC, initialement le nœud i est au repos.
- *$Holder_i$:* une variable entière qui peut prendre l'une des quatre valeurs suivantes : 0 dans le cas où le nœud ne détient aucun jeton, 1 indique que le nœud détient le jeton récolteur, 2 le nœud détient le jeton distributeur et 3 pour indiquer que le nœud détient les deux jetons, initialement pour i=0 : *$Holder_i$*=3, autrement *$Holder_i$*=0.
- *$Session_i$* : cette variable désigne la session demandée par le nœud i. Initialement égale à -1.
- *N_i :* l'ensemble des voisins du nœud i. Initialement égale à ses voisins.
- *$Nb_attempts_i$* : le nombre de fois où le nœud i a demandé l'accès à la SC avec le même NS sans succès. Initialement égale à 0.
- *$Ignored_Neighbors_i$:* l'ensemble des voisins ignorés par i, lors de la phase du traitement des cassures de routes, afin d'éviter le problème de bouclage. Initialement, cet ensemble est vide.
- *$Release_Route_i$:* cette variable permet de sauvegarder la route qui mène vers l'initiateur de la session afin de pouvoir envoyer le message de libération de la ressource lors de libération de la session. Initialement égale à vide.
- *Nb_Inv_i :* compte le nombre de messages d'invitations envoyés par i. Initialement égale à 0.
- *$Next_i$:* cette variable indique le nœud auquel le jeton récolteur a été acheminé la dernière fois. Initialement égale à 0.
- *RNS_i :* cette variable est un numéro de séquence qui indique la position du nœud sur le chemin du jeton récolteur (tel que le détenteur du jeton récolteur possède la valeur de RNS maximale). Initialement pour i=0, RNS=1, autrement, cette variable est égale à 0.
- *Q_i :* la file d'attente du nœud i (le nœud i étant initiateur d'une session), au format (j, $Route_j$, NS_j, $Nb_attempts_j$, $Session_j$). Ce format représente la demande d'un nœud j, où $Route_j$ est l'ensemble des nœuds intermédiaires entre i et j. NS_j est le numéro de séquence de la demande de j. $Nb_attempts_j$ représente le nombre de fois que le nœud a demandé une session avec le même numéro de séquence. $Session_j$ représente la session demandée par le nœud j.

Structures liées à la régénération des jetons

- *$Token_ID_i$:* représente l'identité des jetons actuels. Initialement égale à 0.
- *$Token_ID_NS_i$:* représente l'estampille des jetons associés à *$Token_ID_i$*. Initialement égale à 0.
- *$OldToken_ID_i$:* représente l'identité des derniers jetons avant les courants. Initialement égale à 0.

- *OldToken_ID_NS$_i$:* représente l'estampille des derniers jetons avant les courants associés à *Token_ID$_i$*. Initialement égale à 0.
- *CandidateTokenID$_i$:* représente l'identité du nœud candidat à l'élection pour la création d'nouvelle paire de jetons. Initialement égale à 0.
- *CandidateTokenID_NS$_i$:* représente l'estampille des jetons associés à *CandidateTokenID$_i$*. Initialement égale à 0.
- *Token_CreatedNS$_i$:*c'est le numéro de séquence local au nœud ; qui sera proposé lors de la procédure de création de nouveau(x) jeton(s). il sera incrémenté à chaque fois que le nœud i propose de créer une paire de jetons. Initialement égale à 0.
- *SearchingToken$_i$:* cette variable indique si le nœud i cherche le ou les jeton(s). elle peut prendre l'une des quatre valeurs suivantes : 0 si le nœud ne cherche aucun jeton, 1 s'il cherche le jeton récolteur, 2 s'il cherche le jeton distributeur et 3 s'il cherche les deux jetons. Initialement égale à 0.
- *CreatorHolder$_i$:* cette variable indique si le nœud qui propose la création d'une nouvelle paire de jetons est détenteur d'un des jetons.

5.2. Les messages

- Le message jeton récolteur : ce jeton récolte les demandes des différents nœuds demandeurs et possède la forme suivante :

TOKEN RECOLTEUR (Token_id, Token_idNS, TokenR_Route, RNS, Token_Req_Set, Token_Session_Set, Token_Route_Set, Token_NS_Set) où

- *Token_id* : représente l'identificateur du jeton récolteur.
- *Token_idNS* : représente le numéro de séquence du jeton récolteur.
- *TokenR_Route* : une suite de nœuds qui constituent la route que doit suivre dans le meilleur cas le jeton récolteur pour atteindre une destination donnée.
- *RNS* : cette variable sert à garder la trace du jeton récolteur. Initialement égale à 1.
- *Token_Req_Set* : séquence des demandeurs de SC, il représente une vue partielle de tous les demandeurs dans le réseau.
- *Token_Session_Set* : séquence des sessions demandées par les nœuds qui sont dans l'ensemble *Token_Req_Set*.
- *Token_Route_Set* : séquence des routes qui mènent vers les différents nœuds demandeurs qui sont dans l'ensemble *Token_Req_Set*.
- *Token_NS_Set* : séquence des numéros de séquences des nœuds demandeurs et qui sont dans l'ensemble *Token_Req_Set.*

- Le message jeton distributeur : ce jeton donne le privilège au nœud qui le détient s'il détient aussi le jeton récolteur et possède la forme suivante :

TOKEN DISTRIBUTEUR (Token_id, Token_idNS, RNSD,TokenD_Req_Set, TokenD_NS_Set) où

- *Token_id* : représente l'identificateur du jeton distributeur.
- *Token_idNS* : représente le numéro de séquence du jeton distributeur.
- *RNSD* : cette variable sert à retrouver le chemin vers le détenteur du jeton récolteur dans le cas de perte de route. Elle représente RNS du nœud émetteur du jeton distributeur.

- *TokenD_Req_Set:* séquence des demandeurs concernés par la session courante.
- *TokenD_NS_Set* : séquence des numéros de séquences des nœuds auxquels les demandes ont été satisfaites. Sa sert à purger le jeton récolteur.

- Le message de demande d'entrer en SC : il a le format suivant *REQUEST(i,* NS_i, RNS_i, $Route_i$, *Nb_attempts*$_i$*, X)*, où NS_i est l'estampille de la demande, RNS_i est l'estampille du jeton récolteur, $Route_i$ est la route qui mène à i (elle sera construite au fur et à mesure que la demande de i passe par chaque nœud intermédiaire), *Nb_attempts*$_i$ représente le nombre de tentatives pour l'accès en SC qu'a effectué le nœud i avec le même NS_i, *X* est la session demandée par le nœud i.

- Le message d'invitation pour entrer en SC : il a le format suivant *INVITE(i,* $Route_i$, *Invite_Route*$_i$, NS_i, *X),* où *i* représente le nœud qui invite pour entrer en SC (i.e. le nœud initiateur de la session *X*), $Route_i$ est la route qui mène à i (cette route est utilisée afin de pouvoir acheminer les messages *RELEASE* vers le nœud initiateur). *Invite_Route*$_i$ est la route qui mène à j (destinataire de ce message). NS_j représente le numéro de séquence avec lequel le nœud j à fait sa demande. *X* est la session ouverte actuellement.

- Le message qui indique au nœud initiateur qu'un nœud utilisateur de la session *X* l'a quitté, il a le format suivant : *RELEASE (i, Release_Route*$_i$*, X)*, où i représente le nœud qui a fini d'utiliser sa session, *Release_Route*$_i$est la route qui mène au nœud initiateur et *X* est la session utilisée.

- Le message de recherche des jetons :*IsThereToken(i, Request_NS*$_i$*,Route*$_i$*, Nb_attempts*$_i$*, Token_CreatedNS*$_i$*, X,SearchingToken*$_i$*, Holder*$_i$*, aware*$_i$*)*qui sera diffusé lorsque le nœud i soupçonne la non-présence des jetons dans le réseau. Ce message possède les arguments suivants : *Request_NS*$_i$ qui représente le NS de la demande en cours dont le nombre de tentatives a atteint la valeur *Request_Threshold*, $Route_i$ représente la route qui mène à i, *Nb_attempts*$_i$ représente le nombre de tentatives liées à cette demande, *Token_CreatedNS*$_i$ signifie que i se porte volontaire pour la création d'une nouvelle paire de jetons ayant comme estampille le contenu de cette variable, *X* est la session que demande le nœud i, *SearchingToken*$_i$ peut être égale à 3 si les deux jetons sont recherchés, égale à 2 si seulement le jeton distributeur est recherché et égale à 1 si le jeton récolteur est recherché, $Holder_i$ indique si un nœud a déjà reçu ce message et détient un des jetons, $aware_i$ représente l'ensemble des nœuds qui sont en courant de cette diffusion.

- Le message de confirmation de l'existence des jetons : *ThereIsToken(aware*$_i$*)*est la réponse au message précédent diffusé par le nœud ayant le ou les jeton(s) perdus. $aware_i$ représente l'ensemble des nœuds qui sont en courant de cette diffusion.

- Le message de contrôle*LinkInfo(i, Token_ID*$_i$*, Token_ID_NS*$_i$*)*est émis par le nœud i après la création d'un nouveau lien. Ce message est composé des arguments suivants :*Token_ID*$_i$et *Token_ID_NS*$_i$ qui représente respectivement l'identité des jetons (de leur créateur) et son NS.

5.3. Les différents événements

Dans ce qui suit, nous allons présenter le fonctionnement de notre protocole en utilisant la description événementielle et en faisant parfois appel à des méthodes de traitements qui seront développés plus loin dans ce document. Le texte du protocole est donnée en annexe.
Notons que les primitives sont exécutées de manière atomique.

Les primitives liées aux événements du protocole

- **La demande d'entrer à une session**
  Quand un nœud i souhaite accéder à une session *X*, il met $State_i$ à *Requesting*, incrémente son NS_i, sauvegarde ce dernier dans la variable $NS_Request_i$, met $Session_i$à *X* et$Nb_attempts_i$ à1. Ensuite, deux cas se présentent : Le premier est que le nœud i détient les deux jetons, récolteur et distributeur (les jetons sont alors au repos), dans ce cas le nœud déclenche le *LifesessionTimer* (ce timer définie la durée d'ouverture de cette session et ce pour favoriser la vivacité) et accède directement à la session X. le second cas, désigne la non présence du jeton récolteur au niveau du nœud i et donc il achemine sa requête à travers la fonction *VehicleRequest()* dans le but de rechercher le détenteur du jeton récolteur. A la fin, le nœud i déclenche un Timer (appelé *RequestTimer*) qui dans le cas où ni jetons ni invitation sont reçu, permet de redemander à nouveau avec le même NS que la première fois.

- **La réception d'une demande d'entrer à une session** :*REQUEST(j,* NS_j*,* RNS_j*,* $Route_j$*,* $Nb_attempts_j$*, X)*.

Quand un nœud i reçoit un message de demande d'un nœud j *REQUEST(j,* NS_j*,* RNS_j*,* $Route_j$*,* $Nb_attempts_j$*, X*), si i= Last($Route_j$) (i a envoyé un message de demande à l'un de ses voisins et ce dernier a constaté que son *RNS* est inférieur a celui de i) alors il rajoute ce nœud à $Ignored_Neighbors_i$ et choisit un autre voisin (soit le nœud r) pour lui renvoyer ce même message tout en mettant à jour son next. Dans le cas où tous les voisins de i ont un RNS inférieur à celui de i alors il mettra RNS_i à celui du voisin qui a reçu le jeton récolteur le plus récemment par rapport aux autres voisins moins 1.
Sinon (la demande concerne un autre nœud) alors il met à jour son NS_i (i.e. $NS_i := maximum\ (NS_i, NS_j)+1$). Ensuite, si $RNS_j > RNS_i$ (l'émetteur a des informations plus récentes concernant le détenteur du jeton récolteur)alors il retourne ce message à son émetteur. Sinon ($RNS_j<RNS_i$), si le nœud i détient le jeton distributeur, $session_i= X$, LifeSessionTimer> EndLifeSessionTimer (où EndLifeSessionTimer est le temps minimal pour qu'un nœud puisse exécuter la session qu'il demande) alors si $Nb_Inv_i <k$, il incrémente Nb_Inv_i et envoie un message d'invitation au nœud j à travers la fonction *VehicleInvite()*. Autrement ($Nb_Inv_i = k$), il fait appel à la fonction booléenne *AddRequest(j,* NS_j*,*$Nb_attempts_j$*,* $Route_j$*, X)* pour insérer la demande de j dans sa file.

Sinon, si le nœud i détient le jeton récolteur alors il met à jour ce dernier de la manière suivante :
$Token_Req_Set := Token_Req_Set \cup j$
$Token_Session_Set(j) := X$

Token_NS_Set(j) := NS_j
Token_Route_Set(j) := $Route_j$.
De plus, si les jetons étaient au repos alors i passe les jetons récolteur et distributeur au nœud j à travers la procédure *VehicleToken(3).*
Autrement (il ne détient aucun des jetons), il s'ajoute à $Route_j$ et continue l'acheminement de cette requête à travers la fonction *VehicleRequest()*.

- **La réception du message d'invitation** : *INVITE (j,* $Route_j$*,* $Invite_Route_j$*,* NS_j*, X).*

Quand un nœud i reçoit ce message d'un nœud j. S'il est destinataire, il compare $Request_NS_i$ avec NS_j, s'ils sont différents alors il ignore le message. Sinon, il met $Release_Route_i$ à $Route_j$ et accède directement à la session *X*. Sinon, il continue l'acheminement du message *INVITE* à travers la fonction *VehicleInvite().*

- **La réception du message jeton récolteur**: *TOKEN RECOLTEUR (Token_id, Token_idNS, TokenR_Route, RNS, Token_Req_Set, Token_Session_Set, Token_Route_Set, Token_NS_Set).*

Quand un nœud i reçoit ce jeton, il met à jour ses données en l'occurrence son NS_i (i.e. NS_i := *maximum*(NS_i,*maximum*(*Token_NS_Set*))) et met RNS_i à *RNS* puis incrémente ce dernier. Ensuite, s'il n'est pas destinataire, il met $next_i$à Next[*Token_R_Route*] et continue son acheminement en faisant appel à la procédure *VehicleToken(1)*. Autrement (il est destinataire), il met $Holder_i$ à 1 et se met à attendre le jeton distributeur pour permettre aux nœuds demandeurs de participer à cette session tout en recevant de nouvelles.

- **La réception du message jeton distributeur**: *TOKEN DISTRIBUTEUR (Token_id, Token_idNS, RNSD, TokenD_Req_Set, TokenD_NS_Set).*

Quand un nœud i reçoit ce jeton, si le nœud i est destinataire alors il met $Holder_i$ à 3 et déclenche le $LifesessionTimer_i$, met à jour le jeton récolteur (supprime les requêtes satisfaites concernant la session précédente) de la manière suivante :

Pour chaque nœud j de l'ensemble *TokenD_Req_Set* faire :
Si *TokenR_NS_Set(j)< TokenD_NS_Set(j)* alors
TokenR_Req_Set := TokenR_Req_Set - j.
TokenR_NS_Set := TokenR_NS_Set - TokenR_NS_Set(j)
TokenR_Route_Set := TokenR_Route_Set - TokenR_Route_Set(j)
Token_Session_Set := Token_Session_Set - Token_Session_Set(j)
Finsi.
Fait.

Dans un deuxième temps, il enregistre les demandes concernées par la session en cours à son niveau en les insérant dans sa file locale à travers la fonction *AddRequest(j,* NS_j*,* $Route_j$*,* $Nb_attempts_j$*, X)* et supprime sa demande du jeton récolteur comme suit :

TokenR_Req_Set := TokenR_Req_Set - i.
TokenR_NS_Set := TokenR_NS_Set - TokenR_NS_Set(i)
TokenR_LSSN_Set := TokenR_LSSN_Set - TokenR_LSSN_Set(i)
TokenR_Route_Set := TokenR_Route_Set - TokenR_Route_Set(i)
Token_Session_Set := Token_Session_Set - Token_Session_Set(i)

Ensuite, à partir des informations existantes dans le jeton récolteur, il calcule la prochaine session qui sera utilisée et le nœud qui sera initiateur de cette session à travers la fonction

GetNextSession() et met à jour *Token_R_Route.* Puis, il met $next_i$ à Next[Token_R_Route] et achemine le jeton récolteur au nœud initiateur de la prochaine session à travers la procédure *VehicleToken(1).* Puis, il accède à la session X et tant que $Nb_Inv_{i,} < k$, il envoie un message d'invitation à travers la fonction *VehicleInvit()* au nœuds demandeurs existants au niveau locale selon leurs anciennetés et incrémente $Nb_Inv_{i,}$. Sinon (il n'est pas destinataire), il met $next_i$ à Next[*Token_Route*] et continue son acheminement en faisant appel à la procédure *VehicleToken(2).*

- ***La sortie de la session***

Quand un nœud i a fini d'utiliser sa session, il met $State_i$ à *IDLE*, $Session_i$ à -1 et $Nb_attempts_i$ à 0. Si le nœud i n'est pas l'initiateur alors il achemine le message *RELEASE* vers le nœud initiateur à travers la fonction *VehicleRelease()*. Sinon (le nœud est initiateur), il décrémente $Nb_Inv_{i,}$ puis s'il existe des demandeurs de la session en cours dans sa file et *LifeSessionTimer>EndLifeSession* alors il incrémente Nb_Inv_i et envoie un message d'invitation au nœud demandeur à travers la fonction *VehicleInvit().* Autrement, si Nb_Inv_i =0 (tous les nœuds invités ont fini d'utiliser la session) alors il achemine le jeton distributeur à travers la fonction *VehicleToken(2)* si toutefois, le jeton récolteur a été acheminé auparavant.

- **La réception du message de sortie d'une session** : *RELEASE (j,* $Release_Route_j$*, X)*

Quand un nœud i reçoit ce message de j,s'il est destinataire alors il incrémente NS_j existant dans sa file, il décrémente Nb_Inv_i puis s'il y'a un nœud demandeur de la session X dans sa file et *LifeSessionTimer>EndLifeSession* alors il incrémente Nb_Inv_i et il l'invite à travers la fonction *VehicleInvit()*. Autrement, si Nb_Inv_i =0 il achemine le jeton distributeur à travers la fonction *VehicleToken(2)* etmet $Session_i$à -1. S'il n'est pas destinataire il continue l'acheminement de ce message à travers la fonction*VehicleRelease()*.

- **Fin du Timer LifeSessionTimer**

A la fin du Timer LifeSessionTimer*,* le nœud initiateur met à jour le jeton distributeur en enregistrant les demandes satisfaites et ce pour permettre de purger le jeton récolteur, met $Session_i$à -1 et fait suivre le jeton distributeur la trace du jeton récolteur à travers la procédure *VehicleToken(2).*

- **Fin du Timer RequestTimer** associé à une demande d'un nœud i

A la fin du timer *RequestTimer* associé à un nœud i, dans le cas où $Nb_attempts_i$ n'a pas encore atteint un certain seuil défini par *Request_Threshold*, il appel encore la procédure *Ask_For_CS(i, X)* où X représente la session demandée par le nœud i. Sinon le nœud i change la manière de demander la SC. En effet, sa demande va être diffusée sur tout le réseau, de plus cette dernière n'est plus véhiculée par le message *REQUEST*, mais par un message *IsThereToken(i,* $Request_NS_i$*,* $Route_i$*,* $Nb_attempts_i$*,* $Token_CreatedNS_i$*, X,* $SearchingToken_i$*,* $Holder_i$*,* $N_i \cup i$*)* qui de part sa sémantique véhicule :

- La demande d'entrer en SC du nœud i.
- Une recherche des jetons récolteur et distributeur.
- Une demande de création d'un nouveau couple de jetons (dans le cas de perte).

En diffusant ce message, le nœud i attend la réception d'un message de retour *ThereIsToken* pendant une durée de temps limitée (temps de diffusion sur tout le réseau multiplié par 2 (l'aller

et le retour), plus une certaine marge). Ceci est assuré par le timer de régénération des jetons *SuspicionTimer*, à la fin duquel de nouveaux jetons seront créés.

- **La réception du message*IsThereToken* d'un nœud j**

Quand un nœud i reçoit un message *IsThereToken(j,* $Request_NS_j$*,* $Route_j$*,* $Nb_attempts_j$*,* $Token_ID_NS_j$*, X, SearchingToken,* $Holder_j$*,* $aware_j$*)*, deux cas se présentent : soit que ce message a été envoyé suite à un soupçon de perte des deux jetons(parce qu'un nœud demandeur n'a reçu ni jetons ni invitation, *SearchingToken*=3), soit soupçon de perte de l'un des jetons(*SearchingToken*=2 si soupçon de perte du jeton distributeur, *SearchingToken*=1 si soupçon de perte du jeton récolteur).
Si c'est le premier message reçu de ce type et le nœud ne détient pas le jeton ou les jetons recherchés (*SearchingToken* != $Holder_i$), il le traite comme une requête normale, puis, il sauvegarde l'identité du demandeur et la variable *Token_ID_NSj* respectivement dans les variables *CandidateTokenIDi* et *CandidateTokenID_NSi*, déclenche son propre timer *SuspicionTimer* et se met dans un état d'arrêt momentané du point de vue génération de demandes. De plus, si $Holder_j$est différent de 0 alors il met $CreatorHolder_i$ à vrai. Si le nœud i est porteur de l'un des jetons ($SearchingToken_i$!= $Holder_i$*)*, il serait responsable alors de la régénération de l'autre jeton dans le cas de perte de ce dernier. Pour ce faire, il met à jour les variables *CandidateTokenIDi* et *CandidateTokenID_NSi, met SearchingToken à 2/*$Holder_i$*(*si $Holder_i$ = 1 alors le nœud i enregistre la demande de j dans le jeton récolteur) *et propage le message IsThereToken(j,* $Request_NS_j$*,* $Nb_attempts_j$*,* $Token_CreatedNS_i$*,* $Route_j$*, X,* $SearchingToken_i$*,* $Holder_i$*,* $N_i \cup i$*).*
Autrement (ce n'est pas le premier message reçu de ce type et i n'est porteur d'aucun jeton ($Holder_i$=0)), si le nœud émetteur est détenteur de l'un des jetons alors il met à jour ses variables *CandidateTokenIDi* et *CandidateTokenID_NSi*, met $CreatorHolder_i$ à vrai et continu la propagation de ce message. Sinon, si $Holder_j = Holder_i = 0$, $CreatorHolder_i$ = faut *et* $j <$ *CandidateTokenIDi*, i met à jour les variables *CandidateTokenIDi* et *CandidateTokenID_NSi* par les nouvelles valeurs reçues et continue la diffusion de ce message.
Dans le cas où $Holder_i$ = *Token_Searching* (i est porteur du jeton ou des jetons recherchés), il ne traitera que le premier message reçu. Ce traitement se résume à une attente dans laquelle i s'assure que les différentes diffusions simultanées de ce type de message se soient arrêtées (dans cette période d'attente i ne fait pas passer le jeton ou les jetons au prochain), puis il diffuse le message *ThereIsToken*.

- **Fin du timer SuspicionTimer**

A la fin du timer *SuspicionTimer* associé à un nœud i, ce dernier se débloque, met à jour les variables $Token_ID_i$ et $Token_ID_NS_i$ par $CandidateTokenID_i$ et $CandidateTokenID_NS_i$ respectivement. Si i est l'élu alors si $Holder_i$ = 0 alors il crée une nouvelle paire de jetons et initie sa session. Sinon, si $Holder_i$ = 2 alors il crée un jeton récolteur. Autrement, si $Holder_i$ = 1 le nœud crée un nouveau jeton distributeur et initie sa session.

- **Détection d'un nouveau lien (Link_up)**

Dès qu'un nœud i détecte un nouveau voisin, il lui transmet un message *LinkInfo* qui contient des informations sur l'état de i et selon cet état, les données contenues dans ce message peuvent

contenir des informations normalement transportées par le message *IsThereToken* (ce qui nous permet d'optimiser le nombre de messages) et ceci de la manière suivante :
Si i n'est pas bloqué (i.e. ne recherche pas le jeton) alors le message *LinkInfo* contiendra en plus de l'identité de i, les informations sur l'identité du jeton utilisé par i (i.e. $Token_ID_i$, $Token_ID_NS_i$). Sinon (dans le cas où i est bloqué pour rechercher le jeton), le message *LinkInfo* contiendra en plus des informations transmises dans le cas précédent, les informations contenues dans le message *IsThereToken* normalement passant par i (et ceci pour prévenir les cas d'isolement momentanées).
On peut constater que pour un lien crée, deux messages sont échangés (par cause de symétrie de la détection de liens) et donc dans un souci d'optimiser le nombre de messages *LinkInfo,* on impose la règle suivante : si i est bloqué alors il transmet systématiquement le message *LinkInfo,* mais si au contraire il est dans l'état normal, un seul des deux nœuds formant le lien prend à sa charge la transmission de ce message. Ce nœud est déterminé en appelant la fonction *GetChoosen(i, j)* qui pour une paire d'identificateurs donnée, retourne l'un des deux. Une politique possible consisterait à choisir l'identificateur le plus petit.

- **La réception d'un messageLinkInfo :** $LinkInfo(j, Request_NS_j, Route_j, Nb_attempts_j, Token_ID_NS_j, Token_Searching, Holder_j, Session_j, aware_j)$

Quand un nœud i reçoit un message *LinkInfo* de la part de j, il compare $Token_ID_j$ et $Token_ID_NS_j$ avec ses propres variables et si elle sont égales, i conclut qu'il ne s'agit pas d'un isolement momentané et si en plus $SearchingToken_j!=0$ alors i exécute un traitement équivalent à celui lors de la réception du message $IsThereToken(j, Request_NS_j, Route_j, Nb_attempts_j, Token_ID_NS_j, X, SearchingToken, Holder_j, aware_j)$. Par contre, si $TokenID_j \neq TokenID_i$ ou ($TokenID_j = TokenID_i$ et $Token_ID_NS_j \neq Token_ID_NS_i$) i.e. qu'un des deux nœud était en isolement temporaire lorsqu'une nouvelle paire de jetons a été créé. En conséquence, un des deux nœuds doit mettre à jour les identificateurs de ces jetons (celui qui était en isolement temporaire). Pour ce faire, si $Token_ID_j$et $Token_ID_NS_j$ du nœud j correspondent à $OLD_Token_ID_i$et $OLD_Token_ID_NS_i$ du nœud i alors, i lui envoie un message $LinkInfo(i, TokenID_i, Token_ID_NS_i)$. Dans le cas contraire, il met $TokenID_i$ à $TokenID_j$et $Token_ID_NS_i$ *à* $Token_ID_NS_j$.

<u>*Les routines utilisées :*</u>

- ***AddRequest(j, NS_j, $Route_j$, $Nb_attempts_j$, X)***

Quand la fonction booléenne *AddRequest* est appelée par un nœud i, deux cas se présentent : soit la requête associée à j n'existe pas dans la file Q_i, alors, dans ce cas, la requête de j sera insérée normalement. Soit une requête associée à j existe déjà dans cette file, alors, dans ce cas, i ne met à jour les attributs(en l'occurrence la route si la nouvelle route reçu est meilleur) de la requête de j que si $Nb_attempts_j > Q_i(j).Nb_attempts$.
Si l'insertion de j est réussie alors la fonction *AddRequest* retourne vrai, sinon elle retourne faux.

- ***VehicleToken(l)***

Trois cas se présentent lorsque cette procédure est appelée :

- Le premier cas est que la procédure est appelée par un nœud i avec l'argument l=1, cela veut dire que le jeton récolteur va quitter ce nœud pour un autre sauf dans le cas où i n'aurait aucun voisin et dans ce cas la procédure échoue et le nœud i garde le jeton récolteur jusqu'à ce que de nouvelles requêtes l'atteignent.
  Le jeton récolteur doit être acheminé vers le prochain destinataire. Pour ce faire, le nœud i vérifie si le destinataire n'est pas un de ses voisins, si c'est le cas il lui donne le jeton récolteur immédiatement et met son next à destinataire. Sinon, i doit se contenter de la route présente dans le jeton, si le premier de cette route fait partie du voisinage immédiat, i lui passe directement le jeton tout en mettant à jour son next et $Holder_i$ (si $Holder_i$ =3 alors il l'a met à 2 autrement il l'a met à 0). Dans le cas contraire (i.e. la route a été brisée), le nœud i tente de recalculer la route vers le destinataire en appelant la fonction *Retreive_Route()* avec *TokenR_Route* comme argument, dans le cas où cette dernière réussit, le jeton poursuit son chemin sur la route recalculée, sinon, le jeton récolteur devient sous la responsabilité de i et c'est à ce dernier que revient le choix du prochain destinataire. A ce niveau, i cherche à mettre à jour les routes des nœuds demandeurs de la même session que le destinataire présents dans le jeton récolteur et ceci en vérifiant leur présence dans l'ensemble de ses voisins immédiats (i.e. l'ensemble N_i), et ainsi les demande dans le jeton seront triées une nouvelle fois dans le cas où des modifications auraient eu lieu. Puis, il choisit parmi les demandeurs de la même session le plus proche de i et tente de lui acheminer le jeton, s'il constate que la route associée à cet élément est brisée alors il tente de la recalculer avec la fonction *Retreive_Route()*, si cette dernière réussit, il met à jour $Holder_i$ (si $Holder_i$ = 3 alors il l'a met à 2 autrement il l'a met à 0) et le jeton sera acheminé vers cet élément. Sinon, i passe au second élément et ainsi de suite. Si arrivé au dernier demandeur de cette session existant dans le jeton récolteur, le nœud délègue un autre nœud demandeur d'une autre session pour lui envoyer le jeton récolteur et ainsi de suite.

- Le deuxième cas est que la procédure est appelée par un nœud i avec l'argument l=2,cela veut dire que le jeton distributeur va quitter ce nœud pour atteindre le détenteur du jeton récolteur sauf dans le cas où i n'aurait aucun voisin et dans ce cas la procédure échoue. Dans le cas contraire, le jeton distributeur doit être acheminé vers le nœud initiateur. Pour ce faire, le nœud i vérifie si $Next_i$ fait partie du voisinage immédiat alors il met $Holder_i$ à 0 et lui passe directement le jeton. Sinon (i.e. la route a été brisée), le nœud i tente de retrouver la trace du jeton récolteur. Pour ce faire, il choisit aléatoirement un de ses voisins pour lui envoyer le jeton distributeur et met $Holder_i$ à 0. A la réception de ce jeton, le nœud destinataire compare son RNS avec celui de l'émetteur existant dans le jeton distributeur, s'il est inférieur alors il renvoie le message à son émetteur, dans le cas contraire, à son tour il choisit un de ses voisins pour lui envoyer le message et ainsi de suite jusqu'à arriver au nœud détenteur du jeton récolteur. Dans le cas où tous les voisins ont un RNS inférieur à celui de l'émetteur alors il choisira le voisin ayant le RNS le plus grand par rapport aux autres (soit le nœud r), met son RNS à RNS_r - 1 et achemine le jeton au nœud r.

- Le troisième cas est que la procédure est appelée par un nœud i avec l'argument l=3, cela veut dire que les deux jetons récolteur et distributeur vont quitter ce nœud pour un autre sauf dans le cas où i n'aurait aucun voisin et dans ce cas la procédure échoue. Dans le cas

contraire, les jetons doivent être acheminés vers le prochain destinataire. Pour ce faire, le nœud i vérifie si le destinataire n'est pas un de ses voisins, si c'est le cas, il lui donne les jetons récolteur et distributeur immédiatement et met $Holder_i$ à 0. Sinon, i doit se contenter de la route présente dans le jeton récolteur, si le premier de cette route fait partie du voisinage immédiat, i lui passe directement le jeton tout en mettant à jour $Holder_i$ à 0. Dans le cas contraire (i.e. la route a été brisée), le nœud i tente de recalculer la route vers le destinataire en appelant la fonction *Retreive_Route()* avec *TokenR_Route* comme argument, dans le cas où cette dernière réussit, les jetons poursuivent leur chemin sur la route recalculée, sinon, les jetons deviennent sous la responsabilité de i et c'est à ce dernier que revient le choix du prochain destinataire. Ce choix est similaire au cas de perte de route dans l'acheminement du jeton récolteur (l=1).

- **GetNextSession**

Quand cette procédure est appelée par un nœud i, cela veut dire qu'un nœud veut acheminer le jeton récolteur afin de préparer la prochaine session à ouvrir. Pour ce faire, le nœud détenteur de ce jeton va calculer pour chacune des sessions la moyenne des estampilles des requêtes (pour chaque session, il fait l'addition des NS_i des nœuds demandeurs et divise cette somme par le nombre de demandeurs) et le résultat va être gardé dans un tableau. Une fois toutes les moyennes sont calculées, le nœud va choisir la session qui correspond à la plus petite moyenne d'âge dans le tableau. Cette session sera la prochaine session qui va s'ouvrir. Dans le cas où deux sessions ont les moyenne d'estampilles égales, le nœud va calculer pour chacune des sessions la distance moyenne qui sépare les demandeurs de ce nœud (il additionne le nombre de pas qui sépare les demandeurs de lui et divise celle-ci par le nombre de demandeurs). Une fois les distances moyennes sont calculées, le nœud va sélectionner la session qui correspond à la distance moyenne la plus petite. A ce stade, si deux sessions ont les distances moyennes égales, il calculera la moyenne d'ouvertures de chacune des sessions et c'est celle qui a la moyenne la plus faible qui sera élue.
Une fois la session est élue, le nœud choisira parmi les demandeurs de cette session le nœud ayant la distance la plus proche de la distance pour lui acheminer le jeton récolteur. Le nœud élu, sera l'initiateur de la prochaine session.

- ***VehicleInvit()***

Quand cette procédure est appelée par un nœud i, cela veut dire que le message ***INVITE*** va être transmis d'un nœud à un autre à travers une route prédéfinie. Pour ce faire, le nœud i vérifie si le destinataire n'est pas un de ses voisins, si c'est le cas il lui transmet le message ***INVITE*** immédiatement, sinon, i doit se contenter de la route présente dans ce dernier, si le premier de cette route fait partie du voisinage immédiat, i lui passe directement le message. Sinon (i.e. la route a été brisée), il tente de la recalculer en appelant la fonction *Retreive_Route()* avec $Invite_Route_i$ comme argument, dans le cas où cette dernière réussit, le message poursuit son chemin sur la route recalculée, sinon, il y'a eu échec.

- ***VehicleRequest()***

Quand cette fonction est appelée, celle-ci tente de rechercher le jeton récolteur, Cette recherche peut se faire de deux manières : soit le $Next_i$ est toujours voisin du nœud i et dans ce cas, il envoie sa requête à $Next_i$sous la forme d'un message *REQUEST (i,* NS_i, RNS_i, $Route_i$, $Nb_attempts_i$*,X)*. Soit le $Next_i$ n'est plus son voisin et donc le nœud va tenter de retrouver la trace du jeton récolteur. Pour ce faire, il choisit l'un de ses voisins d'une manière aléatoire (soit le nœud j), met $Next_i$ à j et envoie sa demande à j sous la forme d'un message *REQUEST (i,* NS_i, RNS_i, $Route_i$, $Nb_attempts_i$*,X)*.

- ***VehicleRelease()***

Quand cette procédure est appelée par un nœud i, cela veut dire que le message ***RELEASE*** va être transmis d'un nœud à un autre à travers une route prédéfinie. Pour ce faire, le nœud i vérifie si le destinataire n'est pas un de ses voisins, si c'est le cas il lui transmet le message ***RELEASE*** immédiatement. Sinon, i doit se contenter de la route présente dans ce dernier, si le premier de cette route fait partie du voisinage immédiat, i lui passe directement le message. Sinon (i.e. la route a été brisée), il tente de la recalculer en appelant la fonction *Retreive_Route()* avec $Release_Route_i$ comme argument, dans le cas où cette dernière réussit, le message poursuit son chemin sur la route recalculée, sinon, le message est ignoré.

- **Retreive_Route(Route)**

Quand cette fonction est appelée par un nœud i, cette dernière tente de récupérer la route entrée en argument comme suit : à partir du dernier élément composant cette route, la fonction vérifie si ce dernier est un voisin non ignoré de i. Si c'est le cas, elle retourne la nouvelle route à partir de cet élément. Si arrivée au premier nœud, elle retourne une route vide, synonyme de son échec.

6. Discussion et conclusion

Dans ce chapitre, nous avons proposé un algorithme basé sur l'approche à jeton pour résoudre le problème de groupe k exclusion mutuelle dans un réseau mobile ad hoc. Cet algorithme est adapté à la mobilité des nœuds, n'utilise pas la couche de routage et utilise une topologie plate, distribué, équitable et symétrique.

- L'algorithme utilise deux sortes de jetons. Initialement, nous avons pensé à utiliser un seul jeton mais ce dernier devait récolter les requêtes en plus de la gestion de l'ouverture et de la fermeture d'une session et donc le volume d'informations transportées était important. Dans un souci d'équilibre de charge, on a proposé d'utiliser deux jetons : jeton récolteur et jeton distributeur et chacun d'eux aura une partie de l'information à transporter. La différence entre les deux est que le premier récolte les requêtes et le deuxième gère l'ouverture, les invitations des nœuds et la fermeture d'une session.

- L'algorithme proposé est orienté recherche du jeton et non circulation du jeton. Quand un nœud demande une session, il se mettra à chercher le jeton récolteur. Pour se faire, il choisit parmi ses voisins le dernier qui a reçu le jeton récolteur pour lui transmettre sa requête. Etant donné l'environnement de réseaux mobiles ad hoc, il est possible que le nœud demandeur soit le dernier à recevoir ce jeton. Dans ce cas, ce nœud demandeur à des informations plus récente que ses voisins actuels, alors il choisira le voisin ayant reçu le jeton le plus récemment par rapport aux autres voisins et à ce dernier de chercher le jeton récolteur et ainsi de suite.

- Comme nous l'avons expliqué dans le chapitre précédent, un protocole de Groupe k-exclusion mutuelle doit garantir les deux propriétés (sûreté et vivacité) et doit être efficace.

  - Le premier volet de la première propriété (au plus k nœuds peuvent exécuter simultanément une session donnée) est garanti vu qu'il existe un seul initiateur (une seule paire de jetons) dans le réseau et c'est à lui d'inviter les nœuds à participer à sa session. Comme ces derniers sont comptabilisés par un compteur qui est limité à k, ceci implique qu'au plus k nœuds peuvent exécuter leurs sessions simultanément. Cependant, une sous-utilisation de ressources peut avoir lieu. En effet, dans le cas par exemple où le message *Invite()* est ignoré parce que la route a été brisée et que cette dernière n'a pas pu être récupérée, le nœud destinataire de ce message ne va pas le recevoir. Or, le nœud initiateur le comptabilise avec les nœuds en cours d'utilisation de la session en cours. De ce fait, même si le nœud initiateur invite k nœuds, il n y' a pas forcément k nœuds dans la section critique. Ce paramètre fera l'objet de mesure dans le chapitre suivant.
    Nous pouvons vérifier que notre protocole satisfait le deuxième volet de la première propriété (si deux nœuds i et j sont entrain d'exécuter leurs sessions, alors $Session_i$=$Session_j$.). Si on suppose que deux nœuds demandeurs (soit l et m) ont été invités par deux nœuds (soit i et j), alors nous déduisons que, et le nœud i et le nœud j sont détenteurs chacun d'un jeton récolteur et d'un jeton distributeur, puisque pour qu'un nœud puisse être initiateur il faut qu'il ait les deux jetons. Comme il existe au plus une paire de jetons dans le réseau, implique qu'il ne peut y avoir qu'un seul nœud initiateur et donc i=j. D'un autre côté, un nœud ne peut demander plus d'une session à un moment donné, ce qui en résulte que $session_l$=$session_m$.

  - Afin de favoriser la satisfaction de l'attente bornée des nœuds (vivacité), un certain nombre de mécanismes ont été utilisés. Parmi ces mécanismes nous citons l'utilisation d'un timer (LifeSessionTimer) qui limite la durée de vie d'une session. Aussi, pour le choix de la prochaine session à ouvrir, nous avons tenu en compte l'âge de la requête. En effet, le nœud détenteur des deux jetons (initiateur de la session en cours) va calculer la moyenne d'âges par session et c'est la session ayant la moyenne d'âge la plus petite qui sera élue. D'un autre côté, lorsqu'une session est élue et le nœud initiateur est choisi, ce dernier va devoir inviter d'autres nœuds à utiliser leurs sessions simultanément. Pour se faire, le nœud initiateur ordonne sa file (cette dernière, contient les requêtes des nœuds

intéressés par la même session que lui) selon l'âge des requêtes, de la plus ancienne à la moins ancienne et les nœuds seront invités dans ce sens.

- Un algorithme peut satisfaire la vivacité et la sureté mais peut ne pas être efficace. De ce fait, nous avons introduit des mécanismes qui favorisent l'efficacité. Quand un nœud est qualifié d'initiateur, il invite les demandeurs de la même session que lui à accéder à celle-ci simultanément. Comme nous l'avons mentionné précédemment, une sous utilisation de la ressource peut avoir lieu dans le cas de perte de routes. Néanmoins, des tentatives de récupérations de routes sont faites avant d'abandonner un message *Invite()* ou *Release().* Par ailleurs, quand le nœud initiateur reçoit un message de libération d'une session *Release()* d'un nœud qui vient de quitter la session courante, il invite un autre nœud demandeur de la même session à accéder simultanément, si le timer LifeSessionTimer est supérieur à un certain seuil. Dans ces mêmes conditions, si le nœud initiateur reçoit une demande concernant la session courante et que le nombre d'utilisateurs de cette session est inférieur à k, alors il l'invite. D'un autre côté, pour le choix de la prochaine session élue, nous avons aussi tenue en compte le nombre de demandeurs de la session et ce pour permettre plus de concurrence.

- La solution proposée tolère la panne de nœuds. Si après un certain temps, un nœud demandeur ne reçoit ni les jetons ni invitation, il fait une autre tentative (redemande). Si toutefois, après un certain nombre de tentatives le nœud ne reçoit ni jetons ni invitation, il soupçonne la perte des deux jetons dans le cas où il ne détient aucun des jetons. Pour ce faire, il lance une recherche des jetons dans le réseau. Cette recherche consiste à diffuser un message spécial dans tout le réseau qui permettra d'une part de vérifier l'existence du jeton récolteur dans le réseau mais aussi l'existence du jeton distributeur. Dans le cas où la recherche indique que l'un des jetons existe, le porteur de ce dernier sera responsable de créer le jeton perdu. Autrement(les deux jetons sont perdus), un des nœuds sera élu pour créer une paire de jetons.

- Quant à l'isolement temporaire, quand un nœud revient de cette situation, il compare l'identificateur de ses jetons avec ceux du nœud avec lequel le lien a été formé. Si les identificateurs sont différents, ce nœud doit mettre à jour ses identificateurs. Pour se faire, avant de créer une nouvelle paire de jetons (suite à une perte de jetons), les nœuds gardent trace des anciens identificateurs et à chaque fois qu'un lien est établi les deux nœuds le formant comparent leurs identificateurs. Dans le cas où ils sont différents, le nœud auquel les identificateurs correspondent aux anciens identificateurs de l'autre nœud met à jour ses identificateurs. De cette manière, la duplication d'informations est évitée.

Chapitre 5 :

Simulation du protocole

1. Introduction

Les réseaux de télécommunications et, plus particulièrement les réseaux informatiques ont pris, durant cette dernière décennie, un essor sans précédent. Devant l'évolution des techniques de pointes et de la technologie, de nombreuses solutions sont envisageables pour un même problème, ce qui offre un confort non négligeable aux chercheurs. La preuve de correction d'un algorithme et l'évaluation de ses performances devrait se faire avec des outils mathématiques, les réseaux de pétri, les chaines de markov, etc… La simulation ne donne pas une preuve de correction formelle mais, elle permet non seulement de tester un algorithme sans aucun cout de nouvelles technologies et de nouveaux protocoles, mais aussi d'anticiper les problèmes qui pourraient se poser dans le futur. Elle renseigne sur le comportement de l'algorithme devant des scénarios prédéfinis qui mettent en jeu un ensemble de paramètres tel que la mobilité dans le sens de la variation du déplacement et de la vitesse des nœuds, la charge des demandes, la connectivité, etc… La variation de ces paramètres permet suite aux différentes exécutions du programme une prise des mesures de performances. Ces mesures peuvent être déployées dans une analyse qui permet de tirer des conclusions sur le fonctionnement et l'efficacité de l'algorithme ainsi que les schémas favorables et défavorables quant aux performances calculées.

NS (Network Simulator) est un logiciel de simulation de réseaux informatiques. C'est un simulateur à événements discrets et supporte plusieurs protocoles à différentes couches (session, application, transport, liaison et mac). Il fournit un environnement complet à plusieurs plates-formes telles que les réseaux filaires,

Dans ce présent chapitre, nous allons présenter les résultats de l'évaluation des performances de notre protocole de groupe k exclusion mutuelle pour les réseaux mobiles ad hoc effectuée par simulation à l'aide d'un outil adapté aux environnements mobiles tels que les réseaux mobiles ad hoc, NS2 (***Network Simulator 2***). Ce dernier nous offre la possibilité d'effectuer des mesures, de suivre le comportement des nœuds et de visualiser les changements de topologies du réseau.

Ce chapitre sera organisé de la manière suivante : dans un premier temps, nous allons donner une brève présentation historique du simulateur NS2, ainsi que son mode de fonctionnement. Ensuite, nous allons décrire le processus d'implémentation du protocole ainsi que

l'environnement dans lequel les simulations ont été effectuées. Dans une prochaine étape, les résultats des simulations seront listés sous forme de graphiques qui feront l'objet d'interprétations suivies d'une discussion. Nous terminerons, naturellement, ce chapitre par une conclusion.

2. Le simulateur NS2

2.1 Historique et présentation

En 1989, NS était une variante « The Real Network Simulator » [Site 06] puis a évolué, principalement au cours de ces dernières années, grâce à des efforts administrés par l'université de BERKLEY. En 1995, le développement de NS a été soutenu par DARPA (Defence Advanced Research Projects Agency), et cela, à travers le projet VINT pour (Virtual InterNet Testbed) [Site07] dans les laboratoires de LBL, Xerox PARC, UCB et USC/ISI.

Le simulateur de réseau NS (Network Simulator) est un environnement de simulation à base d'évènements discrets dans le temps et atomiques. A partir d'une configuration (nœuds et liens), on peut simuler des applications réseaux : flux constant (CBR), ftp, telnet et suivre les paquets IP le long de leur trajet. Pour effectuer les simulations, NS2 utilise une version orientée objet du langage tcl appelée Otcl, qui sera l'interface de commande et de configuration à travers laquelle l'utilisateur définit la topologie du réseau, les caractéristiques des liens physiques, les agents ou protocoles intervenant et les applications de communications qui y seront développées.

Malgré une large utilisation de NS, la documentation y afférente reste toujours insuffisante et fait vraiment défaut (en ce qui nous concerne, la meilleure documentation dont nous avons disposée est le code source de NS2 écrit en C++ et tcl). NS est un logiciel très évolutif. En effet, de nombreux composants y sont intégrés chaque année. Ainsi, il existe de nombreuses versions pour les différents protocoles implémentés. De plus, il y'a un grand nombre de classes prédéfinies qui permettent de mettre en œuvre plusieurs implémentations intégrant des protocoles de transmission TCP, UDP/IP, des files d'attente diverses (files de paquets), un routage fixe et dynamique (en particulier : DSR, AODV, DSDV et TORA pour les réseaux Ad Hoc)…etc.

En ce qui nous concerne, nous avons réalisé notre travail à l'aide de la version 2.34 de NS2 sous le système d'exploitation Ubuntu 9.10

2.2 Fonctionnement

Le développement de protocoles dans NS suit une approche orientée objet où deux langages de programmation sont utilisés: C++ et tcl (Otcl).

Les modules de base du simulateur et principaux protocoles (qui forment le noyau) sont implémentés en C++. Une couche Otcl a été rajoutée au noyau pour servir d'interface à l'utilisateur. A partir de cette dernière, l'utilisateur peut faire appel aux différentes classes déjà prédéfinies, les manipuler et les augmenter par de nouveaux attributs et/ou de nouvelles méthodes dans le cas où cela s'avère nécessaire.

La réalisation de simulations à l'aide du langage Otcl est donc très intéressante du point de vue flexibilité et facilité de manipulation. En contrepartie, le fait que ce soit un langage interprété rend la détection d'erreurs et le débogage pas très faciles.

3. Implémentation du protocole de groupe k exclusion mutuelle

3.1 Méthode d'implémentation

Pour la réalisation de notre protocole, nous avons opté pour l'approche objet offerte par Otcl. Nous pouvons noter que l'utilisation d'une approche objet n'est pas obligatoire. En effet, une approche procédurale peut parfaitement y être développée.

Chaque événement du protocole est associé à une méthode, élément d'une classe, dans le but est d'accomplir une certaine tâche.

Prenons l'exemple de la classe « Agent/UDP » dont les instances sont des objets agents de transport UDP (chaque nœud peut posséder un ou plusieurs agents de transport). La méthode « process_data » de ces derniers est associée à l'événement réception de messages.

La classe « Node », dont les instances sont des nœuds mobiles, se définit comme étant la classe la plus importante, car elle regroupe les principales méthodes. Nous pouvons citer :

- ✓ AskForSC: demande d'entrer en SC.
- ✓ EnterCS: entrer en SC.
- ✓ LeaveCS: sortir de la SC.
- ✓ VehicleToken: qui est appelée soit pour passer l'un des jetons ou pour passer les deux au même temps au prochain.
- ✓ VehicleInvite: passer le message ***INVITE*** au prochain.
- ✓ VehicleRelease: passer le message ***RELEASE*** au prochain.

A chaque objet de type « Node », nous avons associé un objet de type « Node_info » qui représente la structure de données propre au nœud (par exemple : la listes des voisins, le status,…) et les méthodes permettant d'attribuer, de modifier ou de consulter ces variable. Par ailleurs, trois objets TIMER sont attribués à chaque nœud. Ces derniers sont respectivement de type « ReAskForSCTimer », « RegenerateTokenTimer » et « LifeSessioTimer ».

3.2. Environnement de la simulation

L'environnement dans lequel les différentes simulations ont été effectuées, peut être décrit à travers un certain nombre de paramètres :

- o *Les paramètres constants :*

  - La surface de mouvement: 500× 1000 mètres.
  - Le nombre de nœuds dans le réseau: 70.
  - Le nombre de sessions: 5.

- La durée de la SC: 0.1 seconde.
- Le rayon de communication entre nœuds mobiles: 250 mètres.
- Le temps de simulation: 300 secondes.
- Le nombre k=10

- *Les paramètres variables :*

- La mobilité: La définition de mobilité que nous avons adoptée est basée sur le mouvement relatif des nœuds. Cette définition donne une bonne estimation de la manière dont les nœuds se déplacent les uns par rapport aux autres. La définition est la suivante:
  - Si les nœuds sont en mouvements pour une certaine durée de temps, alors la mobilité est la moyenne des changements de distance entre les nœuds.
  - Soit $A_x(t)$ la moyenne des distances séparant un nœud x de tous les autres à l'instant t(N est le nombre de nœuds).

$$A_x(t) = (1/N-1)\sum_{i=1}^{n} Dist(n_x, n_i)$$

  - Soit M_x la mobilité moyenne d'un nœud x durant la simulation (dans la formule suivante Δt représente le pas du calcul)

$$M_x = (1/(T-\Delta t)) \sum_{t=0}^{T-\Delta t} | A_x (T+\Delta t)- A_x(t)|$$

  - Donc la mobilité moyenne est:

$$Mob = (1/N) \sum_{i=1}^{n} M_i$$

  - A partir de cette formule, nous avons:
    - ✓ mobilité faible : $0< Mob<=3$
    - ✓ mobilité moyenne : $3<Mob<=8$
    - ✓ mobilité élevée : $8<Mob$
  - Pour notre simulation, nous avons proposé quatre valeurs de mobilité: 0, 1.7, 5, 10 correspondant respectivement aux cas : statique, mobilité faible, mobilité moyenne et mobilité élevée.
  - Les scénarios de mouvements que nous avons utilisés ont été générés avec l'outil « setdest » (générateur aléatoire) fourni avec NS2. Cet outil se montre très utile lorsqu'une multitude de tests doivent être exécutés pour une mobilité donnée. En effet, quand une scène de mouvement de nœuds est générée avec « setdest » elle est sauvegardée dans un fichier sur lequel la mobilité peut être mesurée séparément. Par la suite, les tests de simulations seront exécutés suivant le fichier ainsi généré.

- La charge : La charge est définie par une loi poissoniènne de paramètre λ. Les valeurs prises par ce dernier pour notre simulation sont: 0.07, 0.10, 0.125, 0.165, 0.25, 0.5, 1 correspondants respectivement aux intervalles (1/λ) 15s, 10s, 8s, 6s, 4s, 2s, 1s durant lesquels un nœud, après satisfaction, génère une nouvelle demande d'entrée à une session.

3.3. Mesures réalisées

Afin d'évaluer les performances de notre protocole, nous avons réalisé un certain nombre de tests. Deux sortes de mesures ont été réalisées : les mesures préalable et mesures effectives. La première sorte de mesures a été réalisée dans le but de faciliter l'interprétation des mesures effectives.

<u>Mesures préalables</u>

- Taux de sous-utilisation d'une session : Cette métrique indique le nombre d'invitations non aboutis (i.e. le nombre de nœud qui n'ont pas reçu leurs invitations alors qu'ils ont été invités par l'initiateur). Elle est calculée de la manière suivante : **nb_Invit_Non_Abouti/nb_Invite** = nombres *Invite_Non_Abouti*/nombre de messages *Invite()***.**
- Taux de *Release()* non aboutis : Cette métrique indique le nombre de message *Release()* non aboutis (c'est le nombre de message *Release()* ignorés). Elle est calculée de la manière suivante : **nb_Release_Non_Abouti/nb_Release** = nombres de *Release_Non_Abouti*/nombre de messages *Release()*.

<u>Mesures effectives</u>

- Taux de redemandes **(nb_redemandes/SC)** : Cette métrique indique le taux des redemandes d'entrées en SC liée au nombre total d'entrées en SC. Cette métrique est calculée comme suit : **nb_redemandes/SC**= nombre total de redemandes/nombre de demandes.
- Le nombre de messages redondants (**nbMsgRedondants/SC)** : Cette métrique indique le nombre de messages redondants par nœud, c'est à dire la différence (en nombre de messages) entre les messages **REQUEST** reçus et ceux traités. Cette métrique est calculée comme suit: **nbMsgRedondants/SC** = (le nombre de messages *Request()* reçus- le nombre de messages *Request()*traités) / nombre d'entrées en SCs.
- Le nombre de messages par section critique **(nb_MSG/SC)**: Cette métrique représente le nombre moyen de messages nécessaires pour la réalisation d'une SC, il s'agit de tous les messages échangés. Elle est calculée de la façon suivante :**nb_MSG/SC=(**nombre de messages de demandes d'entrée en SC + nombre de messages de redemandes +le nombre de sauts effectués par les jetons + nombre de sauts effectués par les messages *Invite()* + nombre de sauts effectués par les messages *Release()*)/nombre d'entrées en SCs.

- Le délai de synchronisation **(Délai_Syn)** : Cette métrique indique le nombre de sauts moyen effectués par le jeton Distributeur et forcément par les messages **Invite** entre deux sessions. elle est calculée de la manière suivante : **Délai_syn**=(nombre de sauts effectués par le jeton Distributeur + le nombre de sauts effectués par les messages *Invite()*)/nombre d'entrées en SCs.

- Le taux de satisfaction **(nb_SC/demande)** : Cette métrique indique le taux de satisfaction des demandes d'entrées en SC. Elle est calculée de la manière suivante : **nb_SC/nb_demandes** = nombre d'entrée en SCs/nombre de demandes, (les redemandes ne sont pas considérés).

4. Résultats et interprétations

4.1. Taux de sous-utilisation d'une session

Comme le montre le graphe de la figure 5.1, le taux de sous-utilisation d'une session est croissant et donc "proportionnel" à la charge. Par ailleurs, nous pouvons remarquer que pour une mobilité nulle le taux de sous-utilisation d'une session est nul, ceci est expliqué par l'absence de rupture de route et donc toutes les invitations vont atteindre leurs destinataires. D'un autre côté, plus la mobilité est élevée plus nous avons de messages *Invite ( )* non aboutis, ceci est dû aux nombres d'invitations ignorés : plus la mobilité est élevée plus il y a risque de perte de route vers le destinataire et donc d'ignorer l'invitation.

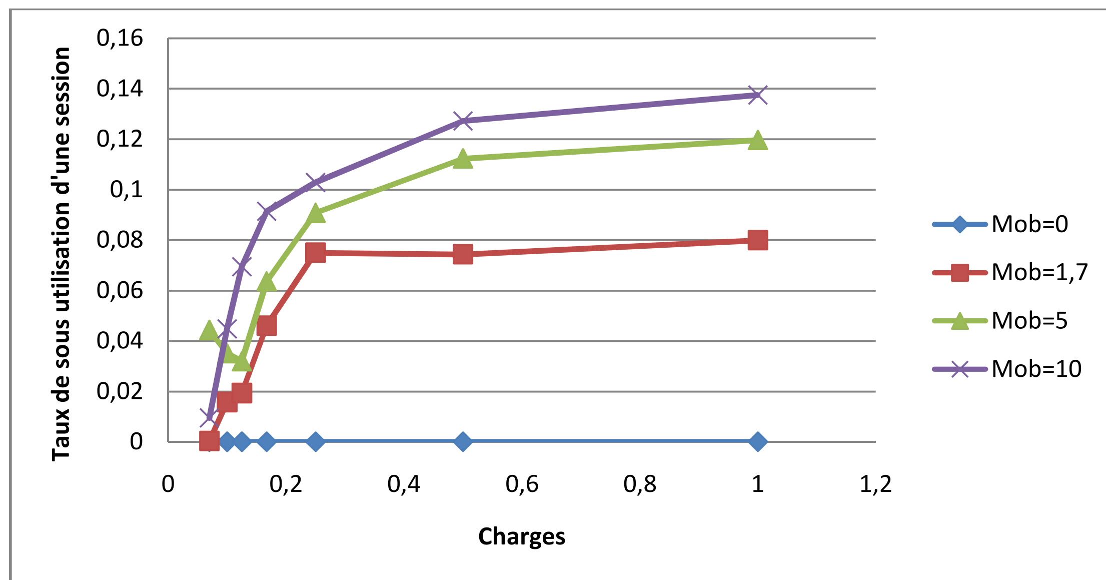


Figure5.1 : Les graphes représentant le taux de sous-utilisation d'une session.

D'autre part, plus la charge est élevée plus nous avons de demandes concernant la session courante dans la file et donc plus de nœuds seront invités. Nous remarquons aussi que pour les cas de fortes charges, une certaine stabilisation des graphes s'annonce. Ceci est dû au fait que le nombre de demandeur d'une même session dans la file d'attente du détenteur du jeton distributeur peut dépasser le nombre k et donc même s'il y'a plus de k nœuds demandeurs de la session en cours seulement k seront invités.

4.2. Taux de Release non aboutis

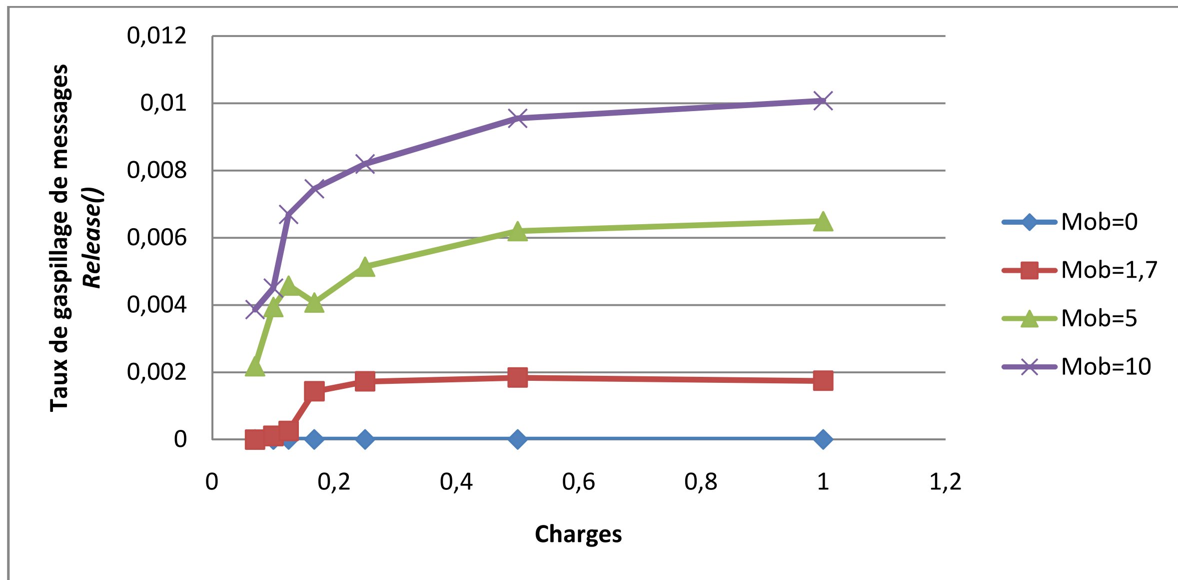


Figure 5.2 : Les graphes représentant le taux de messages Release( ) non aboutis.

Comme le montre la figure 5.2, le taux de messages *Release( )* non aboutis est faible comparé au taux de messages *Invite( )* non aboutis (figure 5.1). Ceci est expliqué par le faite que le nœud qui doit envoyer un message *Release( )*a déjà reçu une invitation juste avant d'entrer en section critique, donc une mise à jour des routes vers le destinataire a été faite. Nous remarquons aussi, que pour une mobilité nulle le taux de messages release ignorés est nul. Ceci est dû à la stabilité des routes et donc tous les messages release aboutissent.

D'un autre côté, les tracés montrent que plus la mobilité est élevée plus le taux de release ignorés est élevé. Cette hausse est une conséquence de la forte fréquence de changements de routes.

Par ailleurs, plus la charge est élevée, plus le nombre d'accès simultané des nœuds à une session (suite à des invitations) est élevé, ce qui implique un nombre important de messages *Release( )* échangés et par conséquence plus de messages de ce type sont ignorés. Aussi, une certaine stabilisation s'annonce pour les fortes charges en résultat du nombre limité des nœuds invités (limité à k). Même si le nombre de nœuds demandeurs de la même session est supérieur à k seulement k d'entre eux seront invités et pourront probablement accéder à la ressource simultanément.

4.3. Le taux de redemandes (nbRedemandes/nœud)

Le taux de redemande d'une session est représenté par la figure 5.3.

A partir de la figure 5.3, nous pouvons remarquer que, globalement les tracés sont décroissants pour les faibles et moyennes charges.

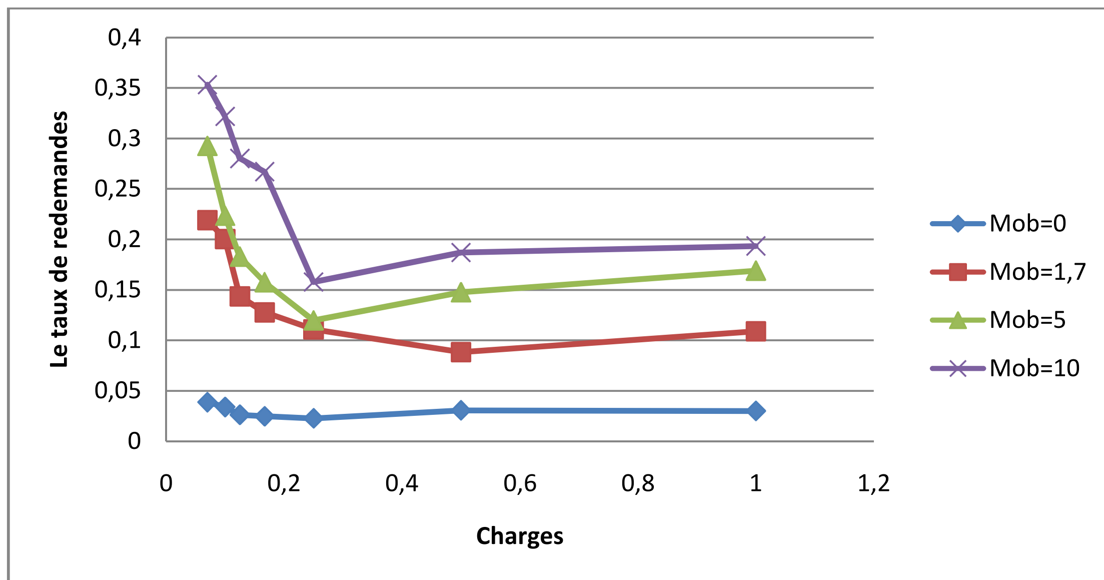


Figure 5.3 : Les graphes représentant le taux de redemandes d'une session.

Comme le montre la figure ci-dessus, le taux de redemande est décroissant pour les faibles et moyennes charges. Pour les faibles charges nous constatons que le nombre de redemandes est élevés, ceci est expliqué par le fait que le nombre d'entrées en section critique est faible par rapport aux autres charges. Plus la charge est élevée plus on permettra un accès concurrent à la session courante (sans dépasser le nombre k), arrivé à un certain seuil (k nœuds), le nœud initiateur ne pourra plus inviter de nœuds même s'il existe des demandeurs de la session en cours dans sa file. Donc, ces nœuds-là ne pourront accéder à leur session qu'a la prochaine ouverture de celle-ci. Ce qui implique plus de redemandes vont être formulées par les nœuds ce qui explique la croissance des tracés pour les fortes charges.

D'un autre côté, plus la mobilité est élevé plus les demandes mettront du temps pour atteindre le jeton récolteur. Par ailleurs, les invitations auront aussi des difficultés à atteindre le destinataire (comme le montre le graphe de sous-utilisation de session) dans le cas où les nœuds seront invités. Aussi, dans le cas de fortes mobilités le jeton récolteur risque de ne pas atteindre son destinataire (la route vers ce dernier est brisée et donc un autre destinataire sera élu). En conséquence, plus la mobilité est élevée plus le taux de redemandes est élevé.

4.4. Le nombre de messages redondants (nbMSGRedondant/SC)

Les tracés prennent une allure décroissante. Le nombre de messages redondants est faible pour les mobilités nulles et faibles, ceci est expliqué par le fait que les routes sont stables.

Plus la mobilité est élevée plus le risque de rupture de route augmente, plus le risque de perdre la trace vers le jeton récolteur est important. Ce qui engendre plus de messages redondants.

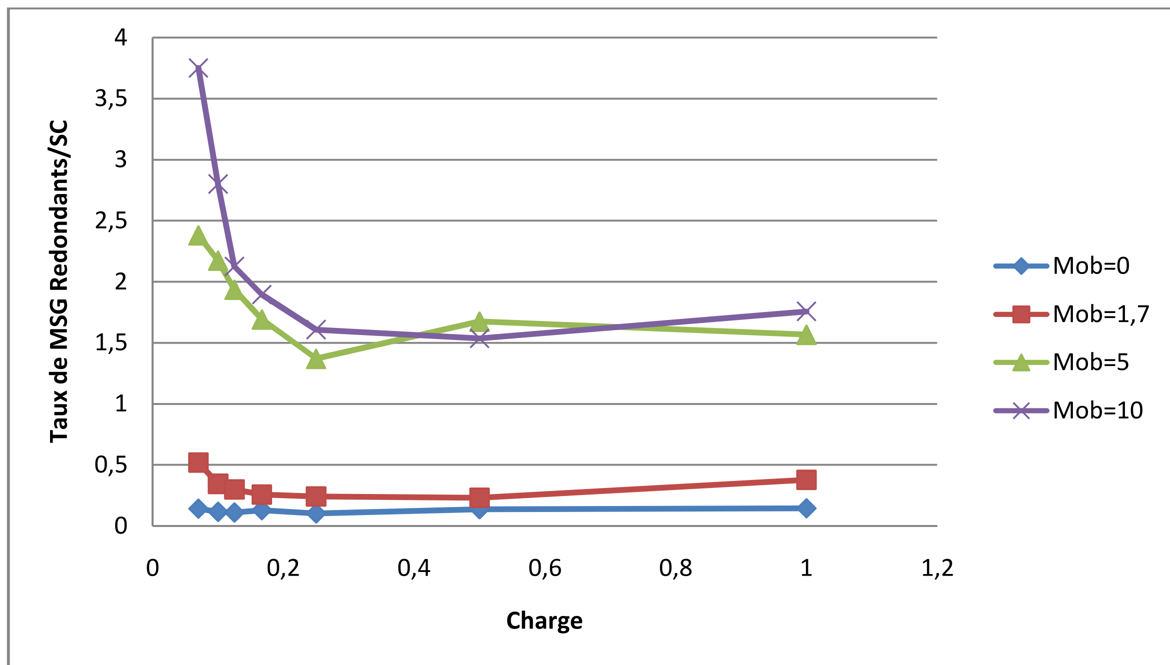


Figure 5.4 : les graphes représentant le nombre de messages redondants par SC.

Pour les faibles charges, le taux de messages redondants est élevé, car le nombre d'entrées en section critique dans ce cas est faible, en plus la trace vers le détenteur du jeton récolteur risque d'être perdue. Plus la charge est élevée, plus les nœuds feront des demandes. Par conséquent, ces derniers auront connaissance de l'emplacement du jeton récolteur. Par ailleurs, le nombre d'entrées en section critique augmente.

4.5. Le nombre de messages par nœud et par entrée en section critique (nbMSG/SC)

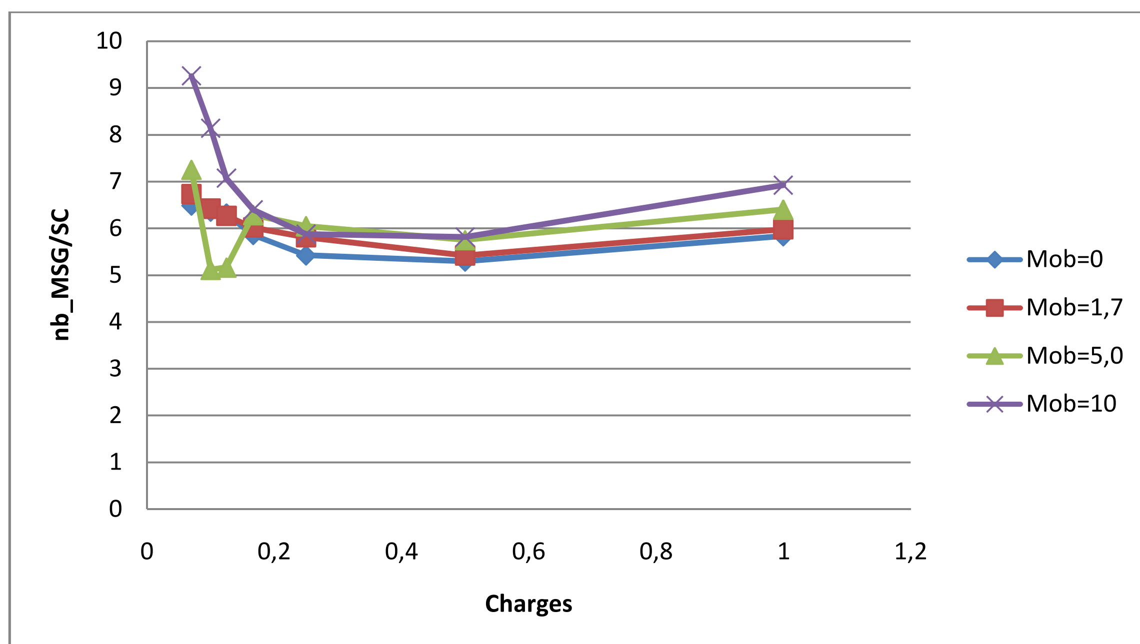


Figure 5.5 : Les graphes représentant le **nbMSG/SC**

Comme le montre le graphe de la figure 5.5, le **nbMSG/SC** est décroissant pour les faibles et moyennes charges et croissant pour les fortes charges.

Plus la mobilité est élevée, plus les demandes auront des difficultés pour atteindre le jeton récolteur. D'un autre côté, une partie des invitations va être ignorée en cause de perte de routes vers les destinataires et donc les nœuds concernés ne recevront pas leurs invitations. En résultat, ils ne pourront pas participer à la session courante et donc ils attendront la prochaine ouverture de cette dernière pour participer peut être à celle-ci. Ce qui va entrainer des redemandes. Aussi, nous avons le jeton distributeur qui doit suivre la trace du jeton récolteur. Dans le cas de mobilité élevée, cette trace risque d'être perdue et donc le jeton distributeur va devoir chercher le jeton récolteur ce qui va augmenter le nombre de sauts du jeton distributeur. Ce qui en résulte que plus la mobilité est élevée plus le nb_MSG/SC augmente. Néanmoins, nous pouvons remarquer que pour les fortes charges la mobilité n'a pas un grand impact sur le nombre de messages. En effet, même pour les fortes mobilités, quand la charge est élevée le jeton distributeur ne cherche pas beaucoup le jeton récolteur, puisque ce dernier ne va pas s'éloigner et donc le nombre de pas du jeton récolteur et distributeur ne sera pas très grand comparé aux faibles mobilités et puis le nombre de messages *Invite()* diminue légèrement pour les fortes mobilités (les messages *Invite()* ignorés).

Pour les faibles charges, le nombre de requêtes n'est pas très important dans le jeton récolteur et donc quand une session est ouverte le nombre de nœuds qui vont participer à cette session n'est pas très important en plus de la sous-utilisation de la ressource. Automatiquement, le nombre de message va augmenter vu que le jeton récolteur va s'éloigner du jeton distributeur pour satisfaire les demandes des nœuds et donc le jeton distributeur va devoir chercher le jeton récolteur dans le cas de perte de routes. D'un autre côté, plus la charge est élevée plus le jeton récolteur ne vas pas trop s'éloigner du distributeur et donc même dans le cas de perte de route, il le retrouvera plus vite.

Plus la charge est élevée, plus le nombre de nœuds demandeurs d'une même session augmente dans le jeton récolteur et donc à l'ouverture d'une session plus de nœuds vont utiliser cette session simultanément. Par ailleurs, plus la charge est élevée plus la probabilité qu'un nœud demandeur de la prochaine session à ouvrir soit dans le voisinage du nœud initiateur. Par conséquence, le nombre de saut du jeton récolteur est réduit et automatiquement le jeton distributeur mettra moins de temps pour retrouver le jeton récolteur après la fermeture de la session courante. Ceci explique le fait que les tracés soient décroissants pour les faibles et moyennes charges. Pour les fortes charges, nous remarquons que les tracés sont croissants. Cette croissance est un résultat de la contrainte k, même si un nœud initiateur d'une session a plus de k nœuds demandeurs de la session courante dans sa file, il ne pourra pas générer plus de k invitations. Les autres nœuds devront attendre alors la prochaine ouverture (ils devront attendre alors que l'un d'entre eux l'initie) ce qui implique plus de redemandes.

4.6. Le délai de synchronisation

A partir de la figure 5.5, nous constatons que les tracés ont une allure décroissante.

Nous remarquons que la mobilité a une influence limitée par rapport à ce paramètre. En effet, dans ce dernier, on prend en considération les invitations et le nombre de pas que fait le jeton distributeur. Plus la mobilité est élevée, plus le nombre de messages *Invite()* ignorés augmente et ces nœuds invités n'accéderont pas tous à la session courante (nombre de message *Invite()*) diminue) ce qui fait que le nombre d'entrées en section critique diminue aussi.

Plus la charge est élevée, plus le nombre d'invitations augmente et plus le nombre d'entrée en section critique augmente. De plus, le jeton distributeur cherchera moins le jeton récolteur ce qui fait que le délai de synchronisation est décroissants. Pour les fortes charges nous remarquons une légère décroissance ceci est expliqué par le fait que pour les fortes charges le nombre d'invite va être le même puisque un nœud ne peut pas inviter plus de k nœuds simultanément. Par contre, le nombre de pas du jeton distributeur va diminuer puisque le jeton récolteur ne va pas s'éloigner du jeton distributeur.

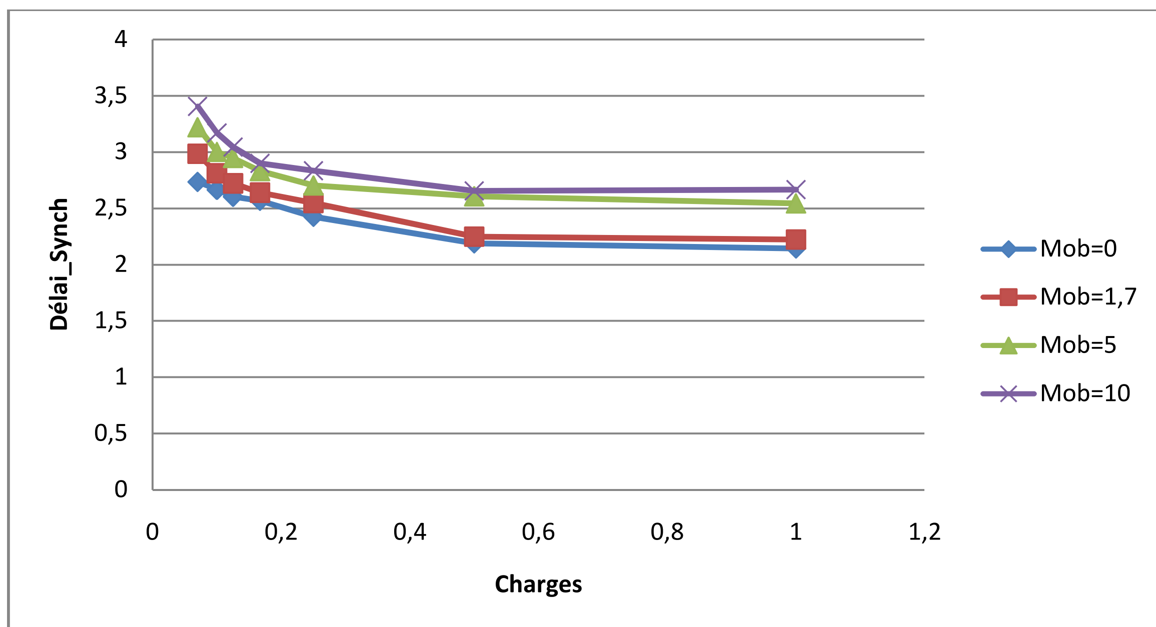


Figure 5.6 : Les graphes représentant les délais de synchronisation.

4.7. Taux de satisfaction

Les tracés prennent une allure croissante pour les cas de charges faibles et moyennes puis se stabilisent pour les fortes charges. Ceci est expliqué par le fait que plus la charge est élevée, plus on permettra à des nœuds d'accéder concurremment ce qui fait que les nœuds demandeurs auront plus de chance d'accéder à la session qu'ils demandent dès leur première tentative. Une certaine stabilisation est remarquée pour les fortes charges ceci est expliqué par le fait que plus la charge est élevée, plus le nombre de nœuds demandant une session augmente (peut être supérieur à k). Ceci dit, le nœud initiateur ne pourra satisfaire toutes les demandes intéressées par la même session que lui et donc les nœuds vont redemander.

Plus la mobilité est élevée plus le taux de satisfaction diminue ceci est dû au fait que plus de mobilité implique que les demandes mettront plus de temps pour atteindre le jeton récolteur, en plus des messages *Invite( )* ignorés et plus de routes brisées vers le prochain initiateur et donc plus de risque qu'un autre initiateur soit élu. Puisque pour qu'un nœud puisse accéder à une session qu'il demande, il faut soit qu'il reçoit une invitation, soit qu'il reçoit les deux jetons.

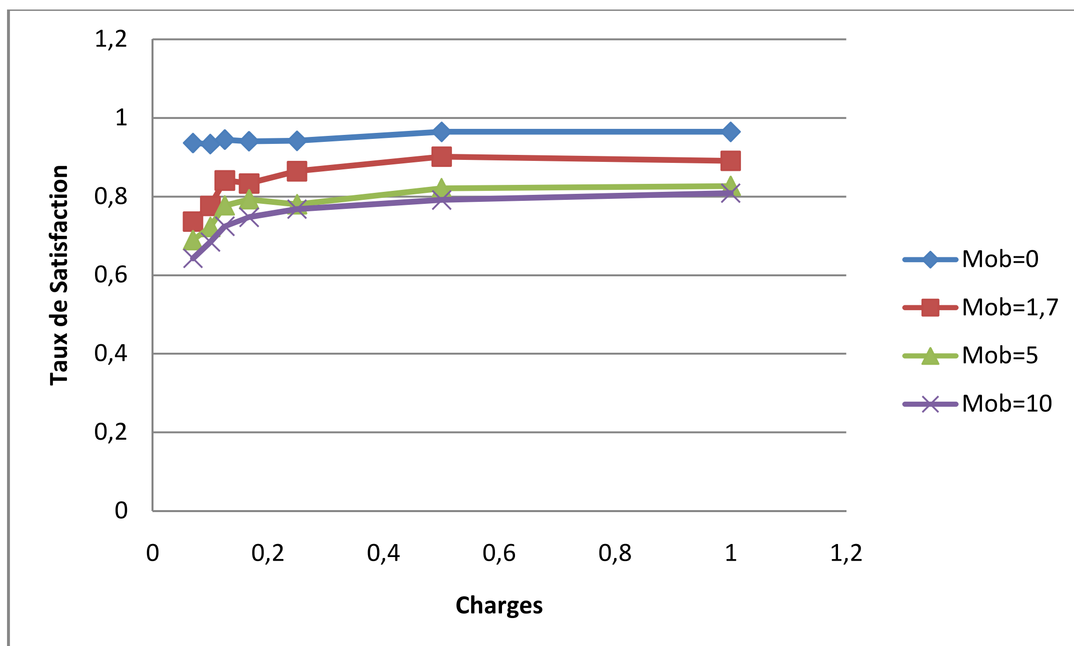


Figure 5.7 : Les graphes représentant Le taux de satisfaction.

5. Discussion et conclusion

- D'après les graphes des différentes mesures effectuées, le protocole donne ses meilleurs résultats pour des charges moyennes. Ceci est dû, comme on l'a expliqué précédemment, à la densité des demandes dans la file de l'initiateur. Ce dernier permet aux nœuds demandeurs de la même session que lui d'y accéder simultanément. Cependant, il ne peut pas permettre à plus de k nœuds d'utiliser une même session simultanément. Les autres nœuds demandant cette même ressource attendront soit la réception des jetons pour permettre aux autres de participer soit une autorisation, ce qui peut engendrer des redemandes. En fait, c'est pour la raison citée qu'on remarque que pour les fortes charges les résultats sont légèrement moins meilleurs. Par rapport aux faibles charges, nous remarquons que les résultats sont moins meilleurs que pour les moyennes et fortes charges. Dans ce cas, le nombre de demandes dans la file du jeton récolteur n'est pas important, ce qui implique que moins de messages *Invite()* seront envoyés et donc moins de nœuds vont utiliser la ressource simultanément (le nombre d'entrée en section critique diminue). Malgré le nombre réduit de nœuds qui accède à la section critique suite à des

invitations, les demandes de ces nœuds parcourent une bonne partie du réseau pour atteindre le jeton récolteur. De plus, lorsqu'une session est élue, le nœud initiateur de la session en cours prépare la prochaine session en envoyant le jeton récolteur au nœud le plus proche de l'initiateur et c'est à ce nœud d'inviter les autres nœuds à participer à cette session après son ouverture. Comme nous venons de le dire, le nombre de messages de demande concernant une même session n'est pas important et donc il se peut que les demandeurs de la session élue ne soient pas dans le voisinage de l'initiateur et donc le jeton peut ne pas suivre sa route. Par conséquent, s'il ne peut pas la récupérer, il choisira un des nœuds parmi ceux qui demande cette même ressource pour être initiateur. Mais comme le jeton distributeur suit la trace du jeton récolteur, il y'a risque de perte de cette trace et surtout si le jeton récolteur c'éloigne du jeton distributeur. Rappelons que les autorisations d'accès se font en envoyant des messages d'invitations.

- Par ailleurs, nous remarquons que la mobilité augmente, plus la recherche du jeton récolteur se fera par les demandeurs (à cause des pertes de routes) et le jeton distributeur cherchera le jeton récolteur et aussi moins de messages d'invitations atteindront leurs destinataires et enfin moins de messages *Release()* atteindront leurs destinataires. Ce qui retardera de purger la file et donc de couter en messages. Dans un tel type de réseau, les changements se font fréquemment surtout pour les mobilités fortes et afin d'éviter les diffusions, nous avons choisi d'abandonner les messages *Invite()* (respectivement *Release()*) si après un certain nombre de tentatives la route n'est pas réparée (ni le nœud ni ses voisins ont une idée récente sur le détenteur du jeton distributeur).

- Rappelons que le protocole de Gk-EM est un service continu. D'après les simulations, il passe par une brève phase transitoire avant d'arriver à la phase continue. Nous avons remarqué que la complexité moyenne de messages est sensiblement élevée durant la phase transitoire pour atteindre un taux raisonnable. Ceci, est expliqué par le fait que les différentes routes ne sont pas construites dans la phase transitoire, donc un grand nombre de messages de type *Request()* sera généré. La stabilité sera atteinte progressivement avec la construction des routes par la circulation du jeton récolteur. Etant donné que les mesures réalisées ont inclues les messages de la période transitoire, un impact a été enregistré dans les différentes complexités.

- D'après les graphes obtenus des différentes simulations, on a remarqué que les résultats sont prometteurs en l'occurrence le nombre de messages, le délai de synchronisation, le taux de satisfaction,…etc. Les résultats obtenus sont en accords avec les principes et mécanismes proposé. Par ailleurs, nous avons noté globalement que le protocole est sensible à la mobilité mais reflète bien les résultats attendues.

Conclusion générale

Le travail réalisé, consiste en le développement d'une solution au problème de groupe k exclusion mutuelle adaptée aux réseaux mobiles ad hoc, qui tient compte d'un certain nombre de caractéristique de ces réseaux, en l'occurrence la mobilité (changement de liens), la panne de nœuds et l'isolement temporaire. Cependant, le partitionnement ne peut pas avoir lieu. Notre solution, s'écarte de toutes celles qui la précèdent : elle ne généralise ni adapte aucune autre solution, elle se veut indépendante de toute topologie. Elle est orientée jeton et suppose l'existence d'une seule paire de jetons dans le système. Elle est équitable, distribuée et symétrique.

Un nœud souhaitant accéder à une ressource, doit atteindre le détenteur du jeton récolteur. Pour des raisons de mobilité, le nœud demandeur doit alors découvrir la route vers le détenteur de ce jeton. Un nœud demandeur, pourra accéder à la ressource qu'il demande, s'il reçoit le jeton récolteur (il est qualifié d'initiateur) et dans ce cas il doit attendre le jeton distributeur pour pouvoir accéder à sa ressource ou s'il est invité par le nœud initiateur de cette ressource. Si après un certain temps, le nœud demandeur ne reçoit ni les jetons ni invitation, il fait un certain nombre de tentatives avant de soupçonner la perte des deux jetons. Dans le cas où la recherche indique que les deux jetons sont perdus, le nœud ayant le plus petit identificateur crée une autre paire de jetons. Autrement, c'est le détenteur du jeton existant qui sera responsable de créer le jeton perdu.

Nous avons introduit des mécanismes pour assurer la propriété de sûreté et favoriser la propriété de vivacité. Aussi, nous avons essayé de proposer une solution efficace.
Concernant la propriété de la sûreté, étant donné qu'il existe un unique initiateur (une seule paire de jetons) dans le réseau, et c'est ce même nœud qui invite d'autres nœuds, implique qu'au plus k nœuds peuvent exécuter une même ressource simultanément et donc cette propriété est alors garanti.
Quant à l'efficacité, un certain nombre de mécanismes ont été introduits afin de favoriser cette propriété. Parmi ceux-là, le timer (LifeSessionTimer) qui limite la durée de vie d'une session. Aussi, pour le choix de la prochaine session à ouvrir nous avons tenu en compte l'âge de la requête. En plus, les nœuds sont invités du plus ancien demandeur vers le plus récent.
Quant à l'efficacité, quand un nœud est qualifié d'initiateur, il invite les demandeurs de la même session que lui à accéder à celle-ci simultanément. Aussi, pour le choix de la prochaine session élue, nous avons tenue en compte le nombre de demandeurs de la session et ce pour permettre plus de concurrence.

L'étude de simulation a été effectuée en utilisant le simulateur NS2 sous UBUNTU. Dans cette simulation, il est question d'évaluer les performances de la solution proposée.

Le protocole de Gk-EM est un service continu. D'après les simulations, il passe par une brève phase transitoire avant d'arriver à la phase continue. Nous avons remarqué que la complexité moyenne de messages est sensiblement élevée durant la phase transitoire pour atteindre un taux raisonnable. Ceci, est expliqué par le fait que les différentes routes ne sont pas construites dans la phase transitoire, donc un grand nombre de messages de type *Request()*

sera généré. La stabilité sera atteinte progressivement avec la construction des routes par la circulation du jeton récolteur. Etant donné que les mesures réalisées ont inclues les messages de la période transitoire, un impact a été enregistré dans les différentes complexités.

.

D'après les graphes obtenus des différentes simulations, on a remarqué que les résultats sont prometteurs en l'occurrence le nombre de messages, le délai de synchronisation, le taux de satisfaction,…etc. Les résultats obtenus sont en accords avec les principes et mécanismes proposés. Par ailleurs, nous avons noté globalement que le protocole donne ses meilleurs résultats pour les moyennes charges et est sensible à la mobilité mais reflète bien les résultats attendues.

Le travail réalisé est loin d'être fini. On peut envisager le partitionnement qui est une caractéristique principale d'un réseau mobile ad hoc. Une comparaison de la solution proposée avec d'autres qui résolvent le problème GK-EM dans le même type de réseau, est d'une grande utilité afin de pouvoir la situer par rapport aux solutions qui la précèdent. Désormais, dans la littérature on a pu trouver qu'un seul travail réalisé pour le même problème dans le même type de réseau. C'est celui de [TNG09], malheureusement ce dernier c'est contenté d'une étude de correction de sa solution.

BIBLIOGRAPHIE

[Bad98] N . Badache. «La mobilité dans les systèmes répartis», *Technical Report, France*, 1998.

[NT87] M. Naimi and M.Trehel, «How to detect a failure and regenerate the token in the log (n) distributed algorithm for mutual exclusion», *Proc, of the second Int workshop on distributed algorithms, lecture notes in S, pages. 155-166*, 1987.

[TNG06] O. THIARE, M. NAIMI, M. GUEROUI, «A Group Mutual Exclusion Algorithm for Mobile Ad Hoc Networks», *IEEE 2nd International Conference on Systems, Computing Sciences and Software Engineering (SCS2'06) CISSE 2006 December 4-14 Bridgeport, USA,* 2006.

[CDPV01] S. Cantarell, A.K. Datta, F. Petit, and V. Villain, « Token based group mutual exclusion for asynchronous rings», *In proceedings of the IEEE Conference on Distributed Computing Systems (ICDCS)*, 2001.

[CR83] O.S.F. Carvalho and G. Roucairol, « On mutual exclusion in computer networks », Communications *of the ACM*, 26:146–147, 1983.

[DBC+93] N. Davies, G.S. Blair, K. Cheverst, P. Friday, A. Raven, and A. Cross, «Mobile open systems technology for the utilities industries», *In Proceedings of the IEEE Colloquium on CSCW Issues for Mobile and Remote Workers, London*, 1993.

[DBCF92] N. Davies, G.S. Blair, K. Cheverst, and A. Friday, «Supporting collaborative applications in a heterogeneous mobile environment», *Technical Report MGP- 94-18, Lancaster University, Computing Department Bailrigg*, 1992.

[FZ94] G.H. Forman and J. Zahorjan, «The challenges of mobile computing», *IEEE Computer Society*, 27 :38–47, 1994.

[GB81] E. Gafni and D. Bertsekas,« Distributed algorithms for generating loop-free innetworks with frequently changing topology», *IEEE Transactions on Communications*, pages 11–18, 1981.

[IB92] T. Imienlinski and B.R. Badrinath, «Querying in highly mobile distributed environments», *Proceedings of the 18th VLDB*, 1992.

[IB94] T. Imienlinski and B.R. Badrinath, «Mobile wireless computing: solutions and challenges in data management» *CACM*, 37:18–28, 1994.

[PER94] Charles E.Perkins and Pravin Bhagwat, «Highly Dynamic Destination Sequenced Distance-Vector Routing (DSDV) for Mobile computers». *In proceedings of the SIGCOMM'94 conference on communication architectures, protocols an application, pages 234-244, Aug 1994.*

[PER98] C. E. Perkins and E.M. Royer, «Ad hoc on demand distance vector (AODV) routing (Internet-Draft) », *Aug 1994.*

[JOH01] David B. Johnson, David A. Maltz and Josh Broch. « The Dynamic Source Routing for Multihop Wireless Ad Hoc Networks», *in ad hoc networking. Edited by Charles E. Perkins, chapter 5, pages 136-172. Addison-Wesley, 2001.*

[Jia02] J. R. Jiang, «A group mutual exclusion algorithm for ad hoc mobile networks», *In: Proc. of the 6th International Conference on Computer Science and Informatics, pp. 266–270* 2002

[Jia03] J.R .Jiang, « group k mutual exclusion for distributed systems», Proc. Of pages 392-074, 2003.

[Jou01] Y.J. Joung, «The congenial talking philosophers problem in computer networks», *Distributed Computing*, *vol. 15 paages.155–175*, 2001.

[Jou98] Y.J. Joung, «Asynchronous group mutual exclusion algorithm». *Proc. in the 17th Annual ACM symposium on Principles of Distributed Computing*, 1998.

[JSDP03] J. Beauquier, S. Cantarell, A.K. Datta, and F. Petit,« Group mutual exclusion intree networks», *Journal of Information Science and Engineering*, 19 :415–423, 2003.

[KM01] P. Keane and M.Moir,« A simple local spin group mutual exclusion algorithm», *Proceedings in the 18th annual ACM Symposium on Principles of Distributed Computing*, 2001.

[LAM78] L.Lamport, «Time, clocks, and the ordring of events in a distributed systems», *Communication of the ACM, vol. 21, no. 7, pages 558-565*, Juillet 1978

[MAE85] M.Maekawa, « A√n algorithm for mutual exclusion in decentralized systems», *ACM Transactions on Computer Systems*, *vol. 3 pages.145–159*, 1985.

[PB93] E. Pitoura and B. Bhargava, «Dealing with mobility», *Department of Computer Science, Perdue University*, 1993.

[RA 83] G.Ricart and K.A. Agrawala, « Auther response to 'on mutual exclusion in computer networks by Carvalho and Rocairol», *Communication of the ACM vol.26, no. 2 , pages. 147-148, Janvier 1983.*

[RA81] G.RICART, A.K. AGRAWALA, « An optimal algorithm for mutual exclusion in computer networks», *Communication of the ACM, vol.24, no.1, pages. 9-17*, Janvier 1981.

[RN99] A.Ranganath and M.Neeraj, « A Distributed Group Mutual Exclusion Algorithm using Surrogate-Quorums», *In Proceedings of the 13th International Parallel and Distributed Processing (IPPS/SPDP '99), pages . 539-543,* 1999.

[Site01] http://www.intel.fr/sans-fil

[Site02] http://www.isi.edu/nsnam/vint/index.html

[SK85] I. Suzuki and T, Kasami, « A distributed mutual exclusion algorithm », *ACM transactions on Computer Systems, vol. 3 no, 4, pages. 344-349*, Novembre 1985.

[TAN96] TANENBAUM Andrew S. Computer Networks, Prentice Hall, Inc. 1996, ISBN.

[WJ99] K.P. Wu and Y.J. Joung, « Asynchronous group mutual exclusion in ring networks », *Technical Report 28-01, Department of Information Management, National Taiwan University, Taipei, Taiwan*, 1999.

[WWV01] J. Walter, J. Welch, and N.H. Vaidya, « A mutual exclusion algorithm for ad hoc mobile networks », Departement of computer sciences, Texas A&M UNIVERSITY, College station, USA, 2001.

[WWV98] J. Walter, J. Welch, and N.H. Vaidya, « A mutual exclusion algorithm for ad hoc mobile networks», *Dial M for Mobility Workshop, Dallas TX*, 1998.

[COR95] M. Scott Corson and Anthony Ephermides, «A distributed routing algorithm for mobile wireless networks », wireless *networks, 1 (1): 61-81, Feb 1995.*

[PEA96] Z. J. Haas and M. R. Pearlman, « The zone rouring protocol (ZRP) for ad hoc networks »

[ZYG97] The zone routing protocol (ZRP) Zygmunt J. Haas / Marc R. Pearlman (November 1997, expire en mai 1998) traduit de l'anglais par Erwan Doceux (Juillet 2000) pour l'ESEO.

[KO98] Young-Bac KO and Nitin, « Location Aided Routing (LAR) in Mobile Ad Hoc

[MAL98] Networks »
N.Malpani, N.H. Aidia, Welch.J.L, « Distributed Token Circulation on Mobile ad hoc networks », 1998

[MAN96] D. Manivanan and M. Singhal, « A Decentralized token generation scheme for token based mutual exclusion algorithm », in international journal of computer systems Sciences and engineering, vol 11, pp: 45-55, 1996

[MIC02] P. Sarajevo, « Diikstra's algorithm for short paths, loading the standard package for graphs processing », Matimaticki Algoritmi , January 2002

.

```
################################################################################
Initialisation

##########################################################################

init_node()

begin

    if(i=0) then {

            Holder_i:=3;

            RNS:=1

            Token_Route:={};

            Token_Req_Set:={};

            Token_NS_Set:={};

            Token_Session_Set:={};

            Token_NLSS_Set:={};

            Token_Route_Set:={};

            Token_Last_Holder:=0;

            TokenD_Req_Set:={};

            TokenD_NS_Set:={};

            Token_Locked:=False;

            Token_id:=0;

            Token_idNS:=0;

    }else{

        Holder_i:=0;

    }end_if


    NS_i:=0;
```

$NLSS_i$:=0;

RNS_i:=0;

Request_NS_i:=0 ;

$State_i$:=IDLE ;

Nb_$attempts_i$:=0 ;

Father_$Route_i$:={};

N_i:={};

Q_i :={} ;

Nb_Inv_i :=0 ;

Token_ID_i:=0;

Token_ID_NS_i: = 0;

Token_old_i := {} ;

Token_old_NS_i := {} ;

$CandidateTokenID_i$:=0;

CandidateTokenID_NS_i:=0;

Ignored_$Neighbors_i$:={};

Token_$CreatedNS_i$:=0;

$Session_i$:=-1;

Searching_$Token_i$:= 0;

Blocked_$State_i$:=NULL;

end

```
##############################################################################
Demander l'accès à la SC
#########################################################################
Event Ask_For_CS( i, X)
begin
   if(SearchingToken_i=0) then {
     if (Nb_attempts_i = 0) then {
            State_i:=Requesting;     NS_i:=NS_i+1;
            if(NLSS_i = 0) then {
                  NLSS_i:=NLSS_i+1;
            }end_if
            Request_NS_i:=NS_i;    /* sauvegarder l'estampille du nœud i */
            Session_i:=X;              /* la session demandée est X */
     } end_if
     Nb_attempts_i := Nb_attempts_i +1 ;
     If(Holder_i=3) then {
              Set(LifeSessionTimer) ;
              Enter_CS( );           /* la fonction entrer en SC */
     }else{
           If (Next_i ∉ N_i ) then {
                  Next_i :=Select(N_i - Ignored_Neighbors_i) ;
            }end_if
            Send REQUEST(i, NS_i, RNS_i, NLSS_i, i, Nb_attempt_i, X) to Next_i ;
            Set( RequestTimer_i);      /* déclencher le timer de la demande de i */
     }end_if
```

}else{

BlockedState$_i$:= Ask_For_CS;   /* sauvegarder l'état pour le dèblocage */

}end_if

End

End_Event

##
Recevoir un message REQUEST

###

Event Recevoir_Request: REQUEST(j, NS$_j$, RNS$_j$, NLSS$_j$, Route$_j$, Nb_attempts$_j$, X)

bool Temp_insert :=False ;

begin

if(i!=Last(Route$_j$)) then{   /* il s'agit d'une requete provenant d'un autre noeud */

NS$_i$:=max(NS$_i$, NS$_j$) + 1;

If (RNS$_i$ < RNS$_j$) then {   /* l'émetteur a des information plus récentes

Send REQUEST(j, NS$_j$, RNS$_j$, NLSS$_j$, Route$_j$, Nb_attempt$_j$, X) to

Last(Route$_j$);

}else{

If((Holder$_i$=2)and(Session$_i$=X)and(LifeSessionTimer> EndLifeSession))

then {

If(Nb_Inv$_i$ <k) then{

Nb_Inv$_i$:=Nb_Inv$_i$+1;

VehicleInvite(j, NS$_j$, Route$_j$, NLSS$_j$, X);

}else{

AddRequest(j, NS$_j$, NLSS$_j$, Route$_j$, Nb_attempts$_j$, X); /* la file */

}end_if

```
}else {
    If(Holderi=1) then { /* le nœud i détient le jeton récolteur */
        Token_Req_Set:=Token_Req_Set ∪ j;
        Token_Session := Token_Session+ add_element(j);
        Token_Session(j):= X;
        Token_NS:=Token_NS+ add_element(j);
        Token_NS(j):= NSj;
        Token_NLSS:=Token_NLSS+ add_element(j);
        Token_NLSS(j):= NLSSj;
        Token_Route:=Token_Route+ add_element(j);
        Token_Route(j):= Routej;


            if (Statei=IDLE) then {
                VehicleToken(3);
            }end_if


    }else{      /* la requête doit suivre la trace du jeton récolteur
      If (Nexti ∉ Ni ) then {
         Nexti :=Select(Ni - Ignored_Neighborsi) ;
      }end_if
      Send REQUEST(j, NSj, RNSi, NLSSj, Routej U i, Nb_attemptsj, X)
         to Nexti ;
    }end_if
}end_if
} else {        /*le next ma retourné ma demande */
    Ignored_Neighborsi := Ignored_Neighborsi ∪ Nexti ;
```

```
        Next_i :=Select(N_i - Ignored_Neighbors_i) ;

        Send REQUEST(j, NS_j, RNS_j, NLSS_j, Route_j, Nb_attempt_j, X) to Next_i ;

   }end_if


End

End_Event

##########################################################################
Recevoir le message INVITE

#######################################################################

Event Receive_Invite:INVITE(j, Route_j, Invite_Route_j, NLSS_j, X)

list Temp_Route_i ;


begin

   if(Last(Route_j) != i)    then {

            if( Last(Route_j) ∈ N_i) then {

              Send INVITE(j, Route_j, Invite_Route_j, NLSS_j, X) to  Last(Route_j);

            }else{

                 Temp_Route := Route_j – i ;

                 VehicleInvite(j, Temp_Route_i, Invite_Route_j, NLSS_j, X)

            }end_if

   }else{

        If(NLSS_j= NLSS_i) then {  /* pour verifier que ce n'est pas un msg caduque

            Father_i := j;

            Release_Route_i := Invite_Route_j

            Enter_CS();

        }end_if
```

```
    }end_if

End

End_Event

################################################################################
Recevoir le jeton Récolteur
########################################################################

Event Receive_Token : TOKEN_RECOLTEUR(Token_id, Token_idNS,
        TokenR_Route, RNS, Token_Req_Set, Token_Session_Set,
        Token_Route_Set, Token_NS_Set,   Token_NLSS_Set, Token_Last_Holder)
Char Temp_Route_Invite_i ;
Char Temp_Route_i ;
int RNS_i;
bool break:=False;

begin

  If((Token_id= Token_ID_i) and (Token_idNS= Token_ID_NS_i)) then {/*jeton valide

      NS_i:=max(NS_i,max(Token_NS))+1;
      RNS_i:= RNS;
      If(last(Token_Route)= i)
          Holder_i :=1;

      }else{
```

```
                    Vehicle Token(1);   /* faire acheminer le jeton selon cette
                                                Procédure */
         }end_if

   } else{            /*  jeton non valide */
      For each((j ∈ Token_Old_i) and (!break )) do {
        If ((Token_Oldi(j) = Token_id) and (Token_Old_NSi(j)= Token_idNS)) then {
                   Break:=True;
       } end_if
      }done
      If (!break ) then { /*le jeton reçu est plus prioritaire
          Token_Locked :=False;              /* le jeton n'est pas verrouillé */
          Dest:= Token_Last_Holder;
          Token_Last_Holder := i;
          Token_Route :={};
          Send TOKEN_RECOLTEUR(Token_id, Token_idNS,  TokenR_Route,
                 RNS, Token_Req_Set, Token_Session_Set, Token_Route_Set,
                 Token_NS_Set,   Token_NLSS_Set, Token_Last_Holder) to dest;
      }end _if
   }end_if

 End
End_Event
```

##
Recevoir le jeton distributeur

```
########################################################################

Event   Receive_Token_D:   TOKEN_Distributeur(Token_id,   Token_idNS,      Token_Route,
TokenD_Req_Set, TokenD_NS,_Set)

Char Temp_Route_Invite_i ;

Char Temp_Route_i ;

int j:=1;

int Node :=-1 ;

int Temp_NLSS_i:=0;

begin

    If((Token_id= Token_ID_i) and (Token_idNS= Token_ID_NS_i)) then {

            If(Holder_i =1) then {

                Holder_i :=3;

                NS_i:=max(NS_i,max(Token_NS))+1;

                Node := GetNextSession( ); /* permet de retourner le nœud*/

                                        /*initiateur de la prochaine session /*

                UpdateQueues(3); /* MAJ récolteur et copies des requêtes Q_i*/

                Holder_i :=2;

                Vehicle Token(1);

                While ((Nb_Inv_i <k) and (!Empty(Q_i))) do{

                    Nb_Inv_i:=Nb_Inv_i+1;

                    Temp_Route_Invite_i :=Q_i [j].Route ;

                    Temp_Route_i := Temp_Route_Invite_i ∪ i;

                    Temp_NLSS_i :=Q_i [j].NLSS;   j++ ;

                    VehicleInvite(i, Temp_Route_i, Temp_Route_Invite_i,

                    Temp_NLSS_i , X);

                }done
```

```
            Set(LifeSessionTimer_i) ;
            Enter_CS();
        }else{   /* il n'est pas destinataire */
              Vehicle Token(2);

        }end_if

    }else{          /* jeton non valide */
       For each((j ∈ Token_Old_i) and (!break )) do {
          If ((Token_Old_i[j] = Token_id) and (Token_Old_NS_i[j] = Token_idNS))
             then {
                 Break:=True;
          } end_if
       }done
       If (!break ) then { /*le jeton reçu est plus prioritaire

           Token_Locked :=False;               /* le jeton n'est pas verrouillé */
           Dest:= Token_Last_Holder;
           Token_Last_Holder := i;
           Token_Route :={} ;
           Send TOKEN_Distributeur(Token_id, Token_idNS,  Token_Route,
                  TokenD_Req_Set, TokenD_NS,_Set, Token_Last_Holder) to dest;

       }end_if

    }end_if
```

```
End

End_Event

##############################################################################
Quitter la SC

#######################################################################

Event Leave_CS

begin

    Statei:=IDLE;  /* à la sortie de la SC, le nœud i devient au repos */

    Nb_attemptsi := 0;

    NSi:= NSi + 1 ;

    NLSSi:= NLSSi + 1 ;

    if (Holderi=2)   then{

          TokenD_NS,_Set [i] := NSi ;

          Nb_Invi= Nb_Invi - 1;

          If ((!Empty(Qi)) and (LifeSessionTimeri > EndLifeSession))  then {

              Nb_Invi:=Nb_Invi+1;

              Temp_Route_Invitei :=Qi [1].Route ;

              Temp_Routei := Temp_Route_Invitei ∪ i;

              Temp_NLSSi :=Qi [1].NLSS;    j++ ;

              VehicleInvite(i, Temp_Routei, Temp_Route_Invitei, Temp_NLSSi X);

          }else{

             if(Nb_Invi=0) then {

                  UpdateQueues(2);  /*  MAJ du jeton distributeur */

                  Vehicle_Token(2);        /* faire suivre le jeton distributeur la */

                                      /*trace du jeton récolteur   */
```

```
                }end_if

            }end_if

      }else{    /* Holderi=0 */

            VehicleRelease(i, Release_Routei, X) ;

      }end_if


End

End_Event


###########################################################################
Réception du message Release

###########################################################################Event
Receive_Release : RELEASE(j, X, Release_Routej)

list Temp_Release_Routei ;

begin

   If(Last(Release_Routej) = i) then {

            Nb_Invi= Nb_Invi - 1;

            If ((!Empty(Qi)) and (LifeSessionTimeri > EndLifeSession))  then {

                Nb_Invi:=Nb_Invi+1;

                Temp_Route_Invitei :=Qi [1].Route ; /* inviter le 1 élemt de la file */

                Temp_Routei := Temp_Route_Invitei ∪ i;

                Temp_NLSSi :=Qi [1].NLSS;

                VehicleInvite(i, Temp_Routei, Temp_Route_Invitei, Temp_NLSSi X);

            }else{

                if(Nb_Invi=0) then {

                     UpdateQueues(2);  /*  MAJ du jeton distributeur */

                     Vehicle_Token(2);        /* faire suivre le jeton distributeur la */
```

```
                                        /*trace du jeton récolteur   */
            }end_if
        }end_if



}else{
        Temp_Release_Route_i := Release_Route_j – i;
        VehicleRelease(j, Temp_Release_Route_i, X) ;


}end_if
 Session_i := -1 ;
End
End_Event

###############################################################################
#                       Fin de Request Timer
#########################################################################
Event End_Timer Request_Timer_i
Begin
If(Searching_Token_i=0) then {
    if(Nb_attempts_i modulo Request_Threshold) then {
      Ask_For_CS();
     }else{
            Searching_Token_i:=3;   /* recherche des jetons
            Route_i := i;
            Token_CreatedNS_i := Token_CreatedNS_i+1;
```

```
        CandidateTokenIDi := i;
        CandidateTokenID_NSi := Token_CreatedNSi;
        Awarei:=Ni ∪ i;
        For each( j ∈Ni) do {
         Send IsThereToken(i, Request_NSi, NLSSi, Routei,
          Nb_attemptsi, i, Token_CreatedNSi , X, 3, Holderi, Awarei) to j;
                    /* 3 Signifi qu'on cherche les 2  jetons */
        }done
        Set(SuspicionTimeri);
  }end_if

}else {
 BlockedStatei:= RequestTimeri;
}end_if
End
End_Event

################################################################################
#                       Fin de LifeSessionTimer
########################################################################
Event End_Timer WaitEndSession i
begin

if(Holderi=2) then {
        UpdateQueues(2); /* MAJ du jeton distributeur */
        Holderi=0;
```

```
            VehicleToken(2);
}end_if
End
End_Event
###########################################################################
Recevoir IsThereToken
########################################################################


Event Receive ISTHERETOKEN(j, Request_NSj, NLSSj, Routej, Nb_attemptsj,
                         Token_IDj, Token_ID_NSj,  X, SearchingTokenj, Holderj, Awarej)
bool Temp_insert := False ;
begin


If((Holderi!= SearchingTokenj) and (Holderi!=3)) then { /*i n'a pas le jeton Perdu
        Routej :=Routej ∪ i ;
        If (Holderi !=0 ) then { /*i a l'un des  jetons, prioritaire pour creation */
          If(SearchingTokeni=0) then {
                    If(SearchingTokenj!=3) then {
                      SearchingTokeni := SearchingTokenj ;
                    }else{
                      SearchingTokeni := 2 mod SearchingTokenj ;
                                   // pour dire que l'un des  jetons existe
                    }end_if
                   Set(SuspicionTimeri) ;
           }end_if
           CandidateTokenIDi :=i ;
```

```
   CandidateTokenID_NSi := Token_ID_NSi ;
   For each (k є (Ni – Awarej)) do {
    Send ISTHERETOKEN (j, Request_NSj, NLSSj,  Routej,
              Nb_attemptsj, i, Token_ID_NSi, X, SearchingTokeni, Holderi,
              Awarej ∪ Ni) to k ;
   } done
}else {
  If ((Holderj!=0) or (SearchingTokeni=0) or
     (SearchingTokeni!=0 and CondidateTokenIDi>j) then { /*j prioritaire*/
        If(SearchingTokeni=0) then {
           SearchingTokeni := SearchingTokenj ;
           Set(SuspicionTimeri) ;
        }else{
            If(SearchingTokeni!= SearchingTokenj) then {
                    SearchingTokeni :=3 ;
            }end_if
        }end_if
      CandidateTokenIDi := Token_IDj ;
      CandidateTokenID_NSi := Token_ID_NSj ;
      For each (k є (Ni – Awarej)) do {
         Send ISTHERETOKEN (j, Request_NSj, NLSSj,  Routej,
          Nb_attemptsj, Token_IDj, Token_ID_NSj, X, SearchingTokeni,
          Holderj, Awarej ∪ Ni) to k ;
      } done
  }end_if
}end_if
```

```
    }else{        il détient le ou les jeton(s) perdu
       if(SeachingTokeni=0) then {
               SeachingTokeni := SearchingTokenj ;
               Set(StabilisationTimeri) ;
       }end_if
       If(Holderi=1) then {
        Add Request(j, Request_ NSj, NLSSj, Routej, Nb_attemptsj,
                                        X) ; /* j'insert la Demande dans le jeton
       }end_if
    } end_if
end
End_Event
######################################################################
                   Fin de SuspicionTimer
######################################################################
Event End_Timer SuspicionTimer
Begin
Int Node ;
      UpdateQueues(0);  /* vider les files des jetons
      Token_IDi := CandidateTokeni ;
      Token_ID_NSi := CandidateToken_ID_NSi ;
      If(CandidateToken_IDi !=i) then {
                If( !(Nb_attemptsi modulo Request_thereshold)) then {
                          Set(RequestTimeri) ;
                }end_if
      } else{
```

```
            Created_Token( ) ;
            Holderi :=3;
            If(Statei= Requesting) then {
              If(SeachingTokeni =2)  then { /* on a crée le distributeur */
                Node :=GetNextSession( ); /* permet de donner la prochaine
                                 session*/
                UpdateQueues(3); /* MAJ récolteur et copies des requetes Qi*/
                Vehicle Token(1);
                While ((Nb_Invi <k) and (!Empty(Qi))) do{
                        Nb_Invi:=Nb_Invi+1;
                        Temp_Route_Invitei :=Qi [j].Route ;
                        Temp_Routei := Temp_Route_Invitei ∪ i;
                        Temp_NLSSi :=Qi [j].NLSS;   j++ ;
                        VehicleInvite(i, Temp_Routei, Temp_Route_Invitei,
                        Temp_NLSSi  , X);
                }done

              }end_if
              Set(LifeSessionTimeri) ;
              Enter_CS();
            }end_if
    }end_if
    SearchingTokeni := 0 ;
    Nb_attemptsi := 1 ;
    OnStopSearchingToken( ) ;
End
```

```
End_Event

##########################################################################
#####           Lien rompu
##########################################################################
Event Link_Down(i,j)
begin
     Ni := Ni-j;
end
End_Event


##########################################################################
######          Lien établi
##########################################################################
Event Link_Up( i, j )
begin
    Ni := Ni ∪ j;
    if(SearchingTokeni!=0) then {
                Awarei := Ni ∪ i;
                if(i=CandidateTokenIDi)  then {
                                ns := Request_NSi;
                                route := i;
                                attempts := Nb_attempts;
                }else{
                        k := CandidateTokenIDi;
```

```
                    ns := null;
                    nlss := null;
                   route := null;
                   attempts := null;
                   session := null;

              }end_if
              send Link_info(i, Token_IDi, Token_ID_NSi, SearchingTokeni,
CandidateTokenID_NSi, ns, nlss, Route, attempts, session, Awarei) to j;
  }else{
         if(i=GetChoosen(i, j)) then {
               send Link_info(i, Token_IDi, Token_ID_NSi, SearchingTokeni,
                    NULL, NULL, NULL, Holderi,  NULL, NULL) to j;
         }end_if
  }end_if
end
End_Event

#######################################################################
#########              Fin de Stabilisation Timer
#######################################################################
Event End_Timer Stabilisation_Timer
begin
                   SearchingTokeni := 0;
                   for each(j ∈ Ni) do  {
                           send THEREIsTOKEN(Ni ∪ i) to j;
```

```
                }done
                OnStopSearchingToken();
end
End_Event

#####################################################################
##########      Recevoir THEREIsTOKEN
#####################################################################
Event Receive THEREIsTOKEN(Aware_j)
Begin

    if(SearchingToken_i!=0) then {
           SearchingToken_i:= 0;
           Desactivate(Suspicion Timer);
           OnStopSearchingToken();
    }end_if
    for each (j ∈ (N_i-Aware_j)) do {
           send THEREIsTOKEN(Aware_j ∪ N_i) to j;
    }done
end
End_Event
#####################################################################
######           Recevoir Link_Info
#####################################################################
Event Receive Link_Info(j, Token_ID_j, Token_ID_NS_j, SearchingToken_j,
CondidateTokenID_NS_j, ns_j, nlss_j, route_j, attempts_j, holder_j, session_j, Aware_j);
```

```
Bool Temp_insert := False;

Integer l:=SearchingToken_j

Begin

If((Token_ID_i = Token_ID_j) and (Token_ID_NS_i = Token_ID_NS_j))  then {
    //jetons valident
  If(SearchingToken_j!=0) then {
    If((Holder_i!= SearchingToken_j) and (Holder_i!=3)) then { /*i n'a pas le jeton Perdu
      Route_j :=Route_j ∪ i ;
      If (Holder_i !=0 ) then { /*i a l'un des  jetons, prioritaire pour creation */
        If(SearchingToken_i=0) then {
              If(SearchingToken_j!=3) then {
                SearchingToken_i := SearchingToken_j ;
              }else{
                SearchingToken_i := 2 mod SearchingToken_j ;
                          // pour dire que l'un des  jetons existe
              }end_if
             Set(SuspicionTimer_i) ;
        }end_if
        CandidateTokenID_i :=i ;
        CandidateTokenID_NS_i := Token_ID_NS_i ;
        For each (k ϵ (N_i – Aware_j)) do {
         Send ISTHERETOKEN (j, Request_NS_j, NLSS_j,  Route_j,
               Nb_attempts_j, i, Token_ID_NS_i, X, SearchingToken_j, Holder_i,
               Aware_j ∪ N_i) to k ;
```

```
      } done
   }else {
     If ((Holder_j!=0) or (SearchingToken_i=0) or
        (SearchingToken_i!=0 and CondidateTokenID_i>j) then { /*j prioritaire*/
        If(SearchingToken_i=0) then {
              SearchingToken_i := SearchingToken_j ;
              Set(SuspicionTimer_i) ;
        }else{
              If(SearchingToken_i!= SearchingToken_j) then {
                        SearchingToken_i :=3 ;
              }end_if
        }end_if
        CandidateTokenID_i := Token_ID_j ;
        CandidateTokenID_NS_i := Token_ID_NS_j ;
        For each (k ϵ (N_i – Aware_j)) do {
            Send ISTHERETOKEN (j, Request_NS_j, NLSS_j,  Route_j,
             Nb_attempts_j, Token_ID_j, Token_ID_NS_j, X, SearchingTokeni,
             Holder_j, Aware_j ∪ Ni) to k ;
        } done
     }end_if
   }end_if
}else{        il détient le ou les jeton(s) perdu
    if(SeachingToken_i=0) then {
             SeachingToken_i := SearchingToken_j ;
             Set(StabilisationTimer_i) ;
    }end_if
```

```
        If(Holder_i=1) then {
          Add Request(j, Request_ NS_j, NLSS_j, Route_j, Nb_attempts_j,
                                            X) ; /* j'insert la Demande dans le jeton
        }end_if
     }end_if
  }end_if
}else{ /* jeton non valide
For each((j ∈ Token_Old_i) and (!break )) do {
          If ((Token_Old_i[j] = Token_id) and (Token_Old_NS_i[j] = Token_NS)) then {
                    Break:=True;
          } end_if
}done
If (!break )  then { /*le jeton reçu est plus prioritaire
                    Token_ID_i := Token_ID_j;
                    Token_ID_NS_i := Token_ID_NS_j;
                    send DeleteTOKEN(Token_ID_j, Token_ID_NS_j, N_i ∪ j);
                    if(Holder_i!=0) then {
                        Holder_i := 0;
                    }end_if
                    if(SearchingToken_i) then {
                          Desactivate(Suspicion Timer);
                          SearchingToken_i := False;
                          OnStopSearchingToken();
                    }end_if
}else{
     send Link_Info(j, Token_ID_j, Token_ID_NS_j, NULL, NULL,
```

```
                                        NULL, NULL, NULL) to j;

}end_if
end
End_Event
#######################################################################
#######            Recevoir DeleteToken
#######################################################################
Event Receive DeleteTOKEN(Token_IDj, Token_ID_NSj, Awarej)
Begin
For each((j ∈ Token_Oldi) and (!break )) do {
       If ((Token_Oldi[j] = Token_id) and (Token_Old_NSi[j] = Token_NS)) then {
             Break:=True;
       } end_if
}done
If (!break )  then { /*le jeton reçu est plus prioritaire
             Token_IDi := Token_IDj;
             Token_ID_NSi := Token_ID_NSj;
             Nb_attemptsi := 1;
             If(Holderi!=0) then {
                UpdatesQueues(0);
                Holderi := 0;
             }end_if
             for each (j ∈ (Ni-Awarej)) do {
               Send DeleteToken(Token_IDj, Token_ID_NSj, Awarej ∪ Ni) to j;
             }done
```

```
                Q_i := {} ; mise à jour des files
                If(SearchingToken_i) then {
                        Desactivate(Suspicion Timer) ;
                        SearchingToken_i := 0;
                        ONStopSearchingToken();
                }end_if
}end_if
end
End_Event
```

Les méthodes et procédures:

```
###########################################################################
#####                Entrer en SC
#####################################################################
Enter_CS
begin
State_i :=InCS;    /* le nœud i est dans la SC */
Desactivate(RequestTimer_i);    /* désactiver le timer de la requête car elle est
                                   satisfaite */
End


###########################################################################
#                donner le jeton au prochain
#####################################################################
 VehicleToken(l)
bool neighbor:=False
bool Retreived:=False
```

bool break:=False

integer dest:=-1;

begin

if($SearchingToken_i$=0) then {

if(l=2) then {

if(i=Last(Token_Route)) then{ /*le next ma retourné le Distri */

$Ignored_Neighbors_i$:= $Ignored_Neighbors_i \cup Next_i$;

$Next_i$:=Select(N_i - $Ignored_Neighbors_i$) ;

Send TOKEN_Distributeur(Token_id, Token_idNS,

Token_Route, TokenD_Req_Set, TokenD_NS,_Set,

Token_Last_Holder) to $Next_i$;

} else {

If ($RNS < RNS_j$) then {/* l'émetteur a des information plus récentes*/

Send TOKEN_Distributeur(Token_id, Token_idNS,

Token_Route, TokenD_Req_Set, TokenD_NS,_Set,

Token_Last_Holder) to Last($Route_j$);

}else{

If ($Next_i \notin N_i$) then {

$Next_i$:=Select(N_i - $Ignored_Neighbors_i$) ;

}end_if

Send TOKEN_Distributeur(Token_id, Token_idNS,

Token_Route $\cup$ i, TokenD_Req_Set, TokenD_NS,_Set,

Token_Last_Holder) to $Next_i$

}end_if

}else{

```
if(!Empty(TokenR_Route)) then { /* continuer d'acheminer le jeton sur sa route */
        if(Last(TokenR_Route) ∈ Ni )   then  {
                neighbor:=True;
                TokenR_Route:=Last(TokenR_Route);
        }else{
                TokenR_Route:= TokenR_Route-i;
                 neighbor:=First(TokenR_Route) ∈ Ni;
        }end_if
        if( ! neighbor) then  {
                Retreived:=Retreive_Route(TokenR_Route);
        }end_if
        if(neighbor or (!neighbor and Retreived)) then {
          If(l=1) then {
            Send TOKEN_RECOLTEUR(Token_id, Token_idNS,  TokenR_Route,
              RNS, Token_Req_Set, Token_Session_Set, Token_Route_Set,
              Token_NS_Set,   Token_NLSS_Set, Token_Last_Holder) to
               First(Token_Route);
          }else{
             If(l=3) then {
              Send TOKEN_RECOLTEUR(Token_id, Token_idNS, TokenR_Route,
               RNS, Token_Req_Set, Token_Session_Set, Token_Route_Set,
               Token_NS_Set,   Token_NLSS_Set, Token_Last_Holder) to
               First(TokenR_Route);
               Send TOKEN_Distributeur(Token_id, Token_idNS,  Token_Route,
                 TokenD_Req_Set, TokenD_NS,_Set, Token_Last_Holder) to
                 First(TokenR_Route);
```

```
    }end_if
  }end_if
}else{   /* la route vers le destinataire est brisée */
   For each((j ∈ Token_Req_Set) and (!break ) and (Token_Session_Set(j)
      = Token_Session_Set(Node))) do {
      neighbor:= First((Token_Route_Set(j)) ∈ (Ni – Ignored_Neighborsi));
      if(!neighbor) then{
        Retreived:=Retreive_Route(Token_Route_Set(j));
      }end_if
      if(neighbor or ( ! neighbor and Retreived)) then {
        TokenR_Route:= Token_Route_Set(j);
        Token_Last_Holder:=i;
        If(l=1) then {
          Send TOKEN_RECOLTEUR(Token_id, Token_idNS,
          TokenR_Route, RNS, Token_Req_Set, Token_Session_Set,
          Token_Route_Set, Token_NS_Set,   Token_NLSS_Set,
           Token_Last_Holder) to First(TokenR_Route);
        }else{
          If(l=3) then {
              Send TOKEN_RECOLTEUR(Token_id, Token_idNS,
              TokenR_Route, RNS, Token_Req_Set, Token_Session_Set,
              Token_Route_Set, Token_NS_Set, Token_NLSS_Set,
              Token_Last_Holder) to First(TokenR_Route);


              Send TOKEN_Distributeur(Token_id, Token_idNS,
              Token_Route, TokenD_Req_Set, TokenD_NS,_Set,
```

```
                        Token_Last_Holder) to First(TokenR_Route);
                  }end_if
                }end_if
                          break :=True ;
                 }end_if



}end_if


}else{
 BlockedState_i := VehicleToken ;
}end_if
end


################################################################################
#                       donner le message Invite au prochain
########################################################################
VehicleInvite(ID, Temp_Route_i, Temp_Route_Invite_i, NLSS_i, session_i)
bool neighbor:=False
bool Retreived:=False;


begin
if(SearchingToken_i=0) then {
     if(!Empty(Temp_Invite_Route_i)) then {
                   if(Last(Temp_Invite_Route_i) ∈ N_i )   then  {
                   neighbor:=True;
```

```
                Temp_Invite_Route_i := Last(Temp_Invite_Route_i);
            }else{
                neighbor:=First(Temp_Invite_Route_i) ∈ N_i;
            }end_if
            if( ! neighbor) then  {
                Retreived:=Retreive_Route(Temp_Invite _Route_j);
            }end_if
            if(neighbor or (!neighbor and Retreived)) then {
              Send INVITE(ID, Temp_Route_i, Temp_Invite_Route_i, NLSS_i, session_i )
                                    to First(Temp_Invite_Route_i);
            }end_if
      }end_if
}else{
      BlockedState_i := VehicleInvite ;
}end_if
End
################################################################################
#                       donner le message Release au prochain
##########################################################################
VehicleRelease(ID, Temp_Release_Route_i, session)
bool neighbor:=False
begin
if (SearchingToken=0) then {
   if(!Empty(Temp_Release_Route_i)) then {
            if(Last(Temp_Release_Route_i) ∈ N_i )   then  {
                neighbor:=True;
```

```
            Temp_Release_Route_i := Last(Temp_Release_Route_i);
        }else{
            neighbor:=First(Temp_Release_Route_i) ∈ N_i;
        }end_if
           if( ! neighbor) then  {
             Retreived:=Retreive_Route(Temp_Release_Route_i);
           }end_if
           if(neighbor or (!neighbor and Retreived)) then {
             Send RELEASE(ID, Temp_Release_Route_i, session)
              to First(Temp_Release_Route_i);
           }end_if
  } else {
       BlockedState_i := VehicleRelease;
  }end_if
End
```

##
Récupérer la route

##

```
bool Retreive_Route(Route)        /* l'argument route est entré par adresse */
set_j :={};
bool break :=False ;

begin
   for each((j :=Last(Route); j != First(Route); j :=Previous(j, Route))and !break) do{
      if(j∈ (N_i – Ignored_Neighbors_i))   then {
```

```
                set_j := set_j ∪ j ;
                Route :=Reverse(set_j) ;    /* on entre à ce if une seule fois */
                Break :=True ;
        }else{
                set_j := set_j ∪ j ;
        }end_if
    }done
    return break ;
end

##############################################################################
###             Ajouter une demande
########################################################################
bool AddRequest(j, NS_j, NLSS_j, Route_j, Nb_attempts_j, X)
bool Temp_insert :=False ;
begin

    CreatedNew_Element(j, Route_j, NS_j, NLSS_j, Nb_attempts_j, X) ;
    Sort(Q_i) ;

return Temp_insert ;
end

########################################################################
##########              GetChoosen
########################################################################
```

```
GetChoosen(i, j)

begin

  /* une programmation possible est la suivante: */

 if( i < j) then {

              return i;

}end_if

return j;

end


#######################################################################

##########                UpdateQueues

#######################################################################

UpdateQueues(x)

Begin

if (x = 2) then {

       for each (j ∈ Qi) do {

                      Token_Req_Set := Token_Req_Set ∪j;

                      Token_NS := add?_element(j);

                      Token_NS(j) := Qi(j).NS+1;

                      Token_Session := add?_element(j);

                      Token_Session(j) := Qi(j).Session;

                      Token_NLSS := add?_element(j);

                      Token_NLSS(j) := Qi(j).NLSS;

      }done

      Qi:={};

}else{
```

```
If(x=3)  then {
    For each((j ∈ TokenD_Req_Set) and (Token_NS_Set(j) < TokenD_NS_Set(j))
     do{
         Token_Req_Set := Token_Req_Set - j;
         Token_NS_Set(j) := TokenD_NS_Set(j);
    }done
    for each ((j ∈ Token_Req_Set) and (Token_Session_Set(j) =
       Token_Session_Set(node))) do {
          AddRequest(j, Token_NS_Set(j), Token_NLSS_Set(j), NULL, X);
    }done
}end_if

 end

#####################################################################
#######                            Create Token()
#####################################################################
Create Token
begin
    if(Searching_Token_i=1) then {
         Token_Route := {};
         Token_Req_Set := {};
         Token_Session_Set := {};
         Token_NS_Set := {};
         Token_NLSS_Set := {};
    }else{
```

```
        If(Searching_Token_i=2) then {
          TokenD_Req_Set:= {};
          TokenD_NS,_Set:={};
        }else{
              if(Searching_Token_i=3) then {
                Token_Route := {};
                Token_Req_Set := {};
                Token_Session_Set := {};
                Token_NS_Set := {};
                Token_NLSS_Set := {};
                TokenD_Req_Set:= {};
                TokenD_NS,_Set:={};
              }end_if
        }end_if
    }end_if
    Token_Last_Holder := 0;
    Token_Locked := False;
    Token_id := Token_ID_i;
    Token_id_NS := Token_ID_NS_i;

end

##########################################################################
#########            ONStopSearchingToken()
##########################################################################
ONStopSearchingToken();
```

```
begin
        Case(BlockedState_i) of
        {
                Ask_For_CS : Ask_For_CS(); break;
                RequestTimer : RequestTimer(i); break;
                VehicleToken : VehicleToken(); break;
                VehicleInvite : VehicleInvite() ; break;
                VehicleRelease : VehicleRelease () ; break;
        }end_case;
        BlockedSate_i := IDLE;
end



##########################################################################
#########           GetNextSession()
##########################################################################
GetNextSession();
begin
function Nb_Hops;

for each(j∈ Token_Session_Set) do {
    for each(i∈ Token_Req_Set) do {
          A[j]:= A[j]+ Token_NS_Set(i);
          B[j]:= B[j]+ Nb_Hops (Token_Route_Set(i));
          C[j]:= C[j]+ Token_NLSS_Set(i);
```

```
        m++;
     }done
     A[j]:= A[j] / m;
     B[j]:= B[j] / m ;
     C[j]:= C[j] / m;
     m:=0;
  }done
Session:= Token_Session_Set(0);
a:=A(0);    b:=B(0);     c:=C(0);
j:=1;
For each(j∈ A) do {
    If((A(j) < a) or ((A(j) = a) and (B(j)<b)) or ((A(j) = a) and (B(j)=b) and (C(j)< c)))
      then {
          a:= A(j);
          b:= B(j);
          c:= C(j);
          session:= Token_Session_Set(j);
      }end_if
 }done;
 node:= Token_Session_Set(0);
  i:=1;
  for each(i∈ Token_Req_Set )  do {
    if((Token_Session_Set(i) = session) and (Nb_Hops (Token_Route_Set(i))< node)
     then {
           node:= Token_Req_Set(i);
     }end_if
```

```
}done

return node;

end
```