N° d'ordre : 03/2008-M/IN

REPUBLIQUE ALGERIENNE DEMOCRATIQUE ET POPULAIRE

MINISTERE DE L'ENSEIGNEMENT SUPERIEUR ET DE LA RECHERCHE SCIENTIFIQUE
UNIVERSITE DES SCIENCES ET DE LA TECHNOLOGIE HOUARI BOUMEDIENE

FACULTE D'ELECTRONIQUE ET INFORMATIQUE

Mémoire

Présenté pour l'obtention du diplôme de MAGISTER
En : Informatique
Spécialité : Programmation et système
Par MERZOUG Amel

Thème

La h-k exclusion mutuelle dans les Réseaux mobiles ad hoc

Directeur de thèse : Mr Mahfoud BENCHAIBA

Soutenu le 10/05/2008, devant le jury composé de :

Mr Nadjib BADACHE, Professeur à l'USTHB Président
Mr Mahfoud BENCHAÏBA, Chargé de Cours à l'USTHB Rapporteur
Mr Slimen LARABI, Maître de Conférences à l'USTHB Examinateur
Mr Hamid AZZOUNE, Maître de Conférences à l'USTHB Examinateur
Mr Djamel TANDJAOUI, Chargé de Recherche au CERIST Invité

A mon mari Abdelkrim, pour son éternel soutien,
A mes enfants, Safia et AbdAllah, pour m'avoir apporté quelques années de maturité et de « recul » très enrichissantes,

A ma mère, à ma mère, à ma mère, à mon père.

Remerciements

Je tiens tout d'abord à remercier Monsieur Mahfoud BENCHAIBA, chargé de cours à l'Université des Sciences et de la Technologie Houari Boumediènne, pour m'avoir fait confiance, en me proposant ce sujet. Je le remercie également pour sa disponibilité, son aide, ses conseils précieux, ses critiques constructives, ses explications, ses suggestions pertinentes et surtout pour sa patience et son indulgence.

Mes remerciements vont également aux membres du jury qui ont bien voulu examiner et évaluer ce travail.

Je tiens particulièrement à témoigner ma profonde gratitude à mes responsables au sein de mon travail, pour leur compréhension et leur aide.

Je tiens à remercier de même toutes les personnes qui ont contribué à la réalisation de ce modeste travail.

SOMMAIRE

Liste des figures

Liste des tableaux

Introduction générale

Ces dernières années ont vu un vaste déploiement des architectures et systèmes distribués dans le paysage informatique. Ces systèmes répondent à des besoins très précis :

• *Partage des ressources :* Le réseau est l'élément crucial permettant aux utilisateurs de partager des ressources lourdes et coûteuses.
• *Communication :* Le monde d'aujourd'hui est un monde où les besoins en communication sont de plus en plus importants. Les réseaux et les systèmes distribués vont répondre à cette demande en proposant les outils pour communiquer (*talk*, *mail, etc* ...), mais également des possibilités pour se partager des informations.
• *Evolutivité :* Les entreprises, et plus généralement les structures utilisant l'informatique sont en plein essor. Elles exigent des systèmes informatiques un maximum d'évolutivité en terme de poste de travail, de puissance de calcul, d'espace de stockage, d'intégration des nouvelles technologies,...
• *Nomadisme :* Un nouveau besoin lié à la complication de la vie actuelle notamment en terme de contraintes de déplacement et autres, il s'agit du nomadisme. Il consiste à permettre le traitement et l'accès à l'information tout en étant en mouvement. D'où la naissance des réseaux mobiles et particulièrement les réseaux mobiles ad hoc.

Un réseau mobile ad hoc se compose de nœuds mobiles qui peuvent communiquer entre eux en envoyant directement des messages à travers une liaison sans fil, ou à travers une séquence de liens sans fil incluant un ou plusieurs nœuds intermédiaires. Une liaison sans fil se rompe lorsqu'un nœud se déplace de sorte qu'il ne soit plus dans la portée de transmission de son correspondant. En échange, une liaison sans fil se forme quand un nœud se déplace de sorte qu'il soit dans la portée de transmission d'un autre nœud.

Un réseau mobile ad hoc possède un certain nombre de caractéristiques qui imposent de nouvelles exigences dans l'implémentation des algorithmes distribués. Les caractéristiques les plus importantes sont : Le changement fréquent de la topologie du réseau, la déconnexion fréquente d'un hôte, le partitionnement possible du réseau et les ressources modestes (CPU, espace de stockage, source d'énergie et la bande passante).

Occasionnellement, un nœud a besoin d'accéder à une ressource partagée, tel qu'une imprimante partagé ou une table partagée. Le problème de contrôle qui permet aux nœuds de partager une ressource d'une manière mutuellement exclusive, c-à-d, accédée par au plus un nœud à la fois, est appelé le problème de l'exclusion mutuelle. S'il y a k, $k \geq 1$, copies identiques de la ressource partagée, tel que k-licences utilisateur d'un logiciel ou k-canaux de communication, alors, au plus k nœuds peuvent accéder aux ressources partagées à la fois. Le problème de contrôle des nœuds afin qu'au plus k nœuds accèdent aux ressources partagées simultanément est appelé le problème de la k-exclusion mutuelle. Parfois, un nœud nécessite l'accès à h ($1 \leq h \leq k$) copies parmi les k ressources partagées à la fois. Le problème de contrôle des nœuds afin que chaque nœud puisse acquérir le nombre désiré de ressources (et le nombre total des ressources accédées simultanément par les nœuds ne doit pas excéder le nombre total de ressources partagées) est appelé le problème de la h-k-exclusion mutuelle ou le problème de l'allocation de h parmi k ressources [RAY91a].

Le problème de la h-k exclusion mutuelle est alors une généralisation du problème de la k-exclusion mutuelle qui est lui-même une généralisation du problème de l'exclusion mutuelle classique. Si on choisi h égale à 1, alors le problème de la h-k exclusion mutuelle se réduit au

problème de la k-exclusion mutuelle. Si on choisi h et k, les deux égaux à 1, alors le problème de la h-k-exclusion mutuelle devient le problème de l'exclusion mutuelle.

Plusieurs travaux ont été proposés pour résoudre le problème de la h-k exclusion mutuelle dans les systèmes distribués [JIA01, MBR98, MAN99]. Tous ces travaux utilisent le concept de coteries et de quorums ; les quorums sont des collections d'ensembles de nœuds. Un nœud doit contacter tous les membres de son quorum pour accéder à la ressource partagée. Les algorithmes utilisant le concept de structure de quorums sont généralement tolérants aux pannes et ils ont toujours un coût de communication très réduit. Cependant, le concept de structures de quorums n'est pas directement applicable pour les réseaux mobiles ad hoc puisque la topologie de ces réseaux change continuellement et ainsi, il est très difficile d'une part, de contacter tous les membres d'un quorum et d'autre part, il n'est pas intéressant de garder les mêmes membres d'un quorum.

Certains travaux proposés afin de résoudre les problèmes liés à l'exclusion mutuelle dans les réseaux mobiles ad hoc sont une adaptation de ceux définis dans les réseaux statiques. Citons par exemple, l'algorithme RL (Reverse Link) [WWV98] qui est basé sur l'algorithme à jeton de Raymond [RAM89]. L'algorithme G-RL[JIA01a], l'algorithme k-RL [WCM01], et l'algorithme de J-R Jiang [JIA02] sont trois exemples d'algorithmes qui généralisent l'algorithme RL [WWV98] pour résoudre respectivement le problème de l'exclusion mutuelle de groupe, le problème de la k-exclusion mutuelle, et le problème de la h-k-exclusion mutuelle pour les réseaux mobiles ad hoc.

Notre intérêt dans le cadre de ce modeste travail s'articule sur le problème de l'exclusion mutuelle généralisée du type h parmi k dans les réseaux mobiles ad hoc. Dans ce cadre, nous proposons un protocole orienté jeton qui permet de tenir compte d'un certain nombre de caractéristiques fondamentales des réseaux mobiles ad hoc. Ce protocole se base sur une structure logique de niveaux divisée en deux arborescences logiques ; *Racine* et *Distribution*. Cette structure permet de s'adapter aux changements de la topologie du réseau et essaye de minimiser le nombre de messages échangés pour transmettre le ou les jetons. En outre, le principe de niveaux permet une organisation logique des nœuds en sous-ensembles et cette organisation logique suit systématiquement la topologie physique du réseau. Le protocole, pour son fonctionnement, utilise deux jetons : *Racine* et *Clé* ; chacun appartient à l'une ou l'autre arborescence. Le nœud qui possède le jeton racine collecte les requêtes qui sont mises dans une file d'attente suivant l'ordre de leurs arrivées. Ces requêtes suivent les liens logiques prévus dans l'arborescence racine qui mènent vers le détenteur de ce jeton. Le nœud qui possède le jeton clé détient les ressources disponibles dans le réseau et se charge de les distribuer. Les ressources sont envoyées sur le chemin inverse de l'arborescence clé. Les libérations de ressources sont envoyées au détenteur du jeton clé à travers les chemins définis par l'arborescence.

La mobilité des nœuds a fait partie intégrante de la solution proposée grâce au mécanisme d'inversement de liens en cas d'une défaillance de liens. Ce mécanisme d'inversement de liens est utilisé aussi en cas de déplacement de jeton, car la racine doit être dans le niveau le plus bas, de plus les chemins vers la racine doivent rester décroissants (du plus haut niveau jusqu'à la racine).

Dans le but d'évaluer les performances du protocole conçu et d'étudier son comportement, nous étions amenés à effectuer une simulation à travers la réalisation d'un certain nombre de mesures adéquates à son évaluation. Nous avons définit quelques critères de performances ; à

savoir, le nombre de messages générés par entrée en section critique, le temps d'attente et le délai de synchronisation. En outre nous avons étudié l'influence de la charge, du nombre de ressources dans le système et du taux de connectivité sur les performances du protocole pour chaque critère définit.

Ce document se compose de deux parties. La première concerne l'état de l'art et consiste à étudier le problème de l'exclusion mutuelle simple et généralisé, dans les systèmes distribués et dans les réseaux mobiles ad hoc. Elle se compose de trois chapitres. Le chapitre 1, présente le problème de l'exclusion mutuelle distribuée, expose quelques algorithmes, les classifie et les compare. Il traite aussi, le problème de l'exclusion mutuelle dans les réseaux mobiles ad hoc, et présente les algorithmes utilisés. Le chapitre 2 traite le problème généralisé de l'exclusion mutuelle, à savoir, la k-exclusion mutuelle que se soit dans les réseaux statiques que dans les réseaux mobiles ad hoc ; Il présente quelques travaux antérieurs. Le troisième chapitre étudie le problème de la h-k exclusion mutuelle dans les réseaux statiques et dans les réseaux mobiles ad hoc en présentant les algorithmes proposés dans la littérature.

La deuxième partie intitulée 'Contribution' concerne la solution que nous proposons pour le problème de la h-k exclusion mutuelle dans l'environnement mobile ad hoc. Elle est présentée en deux chapitres :

- Le chapitre 4 présente la conception de notre protocole ; à savoir, sa politique générale, les principes sur lesquels il est basé ainsi que l'ensemble des événements qui le composent en précisant les différentes procédures et variables utilisées avec une description détaillée de son fonctionnement.

- Le chapitre 5 présente une étude de simulation : dans ce chapitre, il est question de présenter l'évaluation des performances de notre protocole dans le contexte prédéfini. Il décrit les différents paramètres pris en compte dans les mesures effectuées, les critères de performances évaluées ainsi que les différents résultats obtenus et leurs interprétations.

Partie 1

ETAT DE L’ART

Chapitre 1

L’exclusion mutuelle

1. Introduction

Dans l'ensemble des ***ressources*** qui composent un système informatique, certaines sont ***critiques***. Autrement dit, ces ressources ne sont accessibles que par un seul processus à la fois. Il peut s'agir de ressources physiques, telles que les mémoires secondaires, et les équipements périphériques. Il peut également s'agir de ressources logiques, comme par exemple des portions de codes (procédures, etc) ou des données (tables, etc).

Lorsque des processeurs (processus) autonomes ne partagent aucune mémoire mais qu'ils coopèrent entre eux par échange de messages via un réseau de communication, ils forment alors un ***système distribué*** (réparti). Ces processeurs ont besoin de partager des ressources critiques. La portion de code décrivant un accès à une ressource critique est appelée ***section critique***. Un processus peut naturellement comporter plusieurs sections critiques ***distinctes***. Lorsqu'un processus est dans une section critique, aucun autre ne pourra se trouver simultanément dans une section critique correspondante pour accéder à la même ressource critique. L'exécution d'une section critique est, donc pour les processus, ***mutuellement exclusive***.

Pour garantir un accès exclusif à une section critique, plusieurs ***algorithmes*** ont été développés se distinguant selon les caractéristiques du système distribué et du système de communication utilisé. Ainsi, tout processus désirant entrer en section critique devra nécessairement exécuter un algorithme d'exclusion mutuelle.

Généralement, les algorithmes qui implantent l'exclusion mutuelle comportent deux parties : un ***protocole d'entrée*** (prologue) et un ***protocole de sortie*** (épilogue). Ces protocoles sont exécutés selon un schéma qui peut être illustré par le code du processus P décrit ci-dessous :

Programme : *Protocole d'accès à une section critique.*

```
Process P ;
    Déclaration ;
    DEBUT
        Instructions ;
        Loop
            Instructions ;
            Protocole d'entrée ;
                SECTION CRITIQUE ; (*Accès à la ressource critique*)
            Protocole de sortie ;
                Instructions ;
        Fin loop
    FIN
```

Les solutions proposées dans la littérature s'appuient sur divers outils et concepts développés afin de construire des algorithmes qui répondent au mieux aux exigences dues à l'exclusion mutuelle et qui soient tolérants aux pannes. Deux approches fondamentales ont été dégagées : dans les algorithmes d'exclusion mutuelle basés sur le jeton, la possession du jeton représente un droit d'accès. Dans les algorithmes basés sur la permission, le nœud demandeur doit obtenir la permission de tous (ou d'un sous-ensemble) les nœuds du réseau.

Dans un environnement mobile ad hoc, les nœuds mobiles sont aussi appelés à partager des ressources dont l'accès doit se faire en exclusion mutuelle. L'absence d'infrastructure fixe complique la conception des algorithmes d'exclusion mutuelle, car, ceux-ci doivent tenir compte des nouvelles exigences imposées par cet environnement, caractérisé par ses fréquentes déconnexions *'Link failure'* et connexions *'Link formation'* ainsi que le changement de la topologie du réseau et partitionnement de celui-ci qui en sont la conséquence. Les caractéristiques physiques des unités mobiles sont aussi des facteurs importants à considérer dans la conception d'un algorithme d'exclusion mutuelle.

Ce chapitre se divise en deux sections, chacune étudie des solutions pour le problème de l'exclusion mutuelle, la première dans les systèmes distribués et la seconde dans les environnements mobiles ad hoc.

1.1 L'exclusion mutuelle simple

L'exclusion mutuelle est un problème fondamental dans les systèmes distribués. Elle consiste à allouer en exclusivité une ressource critique à un et un seul processus à la fois.

Dans cette section, l'intérêt est essentiellement porté sur les algorithmes de la 1-exclusion mutuelle pour leur forte mise en œuvre des concepts de coopération et de compétition dont nous avons besoin le long de cette étude. Après avoir définit les propriétés et les paramètres de performances de ces algorithmes, les différents travaux antérieurs sont présentés. Cette présentation vise les principes de base utilisés pour assurer l'exclusion mutuelle. Ensuite une classification générale sera présentée, dans laquelle, quelques solutions seront étudiées dans le détail en mettant en évidence leurs caractéristiques et les mécanismes et outils adoptés pour la résolution du problème posé. Ces solutions seront accompagnées d'une étude critique qui mettra en valeur les performances obtenues en terme de nombre de messages générés par entrée en section critique ainsi que la taille mémoire utilisée.

1.2.1 Propriétés des algorithmes d'exclusion mutuelle

Un algorithme d'exclusion mutuelle doit posséder un nombre minimal de propriétés (selon E.W.Dijkstra [DIJ74]), lui permettant une mise en œuvre fiable :

- Propriété de ***sûreté*** (*safety*) : à tout instant, un seul processus peut se trouver en section critique ; (P1)
- Propriété de ***vivacité*** (*liveness*) :
  - Si plusieurs processus sont bloqués en attente d'entrer en section critique alors qu'aucun processus ne s'y trouve, l'un d'entre eux doit pouvoir y accéder au bout d'un temps fini ; (P2)
  - Le comportement d'un processus en dehors de la section critique et des protocoles qui gèrent l'accès n'a aucune influence sur l'algorithme d'exclusion mutuelle ;
  - Aucun processus ne joue de rôle privilégié, la solution est la même pour tous.

Il convient d'ajouter, qu'un mécanisme d'exclusion mutuelle doit être tel qu'aucun processus ne doit être mis en attente alors qu'une condition pour laquelle il attend est réalisée. De plus, l'algorithme implémenté doit garantir l'équité. C'est-à-dire un algorithme qui évite des situations de *famine* induites par des processus répétitifs plus prioritaires, qui monopoliseraient indéfiniment la ressource.

1.2.2 Paramètres de performances des algorithmes d'exclusion mutuelle

Les ***paramètres*** d'évaluation d'un algorithme d'exclusion mutuelle sont :

- Le *nombre de messages* par entrée en section critique. Ce nombre regroupe les messages de demande, d'acquisition et de libération de la ressource partagée.
- Le *délai de synchronisation* : C'est le nombre de messages échangés entre deux utilisations successives d'une ressource critique (figure 1). Ce délai est défini pour chaque paire de sites.

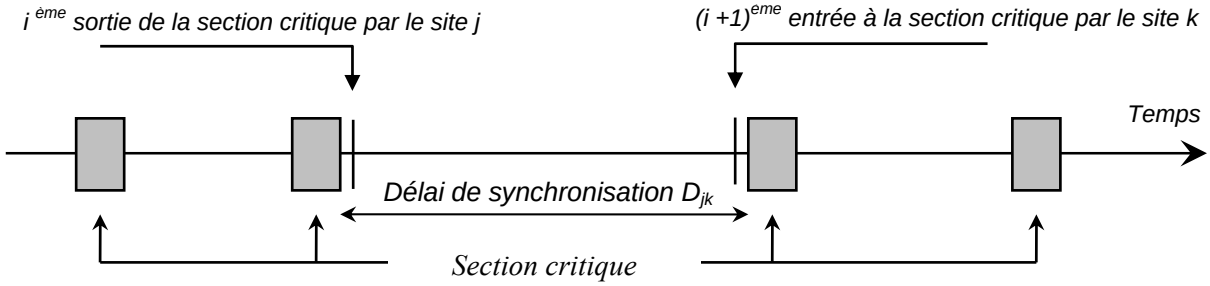


Figure 1 : Délai de synchronisation entre deux sites.

- Le *degré de résistance aux pannes* qui donne une mesure de la vulnérabilité d'un algorithme d'exclusion mutuelle aux pannes d'un site ou de la ligne de communication.
- Le *temps de réponse* : C'est l'intervalle de temps séparant l'envoi d'une requête d'une ressource et l'utilisation réelle de la ressource (figure 2). En général, dans le contexte réseau, le temps est modélisé par le nombre de messages.

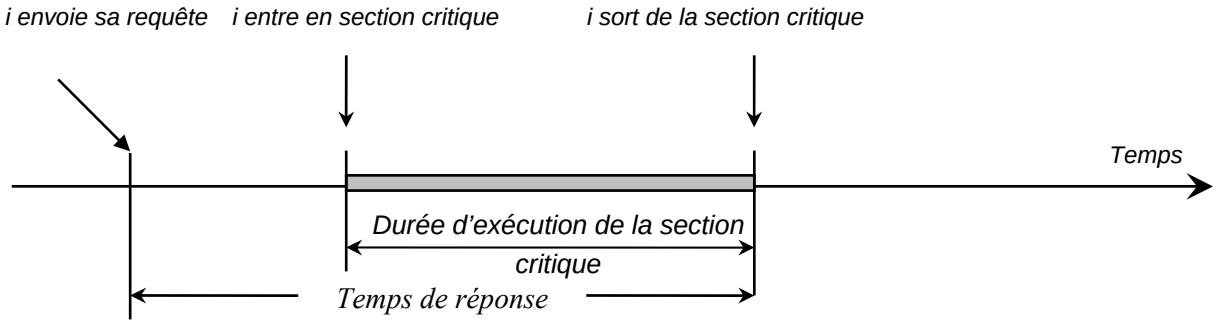


Figure 2 : Temps de réponse d'une requête.

1.2.3 Travaux antérieurs

1.2.3.1 Principes de base

Le moyen le plus simple d'assurer l'exclusion mutuelle est de se servir d'un **serveur** unique (central) qui délivre les autorisations pour entrer dans la section critique. Pour entrer en section critique, un processus envoie un message de requête au serveur et attend une réponse de celui-ci. Conceptuellement, la réponse consiste en un jeton signifiant la permission d'entrer dans la section critique. Si aucun autre processus ne possède le jeton, le serveur répond immédiatement et délivre le jeton. Dans le cas contraire, la requête est placée en file d'attente.

Quand un processus sort de la section critique, il envoie un message au serveur pour l'informer et libère ainsi le jeton. Cet algorithme répond aux exigences de l'exclusion mutuelle mais si un processus émet la requête d'entrer en section critique avant un autre, cette entrée n'est pas garantie dans cet ordre. De plus, au niveau performances, le serveur peut devenir un goulot d'étranglement pour le système tout entier.

Une autre manière très simple pour gérer l'exclusion mutuelle est de placer les processus sur un **anneau logiqu**e. Ceci suppose que chaque processus possède un canal de communication avec le prochain processus sur l'anneau. L'idée est que l'exclusion mutuelle est conférée par l'obtention d'un message passé d'un processus à un autre dans une direction unique. La topologie de l'anneau peut cependant être indépendante de la topologie du réseau sous-jacent. Si un processus ne nécessite pas d'entrer dans la section critique quand il reçoit le message, il le transmet immédiatement à son suivant dans l'anneau. Par contre, un processus désirant l'entrée dans la section critique, va attendre de recevoir le message pour y accéder. Cet algorithme répond aux exigences de l'exclusion mutuelle mais si un processus désire l'entrer dans la section critique avant un autre, cette entrée n'est pas garantie dans cet ordre. Il consomme beaucoup de bande passante car les processus s'envoient des messages continuellement même quand aucun processus ne nécessite d'entrer dans la section critique. Le délai pour un processus voulant entrer dans la section critique varie entre 0 et N envois de messages, mais la sortie de la section critique ne nécessite qu'un seul message. Un exemple de ce type d'algorithme est l'algorithme de Le Lann [LAN77].

Afin de réduire le nombre de messages échangés lors d'une demande d'entrée en section critique par un processus, et d'augmenter la disponibilité du système, une autre classe d'algorithmes est apparue, elle utilise un élégant concept appelé « ***quorum*** » pour résoudre l'accès mutuellement exclusif à la section critique. Ces algorithmes sont appelés généralement, les algorithmes basés-quorum. Ils sont résistants aux pannes de nœud et/ou aux partitionnements des réseaux et ont généralement un coût bas de communication. L'idée de base de ce type d'algorithmes est de "collecter assez de permissions (votes) à partir d'un quorum pour entrer en section critique" [GIF79]. **Garcia-Molina** et Barbara ont proposée la «***coterie***» pour généraliser le concept de quorum [GMB85]. Une coterie est un ensemble d'ensembles avec la propriété que chaque deux membres d'une coterie ont une intersection non-vide. Par la propriété d'intersection, les membres dans une coterie peuvent être utilisés comme arbitres pour garantir l'exclusion mutuelle dans les systèmes distribués.

L'exemple classique d'un algorithme d'exclusion mutuelle basé-quorum est l'algorithme de **Maekawa** [MAE85]. Afin d'entrer en section critique, un nœud doit obtenir la permission d'au moins un quorum de nœuds. Puisque tous deux quorums se croisent et chaque nœud peut seulement accorder sa permission à au plus un nœud à la fois, l'exclusion mutuelle est garantie.

L'algorithme du quorum majoritaire [GIF79, THO79], l'algorithme √n [MAE85], l'algorithme du quorum en arbre [AGA91] et l'algorithme du quorum hiérarchique [KUM91] sont tous des algorithmes basés-quorum. Le coût de communication de l'algorithme basé-quorum est proportionnel à la taille du quorum, donc tous les algorithmes basés-quorum essayent de réduire la taille du quorum tout en gardant un degré élevé de tolérance aux fautes.

Former un quorum dans l'algorithme du quorum majoritaire exige des permissions de plus que la moitié des nœuds. Il est facile de montrer que chaque deux quorums dans l'algorithme du quorum majoritaire ont une intersection non-vide et que la taille d'un quorum est [(N+1)/2]. L'algorithme √n de **Maekawa** utilise le concept de plan projectif fini pour assurer la propriété d'intersection et la propriété de la distribution complète (la charge est la même pour tous les nœuds). Chaque quorum a la même taille et chaque nœud supporte un degré égale de responsabilité pour le contrôle de l'exclusion mutuelle. Comme le titre de l'algorithme insinue, la taille du quorum dans cet algorithme est √n.

Quelques algorithmes utilisent les structures logiques pour aider à former les quorums. Supposons un système organisé en un arbre binaire, l'algorithme quorum en arbre peut avoir la taille [Log_N] dans le meilleur cas pour un quorum. Le quorum formé dans cet algorithme est récursif. Il peut être vu comme essayer d'obtenir des permissions des nœuds le long d'une trajectoire racine-à-feuille. Si la racine manque (tombe en panne), alors, le demandeur de permission doit suivre deux chemins : un chemin racine-à-feuille sur le sous arbre gauche et un chemin racine-à-feuille sur le sous arbre droit. L'arbre quorum le plus grand est de taille [(N+1)/2], c'est celui qui contient tous les nœuds feuilles.

L'algorithme du quorum hiérarchique utilise l'arbre à plusieurs niveaux pour aider à former le quorum. Le concept est simple : un quorum d'un nœud au niveau *i* est formé seulement si assez (plus que la moitié) de quorums de ses nœuds fils au niveau *i*+1 sont formés. Comme montré dans [KUM91], les quorums hiérarchiques peuvent avoir la taille $N^{0,63}$.

Il y a une large classe d'algorithmes qui utilise le concept de passage de **jeton** pour contrôler l'accès à la section critique. L'idée de base de ce type d'algorithme est simple, un nœud doit posséder l'unique jeton (nommé message privilège dans quelques travaux) avant s'entrer en section critique. Donc, dans le meilleur cas, si un nœud possède le jeton, il peut entrer en section critique immédiatement sans aucun frais de communication. Sinon, un mécanisme est exigé pour localiser le jeton. Dans l'algorithme de **Suzuki et Kasami** [SUK85] un nœud envoie (N-1) messages *Request* à tous les autres nœuds et attend jusqu'à la réception du jeton. **Raymond** [RAM89] utilise un arbre recouvrant le réseau pour localiser le jeton et montre que le coût de communication, en utilisant des heuristiques, est O(log N). Le degré de tolérance aux fautes des algorithmes basés - jeton est minime. Si le jeton est perdu, des algorithmes complexes de détection du jeton perdue et de régénération de jeton doivent être exécutés.

1.2.3.2 Classification des algorithmes classiques d'exclusion mutuelle

Une étude plus rigoureuse des algorithmes classiques d'exclusion mutuelle existants a donné naissance à la classification systématisée par le tableau 1. Dans cette classification, La communication entre les sites se fait par échange de messages. Les messages sont délivrés dans leur ordre d'émission et il n'y a ni perte ni duplication, les sites n'ont ni mémoire commune ni horloge globale. Quelques algorithmes sont pris comme exemples pour chaque classe.

Approche	Type		Techniques utilisées	Exemples
Approche basée sur le consensus	Statique (consensus total)		Horloge logique	Lamport 1978 [LAM78]
				Ricart et Agrawala 1981 [RIA81]
	Dynamique (consensus partiel)		Horloge logique	Carvalho et Roucairol 1983 [CAR83]
			HL+ Plans Projectifs Finis	Maekawa 1985 [MAE85]
Approche basée sur le jeton	Basée sur la diffusion (broadcast)	Globale (statique)		Suzuki et Kasami 85 [SUK85]
		Restreinte (dynamique)		Singhal 1989 [SIN89]
	Basée sur une structure logique (unicast)	Statique	Anneau logique	Le Lann 1977 [LAN77]
			arbre	Raymond 1989 [RAM89]
		Dynamique	arborescence	Naïmi et Trehal 1987 [NTM87]
Approche hybride			Singhal + Maekawa	Chang et al 1990 [CSL90a]
			Singhal modifié +Maekawa	Omara et Nabil 2002 [OMN02a]

Tableau 1 : Classification des algorithmes d'exclusion mutuelle

a) Mise en œuvre des algorithmes

Tous les algorithmes que nous verrons ont en commun les conditions opérationnelles suivantes.

1. Le système réparti est formé de *N* sites numérotés de 1 à *N*. Chaque site contient un processus qui de temps en temps désire accéder à une section critique.
2. Les messages émis par un processus sont reçus par les autres processus dans le même ordre qu'ils ont été émis. Il n'y a pas de dépassement des messages provenant du même processus.
3. Chaque processus peut émettre un message directement à tous les autres processus.
4. Chaque message émis est délivré correctement à son destinataire et ceci dans un temps fini.

b) Approche basée sur le consensus

Cette approche suit le principe suivant : un processus désirant entrer en section critique doit demander un ensemble de permissions à d'autres processus et ne peut exécuter sa section critique que s'il a reçu toutes les permissions attendues.

i) Algorithmes statiques (consensus total : permission à/de tout le monde)

Ce type d'algorithmes est dit statique car le nombre de permissions demandées ne change pas d'une exécution à une autre (le site demandeur doit avoir la permission de tous les autres sites). Ces algorithmes utilisent pour la résolution des conflits entre les différentes requêtes, les horloges logiques ou de l'estampille introduite par Lamport 1978 [LAM78]. Cette méthode consiste à appeler une valeur d'horloge à chaque événement (requête). Ce qui permet de créer un ordre entre elles. De ce fait, un processus envoie son autorisation à tous sites demandeurs s'il n'est pas lui aussi demandeur ; sinon, il ne donne son autorisation qu'aux sites dont les requêtes sont les plus anciennes (au sens des estampilles).

Le premier algorithme d'exclusion mutuelle basé-consensus est celui de **Lamport** [LAM78]. Le principe de cet algorithme repose sur la connaissance mutuelle acquise par

chaque site de façon répartie et par la création de files d'attentes dupliquées. Quand un site veut entrer en section critique, il demande l'autorisation à tous les sites et il n'entre en section critique que lorsqu'il sait que sa demande est la plus ancienne. Il a cette information de deux façons, soit parce qu'il a connaissance des dates de demandes d'entrée en section critique des autres sites, soit, pour les sites qui ne demandent pas à entrer en section critique, parce qu'il l'a appris et qu'il sait qu'ils ne feront pas dans le futur de demande qui pourrait être datée avant la sienne. Pour obtenir ce deuxième type d'information, chaque site répond à une demande d'entrée en section critique par un message d'acquittement. Ce message peut être utilisé de façon significative car les canaux sont FIFO et il ne pourra pas être dépassé par une demande d'entrée en section critique. Quand un site quitte la section critique, il averti tous les sites pour que son successeur puisse se reconnaître.

Chaque site établit une veille pour savoir s'il peut entrer en section critique. Il utilise un ensemble de N messages significatifs, un par site du système. En recevant des messages, il met éventuellement à jour cet ensemble et il n'entre en section critique que lorsqu'il reçoive un message de chaque site (il doit avoir connaissance des demandes ou non demandes de tous) et que son message est le plus ancien message de cet ensemble. L'algorithme exige 3(N-1) messages et une file d'attente locale de taille N messages.

En 1981, **Ricart et Agrawala** [RIA81] améliorent l'algorithme de Lamport. Leur algorithme a besoin de 2(N-1) messages pour un nœud pour entrer en section critique. Lorsqu'un nœud veut entrer en section critique, il diffuse un message *Request* à tous les autres nœuds. En recevant un message *Request*, un nœud renvoie un message *Reply* s'il ne veut pas entrer en section critique ; sinon il peut différer le message *Reply*. Des estampilles logiques [LAM78] sont attachées aux messages *Request* des nœuds pour décider s'ils doivent différer ou non leurs messages *Reply*. Seulement le nœud avec la plus petite estampille que celle du message *Request* reçue qui doit différer son message *Reply*. Lorsqu'un nœud reçoit des messages *Reply* de tous les (N-1) nœuds, il peut alors entrer en section critique. Bien que cet algorithme évite l'interblocage et la famine, il est vulnérable aux pannes de nœud et de lien de communication, en plus, il est coûteux en communication (complexité de messages) parce qu'il exige à un nœud de communiquer avec tous les autres nœuds pour entrer en section critique.

ii) Algorithmes dynamiques (consensus partiel : permission demandée à une partie seulement des sites, permission accordée à un seul demandeur**)**

Dans ces algorithmes, un site désirant entrer en section critique envoie sa requête à un **sous-ensemble** de l'ensemble des sites. A un instant donné, un site sollicité par plusieurs requêtes n'accorde son autorisation qu'à l'**une** d'entre elles, les autres étant mises en attente. Une fois que la section critique est exécutée, le site doit rendre les permissions qui lui ont été accordées afin que les autorisations puissent être données aux requêtes qui sont en attente.

Un exemple de ce type d'algorithme est celui de **Carvalho et Roucairol** [CAR83]. Dans cet algorithme, un processus qui désire entrer en section critique, n'envoie un message estampillé qu'aux processus qui ont accédé à la section critique depuis sa dernière demande. Il considère alors qu'il a encore les permissions des autres processus. La structure de la file d'attente locale de chaque site est la même que celle utilisée dans [RIA81]. De plus, la taille de cette file a doublé car il faut indiquer quels sont les sites qui ont envoyé un message de réponse en guise de permission. Le nombre de messages requis est inférieur ou égal à 2(N-1) et une mémoire de tailles 2xN bits est nécessaire pour chaque site.

De même, pour ne pas traiter tous les N sites du système comme un seul ensemble, **Maekawa** [MAE85] a divisé cet ensemble de N sites en √N sous-ensembles mutuellement non disjoints et de taille √N sites. Pour construire ces sous-ensembles, Maekawa fait appel à la méthode dite la structure des plans projectifs finis [MAE85]. Grâce à cette division, chaque site d'un sous-ensemble limite la demande de permission aux √N –1 autres sites qui sont dans le même ensemble que lui. Une fois que le site reçoive la permission de ces autres sites, il exécute sa section critique. Par conséquent, le nombre de messages échangés (NME) de cet algorithme est compris entre 3√N et 5√N. Dans chaque site, la taille mémoire nécessaire à la file d'attente locale pour enregistrer les demandes peut atteindre √N messages.

Agrawal et Abbadi [AGA91] ont utilisé le même principe que Maekawa pour la résolution du problème d'allocation de ressources en exclusion mutuelle. La différence réside dans la manière de construire les groupes (les ensembles, quorums) de nœuds. En effet, dans [AGA91] une structure d'arbre binaire logique est proposée. Un mécanisme de construction d'arbres de quorum générés à partir de l'arbre binaire. Deux quorums distincts ne sont pas disjoints et aucun quorum n'est inclus dans l'autre.

c) Approche basée sur le jeton

Les algorithmes basés sur le jeton comme [LAN77, SUK85, NTA96, SIN89] reposent sur l'unicité de ce jeton. Le site qui détient le jeton exécute sa section critique. Dans cette catégorie, les différents algorithmes se distinguent par la manière dont le jeton circule entre les sites demandeurs. Les inconvénients de cette catégorie d'algorithmes tiennent à la gestion de l'attribution du jeton aux sites demandeurs d'une part et à la détection de la perte du jeton d'autre part. A l'intérieur de cette classe, les algorithmes peuvent être regroupés selon le mode d'envoi des requêtes (i.e : diffusion ou point-à-point).

i) Algorithmes basés sur la diffusion

Dans les algorithmes de la sous-classe diffusion, un site qui veut exécuter sa section critique envoie sa requête en parallèle à tous les autres sites. Cette sous-classe d'algorithmes est divisée aussi en deux sous-classes selon l'évolution du mode de diffusion dans le système qui peut être global ou restreint (dynamique). Les algorithmes à diffusion globale utilisent moins de mémoire car l'état du système n'a pas besoin d'être mémorisé [RIA83, SUK85]. Quant aux algorithmes à diffusion restreinte, ils doivent garder une trace de l'utilisation du jeton pour éviter qu'un site ait à demander le jeton à tous les autres sites. L'avantage de ce groupe d'algorithmes tient à ce que le site qui souhaite exécuter la section critique doit envoyer sa requête uniquement au site qui possède le jeton ou qui a diffusé une demande [CSL91, SIN89].

Dans l'algorithme de **Suzuki et Kazami** [SUK85], un site S_i qui désire exécuter sa section critique envoie sa requête contenant son identificateur *id* et son numéro de séquence *Num* à tous les autres sites. Le numéro de séquence *Num* indique que la demande du site S_i est la $(Num + 1)^{ème}$ demande de la section critique. Chaque site possède un tableau de taille N pour enregistrer les numéros de séquence des autres sites. Le jeton contient un tableau de taille N et une file d'attente qui contient les identificateurs de tous les sites demandeurs de la section critique. Pour chaque site, ce tableau maintient la trace (le numéro de séquence) de la requête accordée le plus récemment. Un site exécute sa section critique quand il reçoit le jeton. Pour ce qui concerne les performances de cet algorithme, comme les

N-1 messages *reply* sont remplacés par un seul message de privilège, la valeur NME est réduite à N (c'est-à-dire N est la somme de (N-1) messages de requête et du message lié au jeton). Cependant, l'amélioration de NME se fait aux dépend de l'introduction de la mémoire pour le jeton qui nécessite un tableau de taille fixée à N. De plus, la taille de la file d'attente du jeton peut augmenter, dans le cas pire, jusqu'à N-1.

Makki [MAK94] a proposé une approche similaire à celle d'une file d'attente du jeton, dans laquelle chaque site, au lieu d'envoyer sa requête à tous les autres sites du système, il l'envoie soit au site possesseur du jeton soit aux sites présumés possesseurs du jeton. Ceci réduit le coût en NME à une valeur inférieure ou égale à N. Afin de permettre aux sites de connaître le possesseur du jeton, celui-ci doit porter plus d'information sur ces sites. Ceci nécessite une mémoire importante pour chaque site et pour le jeton. La quantité d'information supplémentaire dans le jeton varie entre 2N+1 et 4N+1, quant à la taille mémoire de chaque site, elle peut atteindre 2N messages dans le pire des cas.

L'algorithme présenté par **Singhal** [SIN89] emploie une heuristique (dite selon l'état d'information) afin de deviner quel est le site du système qui possède le jeton ou les sites qui sont susceptibles de l'avoir. En effet, un site demandeur doit envoyer sa requête uniquement à de tels sites au lieu de l'envoyer systématiquement à tous les sites du système. Si la charge des requêtes est moyenne ou faible, les résultats expérimentaux indiquent que cet algorithme a de meilleures performances que celui de Suzuki et de Kasami [SUK85] en termes de NME. Dans cet algorithme, chaque site possède deux vecteurs d'état de taille N. Le premier vecteur enregistre les informations relatives aux sites présumés possesseurs du jeton dans un tableau de booléens. Le second vecteur sauvegarde les numéros de séquence dans un tableau d'entiers. Le jeton contient ces deux vecteurs et l'information associée à leurs mises à jour. Au moment de la réception du jeton, cette mise à jour est activée dans les sites en fonction de leurs vecteurs d'état.

L'algorithme de Singhal [SIN89] ainsi que les algorithmes décris dans [SUK85, MAK94] nécessitent une mémoire importante dans chaque site et dans le jeton afin de maintenir la valeur NME qui doit être inférieure ou égale à N. La taille de mémoire dans le jeton consomme de la voie de communication et augmente la probabilité de blocage du jeton. En outre, chaque site s'expose à un overhead (coût d'opération) considérable pour préparer et traiter de longs messages du jeton. Chaque site doit également mettre à jour l'importante information d'état au moment de réception d'un message.

ii) Algorithmes basés sur une structure logique

Dans cette classe d'algorithmes, les sites ont une configuration logique élaborée à partir de variables locales qui caractérisent les liens entre les différents sites. Pour leur représentation, ces liens utilisent des structures fixes (anneau, arbre et autres graphes). Ces liens définissent une structure statique ou dynamique. Dans les algorithmes de la sous-classe statique, la structure logique demeure inchangée et c'est uniquement le sens des arêtes qui change. Si la structure logique est un anneau, le jeton circule sur l'anneau d'un site à un autre, séquentiellement, et le site demandeur de la section critique doit attendre son passage [RAY91, MAR85, LAN77]. Dans le cas des algorithmes statiques utilisant un graphe orienté, le jeton et les requêtes circulent plus librement car il y a plusieurs chemins entre les sites [CSL90, HPR88]. Dans les algorithmes dynamiques, la configuration logique adaptée au réseau change lorsqu'un site demande et/ou exécute sa section critique [NTA96]. Le principal avantage de cette classe d'algorithmes réside dans l'existence de la trace du jeton.

Le premier algorithme basé sur le jeton et une structure logique est celui de **Le Lann** [LAN77]. Dans cet algorithme, les sites d'un système sont organisés sous forme d'un anneau logique dans lequel l'unique jeton circule séquentiellement d'un site à un autre. Lorsque le jeton visite un site, deux cas se présentent : premier cas, le jeton est retenu par ce site pour exécuter sa section critique. Deuxième cas, le jeton est renvoyé immédiatement par ce site à son successeur dans l'anneau. L'activité continuelle du jeton, même s'il n'y a aucune demande en cours, est le seul inconvénient qui se manifeste par une consommation de la largeur de la voie de communication.

Parmi les algorithmes d'exclusion mutuelle basés sur le jeton connus, il en existe quatre qui ont une bonne performance en O(log N) messages par exécution de section critique [RAM89, NEM91, NTM87, CSL90]. L'algorithme de **Naimi et Trehel** [NTM87] a également adopté le concept de file d'attente du jeton comme dans l'algorithme de Singhal [SIN89] et les deux algorithmes [SUK85, MAK94] cités précédemment. Dans ces algorithmes, une file d'attente contient les identificateurs des sites demandant la section critique. Cette file d'attente est attachée au jeton quand il voyage entre les sites. La différence par rapport aux autres algorithmes tient à ce que cet algorithme inclus une structure logique dynamique (un arbre enraciné) représentant un système. La racine est le dernier site demandeur du jeton. Elle le détient en permanence, lorsqu'il n'y a aucune demande dans le système. Quand un site souhaite exécuter sa section critique, il envoie sa requête à son successeur dans l'arborescence (présumé possesseur du jeton) puis la requête est expédiée jusqu'au titulaire du jeton (ou la nouvelle racine). Les inconvénients de cet algorithme sont identiques à [SUK85, SIN89, MAK94], c-à-d, la taille de la file d'attente du jeton n'est pas fixée et elle pourrait augmenter jusqu'à N-1 sites. Le temps de préparation ou de traitement du jeton et celui de la mise à jour de l'information locale dans chaque site ne sont pas négligeables.

Chang, Singhal, et Liu [CSL90] ont surmonté l'inconvénient d'utiliser la file d'attente du jeton en mettant à jour une liste qui joint tous les sites demandeurs (c-à-d, une file d'attente distribuée), telle que chaque site demandeur enregistre seulement l'identificateur de son prochain site demandeur. De ce fait, la réduction et la simplification des données ont été apportées par le jeton.

L'algorithme de **Raymond** [RAM89] est applicable aux systèmes interconnectés par toute topologie. Dans cet algorithme, une structure logique statique est déterminée et maintenue. Le jeton se déplace en réponse à une requête émise par un site souhaitant exécuter sa section critique. Dans le cas contraire, il est stationnaire dans un site. Quand le jeton voyage à travers plusieurs sites (en réponse à une requête du jeton) vers un site demandeur, la direction des arcs traversés est modifiée et le site demandeur devient le nouveau possesseur du jeton après sa réception. Par conséquent, les sens des arcs dans la structure se dirigent toujours vers le présumé possesseur du jeton en faisant de ce dernier un point d'aboutissement dans la structure. Chaque site a une file d'attente locale pour enregistrer les demandes venant des sites voisins. A tout moment, chaque site a une seule demande en attente. La longueur de sa file d'attente locale ne dépasse pas le degré de l'arbre logique du système. Dans l'algorithme de Raymond [RAM89], lorsqu'un site n'utilise plus le jeton, celui-ci est acheminé vers un autre site qui le demande. Cet acheminement est réalisé via les sites intermédiaires entre le demandeur et le possesseur du jeton. Cependant, l'acheminement du jeton dans un réseau augmente le nombre de messages liés à son déplacement. Pour réduire ce nombre, **Neilsen et Mizuno** [NEM91] ont proposé un algorithme basé sur une structure de DAG (Direct Acyclic Graph). Grâce à cette structure, le nombre de messages nécessaires pour une entrée

en section critique est réduit de moitié. Physiquement, les nœuds sont complètement liés par des liens fiables ; logiquement, ils sont organisés en une structure de DAG selon laquelle un seul nœud représente la racine et chaque nœud n'est doté que d'un seul lien sortant. Trois variables sont maintenues au niveau de chaque processus *Last*, *Next* et *Holding*. La variable *Last* représente au niveau de chaque nœud l'identification du dernier nœud lui ayant émis une requête, la variable *Next* indique le prochain nœud à recevoir le privilège, si le nœud en est actuellement le dernier (*Last*=0), sa variable *Next* vaut 0. on peut ainsi retracer la file d'attente depuis le nœud privilégié jusqu'au nœud dont la variable *Next* vaut 0. Grâce à ces variables, les nœuds se passent de la gestion d'une file de requêtes ce qui allège considérablement l'algorithme.

d) Approche hybride

C'est par l'introduction des groupes qu'il a été possible de fédérer les deux approches précédentes pour combiner leurs avantages. Pour minimiser le nombre de messages liés au déplacement du jeton, les sites sont regroupés sur la base de critères définis. A l'intérieur du groupe, sont utilisés des algorithmes basés sur le jeton, et un algorithme basé sur la permission entre les différents groupes, ou vice versa.
L'exclusion mutuelle est gérée à deux niveaux :

- Entre les groupes ;
- A l'intérieur des groupes (entre les sites d'un groupe).

Le choix du couple d'algorithmes *(jeton-permission ou permission-jeton)* réalisant l'exclusion mutuelle à l'intérieur et entre les groupes est basé sur le nombre de messages engendrés [HOT01].

L'algorithme de **Chang et al** [CSL90a] est une combinaison de deux algorithmes d'exclusion mutuelle proposés dans la littérature. Dans [CSL90a], les auteurs les ont appliqués à deux niveaux d'exclusion mutuelle : un niveau local et un niveau global.
-Au niveau local, l'algorithme de Singhal [SIN92] est utilisé. Il utilise les ensembles locaux pour résoudre l'exclusion mutuelle localement.
-Au niveau global, nous retrouvons l'algorithme de Maekawa [MAE85]. C'est lui qui résout les conflits globalement.

Une autre approche hybride a été par la suite développée par **Omara et Nabil** [OMN02]. Dans leur Algorithme hybride, l'algorithme de Lodha et Kashemkayani [LOK00] et l'algorithme de Maekawa [MAE85] ont été combinés et après la modification de l'algorithme de Singhal pour qu'il puisse s'exécuter avec moins de messages que l'algorithme de Singhal d'origine, **Omara et Nabil** ont proposé un nouvel algorithme hybride [OMN02a] en combinant l'algorithme de Singhal modifié et l'algorithme de Maekawa. Ce nouvel algorithme est considéré comme une amalgamation de deux algorithmes, un qui minimise le trafic de messages et l'autre qui minimise le délai de synchronisation. Les mesures de performance prises ont montrées que ce dernier algorithme apporte une amélioration importante par rapport aux deux autres algorithmes.

1.2.3.3 Comparaison entre les différents Algorithmes

Une comparaison des algorithmes basés permission est présentée dans le tableau 2. D'après ce tableau, l'algorithme de Agrawal et d'Abbadi [AGA91] réalise la meilleure performance, en termes de NME si un arbre représentant le système est complet et encastré. En revanche, l'algorithme de Richard et d'Agrawala [RIA81] est très coûteux

en terme de taille mémoire car dans chaque site il faut avoir une mémoire supplémentaire de N bits. En général, chaque site dans cette classe d'algorithmes communique avec tous les autres sites et il enregistre l'information concernant les autres sites. Ceci provoque une circulation dense de messages en utilisant une taille mémoire très importante. La valeur de NME des algorithmes basés sur la permission varie entre O(log N) et 3x(N-1).

Année	Auteurs	Nombre de messages échangés (NME)	Taille de la mémoire dans chaque site
1978	Lamport	3x(N-1)	≤ N messages
1981	Ricart et Agrawala	2x(N-1)	N bits
1983	Carvallo et Roucairol	≤ 2x(N-1)	2xN bits
1985	Maekawa	(3√N, 5√N)	≤ √N messages
1991	Agrawal et Abbadi	(O(log N), ((N+1)/2))	≤ log N ou ≤(N+1) messages
1992	Singhal	≤ 2x(N-1)	≤ N mots

Tableau 2 : Comparaison des algorithmes basés sur la permission

Une comparaison de la classe des algorithmes basés jeton est présentée dans le tableau 3, où la valeur moyenne de NME dans [RAM89, NEM91, NTM87, CSL90] est obtenue en considérant seulement l'intégration d'une structure logique. Notons que le message associé à une requête dans [SUK85, SIN89, NEM91] contient 4 mots (trois mots plus le nombre de séquence ou l'identificateur). A partir du tableau 3, Nous observons que les algorithmes [SUK85, SIN89, MAK94] réduisent la NME à une valeur qui est inférieure ou égale à N au prix de l'incorporation d'une mémoire supplémentaire (au moins N mots) dans chaque site et dans le jeton.

Dans l'algorithme [LAN77], le jeton est actif et voyage constamment dans l'anneau même si aucun site ne le demande. Ceci consomme inutilement de la voie communication du réseau. Par conséquent, il est difficile d'estimer cette dépense en termes de NME, mais la valeur minimale de NME est d'un message. Les quatre algorithmes [RAM89, NEM91, NTM87, CSL90] basés sur le jeton réalisent une meilleure performance en termes de NME. Cette valeur est de O(log N) messages par exécution d'une section critique. Quand la mémoire supplémentaire est prise en compte, l'algorithme de Naimi et Trehel [NTM87] est clairement moins performant que les autres algorithmes de sa classe parce qu'il nécessite (N-1) mots de mémoire pour chaque site et pour le jeton. Les trois autres algorithmes peuvent se dispenser de cette mémoire supplémentaire. Nous ne pouvons pas dire qu'un des trois algorithmes [RAM89, NEM91, CSL90] réalise une meilleure performance, cependant une étude de simulation a été effectuée [FUS97] pour comparer les comportements de ces trois algorithmes. Les résultats de cette simulation indiquent que l'algorithme de **Chang et al** [CSL90] est le meilleur (en termes de NME) quand le taux des demandeurs de la section critique est faible, mais il reste moins performant que l'algorithme de Raymond [RAM89] quand le taux des demandeurs s'élève.

Année	Auteur(s)	NME	Mémoire dans chaque site	Mémoire réservée pour le message jeton	Information totale échangée (en mots)
1978	Le lann	(0, N)	aucune	aucune	/
1985	Suzuki et Kasami	N	N mots	N+(1,N-1) mots	4N+(N+1,2N-1)
1987	Naimi et Trehel	O(log N)	≤ (N-1) mots	≤ (N-1) mots	O(log N)+(1,N)
1989	Raymond	O(log N)	≤ degré de l'arbre logique	aucune	O(log N)
1989	Singhal	O(log N)	N mots + N bits	N mots + N bits	4N+N mots + N bits
1990	Chang, Singhal et Liu	O(log N)	C mots (C est une constante)	1 mot	O(log N)
1991	Neilsen et Mizuno	O(log N)	C mots (C est une constante)	aucune	O(log N)
1994	Makki	≤ N	< 2 messages	2N+(1,2N+) mots	3N+(2N+2,4N+2)

Tableau 3 : Comparaison des algorithmes basés sur le jeton.

1.3 L'exclusion mutuelle dans les réseaux mobile ad hoc

Dans un réseau mobile ad hoc, une paire de processus communique par transmission de message à travers un lien sans fil direct, ou à travers une séquence de liens sans fil en incluant un ou plusieurs processus intermédiaires pour acheminer le message. La communication directe est possible seulement entre paires de processus qui se trouvent dans le rayon de transmission l'un de l'autre. Les *défaillances* de liens sans fil se produisent quand les nœuds communiquant précédemment se déplacent tel qu'ils ne soient plus dans la portée de transmission de l'un l'autre. Egalement, les *formations* de liens de communications se produisent quand les nœuds qui étaient loin, se déplacent tel qu'ils soient dans la portée de transmission l'un de l'autre. Les caractéristiques qui peuvent distinguer les réseaux sans fil ad hoc aux réseaux distribués existants incluent les changements fréquents et imprévisibles de la topologie et l'échange de messages qui se fait dans un environnement où les liens de communication sont non fiables. Ces caractéristiques font que les réseaux mobiles ad hoc changent les environnements d'implémentation des algorithmes distribués.

Les algorithmes de routages dans les réseaux mobiles ad hoc, fournissent la capacité d'envoyer des messages de n'importe quel nœud à tout autre nœud. Les algorithmes distribués existant s'exécutent correctement au-dessus des protocoles de routage ad hoc, puisque ces protocoles sont désignés pour cacher la nature dynamique de la topologie du réseau des couches élevées dans la pile des protocoles.

Après avoir conçu leur premier algorithme qui s'exécute au-dessus des protocoles de routage ad hoc, dans [WAK97] ; **Walter et al** ont soulevé, dans [WWV98], la question de ce qui peut être plus efficace aux algorithmes distribués, être informé de la mobilité dans le réseau ou être protégé de cette mobilité par les protocoles de routages ad hoc. Ils ont présenté un algorithme d'exclusion mutuelle pour les réseaux mobiles ad hoc et ils ont évalué ses performances par simulation par rapport à un algorithme d'exclusion mutuelle statique [RAM89] s'exécutant au-dessus d'un protocole de routage ad hoc qui trouve toujours les plus courts chemins. Il s'avère que leur solution est plus appropriée pour cet environnement.

1.3.1 Travaux antérieurs

A travers cette section, nous allons présenter les algorithmes d'exclusion mutuelle qui existent dans la littérature conçus pour les environnements mobiles ad hoc.

1.3.1.1 L'algorithme de Baldoni et al

La solution de **Baldoni et Al** [BVP02] est basée sur le principe d'utilisation d'un jeton circulant qui symbolise un certain privilège, en l'occurrence, le droit d'accès en section critique. Deux principes de base ont été combinés afin de répondre efficacement aux requêtes des processus : "*demande de jeton*" et "*circulation de jeton*". L'ensemble des processus du système est structuré suivant un anneau logique, construit dynamiquement.

La technique de demande de jeton « Token-Asking » permet à un processus, désirant accéder en section critique, d'envoyer une requête au coordinateur. L'envoi est omis si ce nœud est lui-même coordinateur. A la réception de cette requête, le processus P qui détient le jeton doit l'insérer dans la file des requêtes. En sortant de sa section critique, le processus P doit envoyer le jeton accompagné de la file des requêtes au premier processus dans la file. Dans le cas où la file est vide, le processus P se met en attente d'une requête tout en gardant le jeton (Idle state).

Le principe de l'algorithme consiste à organiser l'ensemble des processus suivant un anneau virtuel qui est construit d'une manière dynamique : pendant un tour, un processus ayant reçu le jeton choisit, parmi les processus ne l'ayant pas encore reçu, le prochain vers lequel le jeton sera transmis, et cela suivant un critère déterminé (Ex : la distance minimale). A la fin du tour, le jeton retourne vers le coordinateur qui est le seul processus pouvant le bloquer pour, ainsi, basculer vers l'état de repos (Idle state). Le jeton circule suivant l'anneau logique, et donne, ainsi, à chaque processus le recevant, la possibilité d'accéder en section critique. Dans un meilleur cas, l'accès en section critique peut s'achever avec un nombre minimal de messages échangés si chaque processus formule une requête pendant un seul tour du jeton (chaque processus est satisfait suite à le formulation d'une seule demande du jeton).

L'algorithme présenté ne définit pas d'hypothèse sur les délais de transfert des messages, ni sur la vitesse d'exécution des processus, ce qui rend le système asynchrone. Toutefois, le réseau peut faire l'objet de partitionnements temporaires sans pour autant nuire au fonctionnement de l'algorithme : tout nœud sera en mesure de communiquer avec tout autre nœud du réseau.

Le fonctionnement de l'algorithme pourrait être vu comme une succession de transitions entre deux états : l'état de repos « Idle state » et l'état de changement de coordinateur « Coordinator-Change » (figure 3). La circulation du jeton est réalisée à travers des tours successifs munis chacun d'un coordinateur et durant lesquels s'alternent les deux états. Un nouveau tour commence lorsqu'il y a transition vers l'état « Coordinator-Change ».

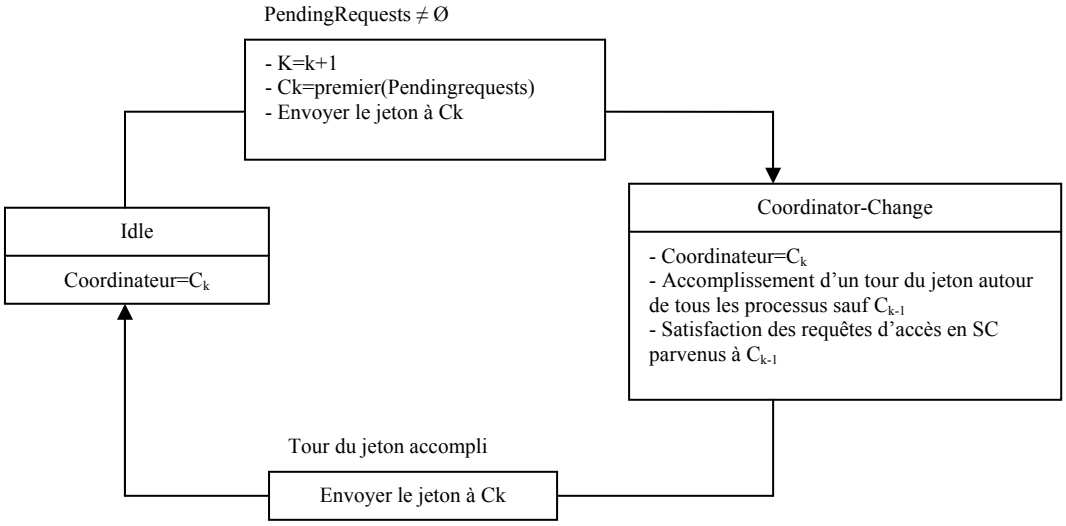


Figure 3 : Diagramme d'état de l'algorithme de Baldoni.

Un tour k est donc identifié par un coordinateur C_k. Le changement de coordinateur de C_{k-1} est provoqué par une transition de l'état « Idle » vers l'état «Coordinator-Change », et le processus provoquant la transition devient le coordinateur courant. Ce dernier est le seul processus pouvant exécuter la prochaine transition de l'état «Coordinator-Change » vers l'état « Idle ».

a- Le fonctionnement de l'algorithme

Tout processus désirant accéder en section critique envoie une requête au coordinateur qui l'insère dans la file «PendingRequest » ordonnée selon un critère donné (par exemple: FIFO).

Initialement, le système est en état de repos, le coordinateur C_0 correspondant au processus P_1 est choisi comme coordinateur courant et est connu de tous les autres processus du système. L'ensemble « PendingRequests » des requêtes pendantes est donc vide. La transition vers l'état « Coordinator-Change » a lieu lorsque le coordinateur courant est en état de repos, et que l'ensemble « PendingRequests » n'est plus vide. Durant cette transition, le coordinateur envoie le jeton au premier processus de l'ensemble « PendingRequests », soit P_j, qui va être le prochain coordinateur C_{k+1}.

L'état « Coordinator-Change » correspond à un seul tour du jeton et permet de le faire circuler suivant un anneau logique qui commence à partir du coordinateur courant et qui est composé de N-1 processus (le processus correspondant au coordinateur C_{k-1} précédent est exclu du tour). Ce tour est organisé afin, d'une part, d'informer l'ensemble des processus de l'identité du coordinateur courant, et d'autre part, de satisfaire les demandes pendantes parvenues au coordinateur précédent C_{k-1}.

Concernant la transition de l'état « Coordinator-Change » vers l'état « Idle », elle ne peut être réalisée que par le processus coordinateur, et cela après qu'il ait reçu de nouveau le jeton.

b- La construction de l'anneau

Comme nous l'avons déjà vu, l'anneau logique parcouru pendant le tour du jeton est construit dynamiquement, il est composé de N-1 processus et ne comporte pas le processus coordinateur précédent C_{k-1}. Ce faisant, durant le tour, le jeton est à chaque fois transmis à un processus de l'anneau, le dernier le retransmet au coordinateur courant C_k qui, ainsi, provoque la transition vers l'état « Idle ».

Selon ce principe, l'anneau logique devrait être différent d'un tour à un autre, et afin de parvenir à sa construction, une structure de données dite «ReceivedToken », représentée par un tableau de booléens de taille égale au nombre de processus (dans notre cas : N processus), est utilisée. Lorsqu'un processus *i* reçoit le jeton, son entrée ReceivedToken[*i*] est mise à TRUE, et cela afin qu'il ne soit pas revisité durant un même tour. Par ailleurs, pendant chaque transition vers l'état « Coordinator-Change », le coordinateur C_{k-1} initialise toutes les entrées du tableau ReceivedToken à FALSE à l'exception de l'entrée qui lui correspond avant d'envoyer le jeton au nouveau coordinateur C_k. Cela permet d'indiquer qu'aucun processus n'a encore reçu le jeton. Le tableau ReceivedToken accompagnera le jeton pendant son tour.
Lorsqu'un processus *i* reçoit le jeton (accompagné du tableau ReceivedToken) pendant un tour dirigé par le coordinateur C_k, il met d'abord l'entrée ReceivedToken[*i*] à TRUE, et définit, par la suite, l'ensemble des successeurs possibles $PossSucc_i$ de la manière suivante :
$PossSucc = \{P_j/ReceivedToken[j]=FALSE \wedge (j \neq i)\}$

Après avoir défini l'ensemble $PossSucc_i$, le processus *i* envoie le jeton (toujours accompagné de ReceivedToken) au premier processus de l'ensemble $PossSucc_i$ ordonné sur la base du nombre de sauts séparant ses éléments du processus *i*. Lorsque deux processus ont une même distance les séparant du processus *i*, celui ayant le plus petit identificateur passera en priorité. Si l'ensemble $PossSucc_i$ est vide, le jeton est transmis au processus coordinateur C_k qui marque, ainsi, la fin du tour en passant à l'état de repos.
Dans la figure 4, un exemple de construction d'anneau est illustré à travers un système composé de 9 processus.

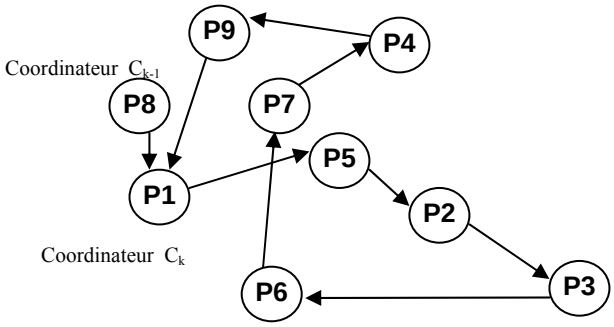


Figure 4 : Un exemple de l'évolution de l'anneau logique.

Dans cet exemple, le coordinateur précédent est le processus 8, tandis que le processus 1 représente le coordinateur courant. L'anneau est donc construit au fur et à mesure que le jeton avance : chaque nœud ayant reçu le jeton construit l'ensemble $PossSucc_i$ selon lequel il choisit la prochaine destination du jeton. Finalement, lorsque le processus 9 reçoit le jeton, il se rend compte, après qu'il ait calculé son ensemble de successeurs possibles $PossSucc_i$, qu'il est le dernier visité par le jeton (dernier processus de l'anneau : $PossSucc_i=Ø$). Ce dernier est donc retransmis par le processus 9 au coordinateur courant C_k qui met fin au tour du jeton en basculant vers l'état « Idle ».

c- Avantages de l'algorithme

1- L'anneau est construit dynamiquement par l'envoi du jeton au processus qui demande le moindre effort en termes de ressources du réseau et de l'énergie.
2- La politique P présente une bonne heuristique, qui tient compte de la mobilité des nœuds avec la supposition que l'algorithme utilise les informations fournies par le protocole de routage.
3- La politique P peut être substitué sans qu'elle ait un impact sur le reste de l'algorithme.

d- Inconvénients de l'algorithme

1- Pour une seule requête, le jeton doit traverser tout l'anneau logique, ce qui représente une charge supplémentaire en terme de communication et consommation d'énergie ;
2- L'algorithme définit un ordre total sur tous les processus du système (chaque nœud connaît les informations de routage de tous les nœuds du réseau), ce qui implique que le protocole de routage exécuté par les nœuds est proactif ;
3- Des nœuds qui n'ont pas demandé l'accès en section critique participent à l'exécution de l'algorithme ;
4- L'envoi du jeton est accompagné d'un vecteur de type booléen ce qui augmente la taille du message de jeton ;
5- L'évaluation des performances de l'algorithme ne tient pas compte des messages de routage ;
6- Le calcul des successeurs à chaque réception de jeton peut être coûteux si le nombre de nœuds est grand ;
7- Certains messages envoyés sont inutiles ; par exemple : l'envoi d'un message de requête au coordinateur alors que le jeton est en cours de circulation ; sachant en plus que ces messages seront ignorés ;
8- L'utilisation de la file "PendingResquests" ne sert qu'à sélectionner le prochain coordinateur : il n'y a pas de gestion des requêtes au niveau de celui-ci.

1.3.1.2 L'algorithme RL de Walter et al

L'algorithme RL (Reverse Link) [WWV98] sera présenté de manière détaillée, car il présente la base d'autres travaux pour la k et la h-k exclusion mutuelle dans les réseaux mobiles ad hoc qui feront l'objet de l'étude dans les chapitres qui suivent.

L'algorithme des liens inversés (RL) [WWV98] est basé sur l'utilisation d'un jeton circulant. Il combine les deux principales caractéristiques des protocoles de routage basés sur la construction des arbres orientés [GAB81], et des algorithmes d'exclusion mutuelle basés sur la circulation du jeton [CSL90]. Ce protocole convient parfaitement à la catégorie des réseaux mobiles caractérisés notamment, par le nomadisme des nœuds composant le réseau, et cela, vu que chaque nœud se contente de la connaissance des informations concernant ses voisins directs.

L'algorithme est fondé sur la mise en place d'un graphe particulier appelé « DAG » (Directed Acyclic Graph) formé sur la base de la topologie du réseau. Tous les liens physiques (i.e liens sans fil) sont orientés de telle sorte que chaque nœud ait un chemin vers le nœud détenteur du jeton. Ce faisant, chaque nœud peut disposer de plusieurs chemins menant vers le nœud détenteur du jeton afin de garantir la validité de certains chemins en cas de défaillance des liens ou suite à un changement de la topologie du réseau. L'algorithme pose les contraintes suivantes sur le réseau sous-jacent :

R_1-Les défaillances des liens sont liées uniquement au mouvement des nœuds, les liens sont supposés bidirectionnels et dépourvus d'erreurs de transmission.
R_2- Chaque nœud a un champ de communication uniforme et ne peut communiquer directement qu'avec des nœuds se situant dans ce même champ.
R_3- Chaque nœud connaît tous ses voisins directs.
R_4- Les défaillances des liens sont détectées par le protocole de la couche de liaison à travers un signal particulier appelé « signal seuil ». Si un tel signal est reçu, le protocole de la couche liaison interrompe la communication au niveau du nœud l'ayant reçu.

L'algorithme pose aussi les hypothèses suivantes sur les sites :
S_1- Chaque nœud a un seul identificateur ;
S_2- La vitesse des nœuds et l'espace de mouvement sont limités, et aucun nœud ne peut se déconnecter du réseau pendant l'exécution de l'algorithme ;
S_3- La vitesse de transmission des messages est plus grande que la vitesse de défaillance des liens, ce qui garantie l'inexistence de messages en transit (dans les canaux de communication).
S_4- Le nombre de nœuds N dans le système est connu par chaque nœud du réseau.
S_5-Chaque nœud P_i possède, au plus, une requête d'accès en section critique non satisfaite.

a- Principe de fonctionnement de l'algorithme RL

Initialement, un nœud est désigné pour détenir un jeton et initier la construction du « DAG ». Les pointeurs logiques correspondent à des liens physiques sans fil dirigés directement vers le nœud détenteur du jeton dit « nœud racine », et leur construction est selon l'ordre lexicographique des nœuds par rapport à leurs voisins immédiats. Un chemin est également établi au niveau de chaque nœud en direction du nœud racine. L'ensemble de ces chemins forme un arbre orienté vers le nœud racine.

Les demandes du jeton sont adressées au nœud racine qui détient une file des requêtes (de la même manière que les autres nœuds) dans laquelle sont rangées (et triées selon l'ordre d'arrivée) les requêtes provenant des nœuds voisins (et naturellement de lui-même). Les éléments de cette file représentent le chemin de retour qui sera emprunté par le jeton jusqu'à l'aboutissement au nœud demandeur. De cette manière, un nœud i ayant exprimé une demande d'accès en section critique et dont l'identité est au sommet de sa file des requêtes pourra accéder en section critique aussitôt qu'il reçoit le jeton.

La réception du jeton signifie, pour le nœud récepteur, la modification de son poids par rapport à ses voisins de telle façon qu'il devienne le moins élevé (le nœud racine doit avoir le poids le plus bas au sein du DAG). Le poids d'un nœud correspond au nombre de nœuds le séparant du nœud racine (l'augmentation du poids d'un nœud est proportionnelle à la distance qui le sépare du nœud racine).

Les requêtes de jeton parviennent naturellement au nœud racine d'après l'hypothèse R_2. Cependant, le chemin de retour risque de devenir défaillant avant que le jeton n'atteigne le nœud demandeur suivant, par exemple, à une longue détention du jeton par le nœud racine. Dans ce cas, et afin de faire parvenir le jeton au nœud demandeur, une recherche de chemin est effectuée par le nœud possédant instantanément le jeton à travers l'envoi de messages de recherche via tous ses canaux de sortie (liens). A la réception du message de recherche du nœud demandeur, chaque nœud en transmet une copie sur ses liens entrants jusqu'à atteindre le nœud recherché et auquel le jeton sera transmis.

Par ailleurs, des créations et des ruptures de liens peuvent avoir lieu durant l'exécution de l'algorithme. Quand cela arrive, le DAG est reconstitué sur la base de la nouvelle topologie et suivant la méthode décrite auparavant [GAB81].

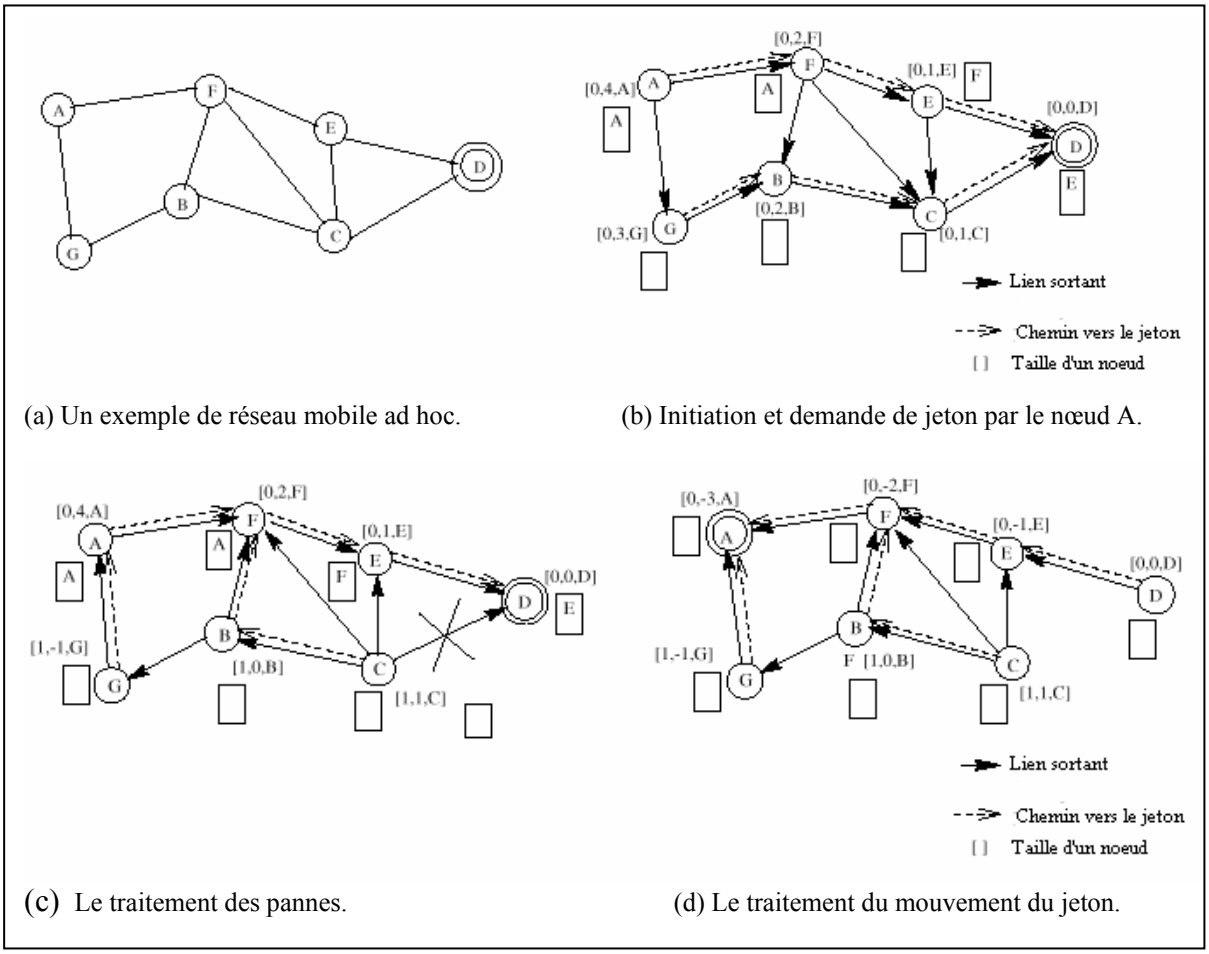
b- Structure de données utilisées dans l'algorithme RL

Chaque nœud *i* maintient à son niveau les informations suivantes :

état : Indique l'état où se trouve le processus d'application, *WAITING*, *CRITICAL*, *REMAINDER*.
$\boldsymbol{N_i}$: L'ensemble des voisins d'un nœud *i*.
$\boldsymbol{myheight_i}$: Une taille d'un nœud i :triplet (α_i, β_i, i).
Height[j] : Un tableau qui contient les tailles des voisins.
$\boldsymbol{tokenHolder_i}$: Un booléen qui vaut vrai si le nœud *i* possède le jeton et faux dans le cas contraire. Initialement $tokenHolder_i$ =vrai pour i=0 et faux pour $i \neq 0$.
$\boldsymbol{Next_i}$: désigne le nœud voisin sur le chemin qui mène au nœud privilégié.
$\boldsymbol{Q_i}$: file qui contient l'identificateur des nœuds voisins et de *i* qui ont demandé d'accéder en section critique.

c- Scénario de fonctionnement de l'algorithme RL

La figure 5 illustre un exemple d'exécution de l'algorithme RL où sont représentés le mouvement du jeton et les défaillances des liens qui peuvent se produire pendant l'exécution.



(a) Un exemple de réseau mobile ad hoc.

(b) Initiation et demande de jeton par le nœud A.

(c) Le traitement des pannes.

(d) Le traitement du mouvement du jeton.

Figure 5 : Scénario de fonctionnement de l'algorithme RL.

La figure 5(a) montre un simple réseau mobile ad hoc formé de sept sites interconnectés par des liens sans fil. La figure 5(b) montre l'état du réseau après l'initiation de l'algorithme. Ici le nœud D possède le jeton, les autres nœuds construisent leurs pointeurs logiques représentés par des flèches pointant vers le nœud D (le nœud racine), ainsi, le « DAG » est formé. Chaque nœud maintient toujours un chemin pointant vers le jeton représenté vers D, où tous les liens sont entrants. Les tailles des nœuds sont mises de telle sorte que tous les pointeurs logiques pointent vers le nœud D. Ce dernier met sa taille à (0,0,D), les autre nœuds C,E,F,B , A, et G mettent respectivement leurs tailles à (0,1,C), (0,1,E), (0,2,F), (0,2,B), (0,4,A), (0,3,G).

Dans cette partie, le nœud A exprime une demande de jeton à travers son chemin déjà construit vers D. Il commence par sauvegarder son identité dans sa file et envoyer sa requête au nœud F. Après l'enfilement de l'identité de A dans sa file, le nœud F envoie la requête, à son tour, au nœud E qui, également, enfile l'identité de F dans sa file et envoie la requête au nœud D. Finalement, le nœud D termine par enfiler le nœud E dans sa file des requêtes.

La figures 5(c) montre l'occurrence de défaillance de lien et comment réagit l'algorithme dans ce cas. Ici, le lien C-D est en panne à cause de l'augmentation de la distance entre les deux sites concernés. Suite à cette défaillance, le nœud C perd tous ses liens sortants et ne possède pas le jeton ; donc, il met sa taille à (1,1,C) pour au moins garantir l'existence d'un lien sortant. Cette mise à jour crée un nouveau lien logique vers B mais engendre un problème

qui est la perte de tous les liens sortants de B. Comme pour C, la mise à jour de la taille de B engendre le même problème pour le nœud G (la perte des liens sortants), et ainsi de suite. Les nouvelles tailles de B et G sont respectivement (1,0,B) et (1,-1,G). De cette façon, le DAG est reconstitué.

La figure 5(d) montre la circulation du jeton afin de satisfaire le nœud demandeur du jeton, à savoir, le nœud A. Le nœud D envoie le jeton au premier nœud figurant dans la file des requêtes (le nœud E) en inversant son lien et en changeant son chemin en même temps. Le nœud E met sa taille à (0,-1,E) et envoie le jeton au nœud F en inversant le lien vers F. A son tour, le nœud F met sa taille à (0,-2,F) et envoie le jeton au nœud A en procédant de la même manière. A la réception du jeton, le nœud A met sa taille à (0,-3,A) et accède à sa section critique. Finalement, tous les nœuds construisent leurs pointeurs logiques vers A et définissent les différents chemins menant vers A.

d- Avantages de l'algorithme RL

1- La structure logique de la topologie du réseau correspond aux liens physiques en traitant les cas de destruction et formation de liens.
2- L'algorithme d'exclusion mutuelle ne dépend pas d'un protocole de routage spécifique.
3- Les nœuds ne contiennent des informations que sur leurs voisins directs.

e- Inconvénients de l'algorithme RL

1- Quelques hypothèses ne sont pas réalistes comme le partitionnement du graphe ne peut pas se produire.
2- Un nœud qui a envoyé une requête et perd son lien vers son voisin *Next*, renvoie la requête vers un autre voisin, ce qui augmente le temps d'attente pour un nœud pour avoir le jeton et peut produire une situation de famine si les changements de la topologie ne s'arrêtent pas.
3- Des nœuds qui n'ont pas demandé l'accès en section critique participent à l'exécution de l'algorithme.
4- Un nœud privilégié qui n'est pas en tête de sa propre file envoie le jeton pour le demander une nouvelle fois.

1.4 Conclusion

Le problème de l'exclusion mutuelle dans les systèmes distribués a été largement étudié dans la littérature, et plusieurs algorithmes ont été développés pour le résoudre. Ces algorithmes peuvent être regroupés en deux grandes familles selon l'approche qu'ils utilisent pour garantir l'exclusion mutuelle. Nous distinguons, les algorithmes basés- jeton et les algorithmes basés- consensus. Nous trouvons également la « famille » des algorithmes combinant les deux techniques précédentes (jeton et consensus). À l'intérieur de ces familles, les algorithmes peuvent être regroupés selon le mode d'envoi des requêtes (point à point ou diffusion), ou selon le graphe de communication (anneau, arbre..). Nous avons discuté les algorithmes les plus connus ; par exemple, l'algorithme de Lamport, de Ricart–Agrawala, l'algorithme de Singhal…etc, qui sont basés- consensus, l'algorithme de Suzuki-Kasami, de Raymond... etc, qui sont basés- jeton, et enfin, l'algorithme de Chang et al, et l'algorithme de Omara et Nabile qui sont des algorithmes hybrides. Cette présentation et classification des algorithmes de la 1-exclusion mutuelle dans les réseaux statiques existants nous ont d'abord

permis de les recenser et de les comparer sur la base des critères et des paramètres définis précédemment.

Nous avons étudié aussi dans cette partie, les premières solutions au problème de l'exclusion mutuelle pour l'environnement mobile ad hoc. Nous avons vu l'algorithme de Baldoni et al [BVP02] qui est basé sur la technique du jeton circulant suivant un anneau logique construit dynamiquement (selon l'évolution de la topologie du réseau), ce qui assure sa convenance aux réseaux mobiles ad hoc. Par la suite, nous avons traité l'algorithme RL de Walter et al [WWV98] qui est basé sur le principe d'un graphe acyclique connu par le DAG et qui est mis à jour, continuellement, en fonction du mouvement du jeton.

L'étude de ces algorithmes nous a permis de découvrir la difficulté de conception des algorithmes d'exclusion mutuelle dans les réseaux mobiles ad hoc, et la nécessité de tenir compte, lors de leur conception, de la mobilité des nœuds, des changements fréquents de la topologie, et des caractéristiques physiques, notamment, la limitation de la source d'énergie.

Chapitre 2

La k-exclusion mutuelle

2.1 Introduction

Le problème de l'exclusion mutuelle distribuée a été largement étudié et beaucoup de protocoles intéressants pour sa solution ont été proposés. La plupart de ces algorithmes essaient de fournir la plus haute performance en réduisant le nombre de messages impliqués et en améliorant les chances d'accomplissement de l'exclusion mutuelle en présence des échecs de sites ou de liens de communication. Une classe plus générale des algorithmes d'exclusion mutuelle distribuée permet à k nœuds d'entrer simultanément à leurs sections critiques [FYA91, KFY93, RAM89a, SRR92] d'où le problème de la k-exclusion mutuelle. Supposons qu'il y a k ressources identiques qui sont partagées par les nœuds dans un système distribué. Chaque ressource peut seulement être accédée par un nœud à la fois, et chaque nœud peut accéder a au plus une ressource à la fois.

La façon la plus simple de résoudre le problème de la k-exclusion mutuelle est d'utiliser l'approche qui essaye simplement de réduire le problème de l'exclusion mutuelle multiple au problème de l'exclusion mutuelle simple. Par exemple, un algorithme d'exclusion mutuelle indépendant peut être utilisé pour le contrôle de l'accès à chaque ressource. Un nœud doit choisir une ressource particulière et utilise l'algorithme pour y accéder. Cependant en utilisant cette approche, un nœud peut être bloqué longtemps en attendant une ressource qui a été demandée par plusieurs autres nœuds, alors qu'il y a d'autres ressources qui sont immédiatement disponibles. Alternativement, un algorithme d'exclusion mutuelle peut être modifié pour permettre à chaque nœud d'octroyer sa permission à au plus k nœuds à la fois. Cependant cette approche peut violer la propriété de sûreté, car plus que k nœuds peuvent être permis d'accéder à la ressource. Tous ces problèmes ont motivé l'introduction d'algorithmes propres à la k-exclusion mutuelle. De tels algorithmes peuvent être utilisés pour coordonner le partage d'une ressource qui ne peut être allouée qu'au plus k sites à la fois.

Les algorithmes de la k-exclusion mutuelle distribuée sont généralement groupés selon la méthode par laquelle ils allouent l'accès à la section critique. Dans les algorithmes basés-permission, un processus demandant l'accès à la section critique doit la demander et doit avoir la permission accordée explicitement de tous ou d'un sous-ensemble de processus dans le système. Dans les algorithmes basés jeton, la permission d'un jeton unique ou de plusieurs jetons permet l'accès à la section critique. Les algorithmes basés jeton sont un meilleur choix pour les réseaux mobiles ad hoc dynamiques parce qu'ils exigent moins de communications directes inter processus, c'est une considération importante quand l'état du lien est constamment incertain. Les algorithmes basés permission sont difficiles à mettre en œuvre vu l'énergie pauvre des environnements sans fil puisqu'ils exigent la communication entre chaque nœud dans le réseau pour tout accès à la section critique.

2.2 La k exclusion mutuelle dans les systèmes distribués

Plusieurs algorithmes de la k-exclusion mutuelle ont été proposés [RAM89a, FYA91, SRR92, MBB92, NAI93, HJK93, KFY94, BUV95] dans la littérature ; quelques uns d'entre eux [FYA91, HJK93, KFY94] comptent sur le concept de k-coterie. Les autres [SRR92, MBB92, BUV95] comptent sur le concept de jeton, certains utilisent k jetons et d'autres n'utilisent qu'un seul.

Le problème de la k-exclusion mutuelle à été résolu en premier par Raymond [RAM89a], qui fournit une simple extension de l'algorithme de Ricart et Agrawala [RIA81]. Srimani et Reddy [SRR92] améliorent ce protocole en utilisant la notion de privilège de Suzuki et

Kasami [SUK85]. Cette solution réduit le nombre de messages requis pour résoudre l'exclusion mutuelle, mais elle est vulnérable aux échecs. Ensuite, il y a eu un intérêt considérable pour les méthodes de tolérance aux fautes pour résoudre le problème de la k-exclusion mutuelle basé sur la notion de quorums [GIF79]. Fujita et al [FYA91] discutent quelques techniques simples et ensuite proposent un schéma avec des petites tailles de quorums. Huang et al [HJK93] proposent une méthode alternative avec des petites tailles de quorums et une haute disponibilité dans la présence de pannes. L'algorithme du quorum majoritaire [THO79], [GIF79] pour la 1-exclusion mutuelle a été utilisé pour développer des protocoles basés-quorums pour la k-exclusion mutuelle par un partitionnement convenable des sites. Fujita et al [FYA91] ont utilisé cette approche dans laquelle les sites sont partitionnés en k groupes est les quorums sont construits afin que l'exclusion mutuelle soit assurée dans chaque groupe. Agrawal et al [AEE97] ont généralisé l'algorithme du quorum majoritaire pour construire des quorums pour la k-exclusion mutuelle.

2.2.1 Travaux antérieurs

Basé sur l'algorithme de Ricart et Agrawala [RIA81], **Raymond** [RAM89a] a proposé le premier algorithme qui permet à k nœuds d'accéder à la section critique simultanément. Cet algorithme ressemble à celui de Ricart et Agrawala (décrit précédemment) excepté que seulement (N-k) messages *Reply* suffisent pour entrer en section critique. Donc, la borne inférieure pour le coût de communication pour chaque entrée en section critique est (2N-k-1). Cependant, chaque message de requête s'attire un message *Reply*, alors, 2(N-1) est la borne supérieure du coût de communication. L'algorithme de Raymond est simple à implémenter mais il souffre aussi des mêmes inconvénients que l'algorithme de Ricart et Agrawala, car il génère beaucoup de messages et il n'est pas tolérant aux pannes.

Pour rendre les protocoles de la k-exclusion mutuelle tolérants aux fautes de nœuds et aux pannes de communication, plusieurs protocoles basés sur des stratégies comme celle de coterie ont été proposés. Dans ce but, **Fujita, Yamashita et Ae** ont proposé un algorithme de la k-exclusion mutuelle basé k-coterie [FYA91, KFY93]. Informellement, un quorum est une collection de nœuds et une k-coterie est une collection de quorums qui contient k paires de quorums disjoints [FYA91, HJK93]. Deux définitions différentes des k-coteries sont données dans la littérature : celle de Fujita, Yamashita et Ae [FYA91], et celle de Hiang, Jiang et Kuo |HJK93]. La première est plus restrictive que la dernière, et nous adoptons la plus restrictive (c-à-d, celle proposée par Fujita, Yamashita et Ae [FYA91]). Dans leur algorithme un nœud doit recevoir la permission d'un quorum de nœud avant l'accès à une ressource. Puisqu'une k-coterie contient au plus k paires de quorums disjoints, alors au plus k nœuds peuvent accéder aux ressources simultanément. Le concept de k-coterie maintient les avantages du concept de coterie, à savoir, la tolérance aux fautes et le bas coût de communication. Cependant, son implémentation est plus complexe.

Basé sur l'algorithme de Suzuki et Kasami, **Srimani et Reddy** [SRR92] ont proposé un algorithme distribué qui permet à k nœuds d'accéder à la section critique simultanément et utilise k jetons; si un nœud possède le jeton, il peut entrer directement en section critique. Donc, la borne inférieure du coût de communication par section critique de cet algorithme est "0" dans le cas où le jeton est présent localement. Si un nœud ne possède pas le jeton, il envoie des messages *Request* à tous les autres (N-1) nœuds. Lorsque aucun de ces nœuds ne veut entrer en section critique et chaque jeton est résident chez un nœud différent, le nœud demandeur va recevoir tous les k jetons. Donc, N+k-1 est la borne supérieure du coût de communication pour cet algorithme. Comme l'algorithme de Suzuki et Kasami, l'algorithme

de Srimani et Reddy a un degré minime de tolérance aux fautes. Car, si un jeton est perdu du a une panne de nœud ou de lien, quelques entrées ne seront pas disponibles, et des algorithmes complexes de détection et de régénération de jeton doivent être exécutés.

Contrairement à l'algorithme de Srimani et Reddy qui utilise k jetons, l'algorithme de **Makki et al** [MBB92] n'utilise qu'un seul jeton avec un compteur pour garder la trace du nombre de processus qui sont actuellement en section critique. Il est montré dans [BUV95] que sous une haute charge, ce système se comporte comme un système à une seule ressource. Aussi, dans certaines étapes de l'algorithme de Makki, il est exigé que tous les nœuds communiquent, ce qui le rend peu efficace.

Huang, Jiang et Kuo [JHK97] ont proposé une structure nommée cohortes pour la construction des quorums dans une k-coterie et ils ont montré que par cette structure, nous pouvons former correctement des quorums dans une k-coterie et donc réaliser une entrée multiple à la section critique tolérante aux fautes. La taille du quorum de la k-coterie dérivée de la structure de cohortes est constante dans le meilleur cas. La complexité de messages est entre 2(S-k+1) et 2(N-1), avec S la taille de la cohorte. Dans le mauvais cas, cet algorithme génère le même nombre de messages que l'algorithme de Raymond, ce qui n'est pas efficace.

Un autre algorithme de la k-exclusion mutuelle distribuée a été proposé par **Kakugawa, Fujita, Yamashita et Ae** dans [KFY94]. Il étend la stratégie du quorum majoritaire à la stratégie du quorum k-majoritaire. L'algorithme est le même que celui de Maekawa excepté la différence suivante : Dans l'algorithme de Maekawa, pour chaque processus *p*, un quorum est déterminé statiquement, et tient à rassembler la permission de tous les membres de *Q*. Cette approche peut être raisonnable pour résoudre le problème de la 1-exclusion mutuelle, puisqu'en échouant à rassembler la permission de *Q* par exemple, ça insinue qu'un autre processus est en section critique, c-à-d, *p* ne peut pas rassembler la permission de n'importe quel quorum. De l'autre coté, lorsque le problème de la k-exclusion mutuelle est considéré, le fait d'insister sur *Q*, ne paraît pas être une bonne idée, car *Q* peut être déjà pris. Dans ce cas, *p* peut trouver un autre quorum de qui il peut rassembler les permissions, parce qu'il y a (k-1) quorums qui ne croisent pas avec *Q*. Donc, l'algorithme de Kakugawa et al essaie de trouver un tel quorum. Le nombre de messages exigé par entrée en section critique est 3|*Q*| dans le meilleur cas, puisqu'un processus envoie *Request*, reçoit *Ok*, et envoie *Release* de tous les membres de *Q*, où *Q* est le quorum choisi par le processus dans *C* (*C* est la k-coterie utilisée dans l'algorithme). En utilisant la méthode de construction de k-coterie proposée par Fujita, Yamashita et Ae dans [FYA91] dont laquelle la taille du quorum est O(√n log n), la complexité de messages de l'algorithme de Kakugawa et al devient O(√n log n) dans le meilleur cas. Lorsqu'un processus *p* échoue à rassembler la permission de tous les membres d'un quorum, pas comme l'algorithme de Maekawa, l'algorithme choisi un autre quorum est essaie de rassembler la permission des membres du nouveau quorum. Par conséquent, l'algorithme est, par aucun moyen effectif. Dans le plus mauvais cas 6*n messages par entrée en section critique sont requis, où n est le nombre de processus. Le mauvais cas se produit à un processus *p*, lorsque pour tout processus *q* (≠*p*), *p* envoie *Request* à *q*, *q* envoie *Query* à quelques processus *r*, *r* envoie *Relinquish* à *q, q* envoie *Ok* à *p, p* envoie *Release* à *q*, et *q* envoie *Ok* à *r*. La complexité de messages 6*n est extrêmement mauvaise, mais en introduisant une fonction de borne c(n) (≥c) qui limite le nombre de processus à qui un processus peut envoyer une requête, la complexité de message dans le mauvais cas peut être réduite à 6*c(n), aux dépend de l'augmentation du temps d'attente pour enter en section critique. La taille du quorum est (n+1)/(k+1) et l'algorithme peut tolérer entre n-k[(n+1)/2k] et 0 panne de nœud.

L'algorithme de la k-exclusion mutuelle de **Bulgannawar et vaidya** [BUV95] est une amélioration et une extension de l'algorithme de la 1-exclusion mutuelle de Trehel et Naimi [NTM87]. Les nœuds dans le système sont numérotés de 1 à N. Il existe k jetons dans le système, numérotés de 1 à k. Initialement le jeton t est possédé par le nœud t, $1 \leq t \leq k$. Chaque nœud maintient un tableau de pointeurs avec une entrée pour chaque jeton. Ces pointeurs définissent k forêts correspondants aux k jetons, une forêt est une collection d'arbres. Donc, initialement, Pointer[t] pour chaque nœud est égal à t, $1 \leq t \leq k$, ie, chaque forêt contient juste un arbre, avec le nœud t à la racine de l'arbre du jeton t. L'algorithme utilise donc k jetons et une structure de forêts dynamique pour chaque jeton. Cette structure est utilisée pour diffuser les jetons. L'algorithme est désigné pour minimiser le nombre de messages et aussi le délai pour entrer en section critique, aussi bien pour des basses que des hautes charges. Dans leur article, ils ont présenté des résultats de simulation plutôt que des estimations analytiques du nombre moyen de messages requis par entrée en section critique. Les paramètres de performances utilisées sont le temps moyen pour entrer en section critique, le nombre moyen de messages par entrée en section critique et l'information moyenne par message. Ils ont simulé leur algorithme et l'ont comparé avec trois autres algorithmes, celui de Raymond [RAM89a], de Srimani et Reddy [SRR92] et de Makki et al [MBB92]. Ils ont montré que leur algorithme réalise le plus bas délai en entrant en section critique aussi bien que le plus petit nombre de messages sans une augmentation sérieuse de la taille des messages. Toutefois, ils n'ont pas spécifié les règles ou les méthodes pour choisir un jeton.

Jiang, Huang et Kuo ont proposé dans [JHK97] un autre algorithme avec une solution tolérante aux fautes pour la k-exclusion mutuelle distribuée. La solution utilise la structure logique de cohortes pour construire les quorums d'une taille constante dans le meilleur cas. Une structure de cohorte, Coh(k,l), a l paire de cohortes disjointes avec la première cohorte qui a k membres et les autres ont plus que (2k-2) membres. Lorsque quelques sites sont inaccessibles, la taille du quorum augmente graduellement et peut être aussi grande que O(n), où n est le nombre de sites. Dans [JHK97], les performances de la coterie de cohortes ont été comparées avec ceux de la k-coterie majoritaire et la k-coterie singleton en terme de taille de quorum et de disponibilité. Résultat, la k-coterie singleton peut être considérée comme un type spécial de coteries de cohorte. La taille du quorum de la k-coterie majoritaire est [(n+1)/(k+1)] alors que la taille du quorum de la coterie de cohortes est entre deux bornes 2 (si k=1) ou k (si k>1) et O(n). En ce qui concerne la disponibilité, ils ont montré que la disponibilité des coteries de cohortes est haute par rapport aux k-coteries majoritaires. L'algorithme tolère entre n-ks+(k(k-1))/2 et s-k+1 pannes de nœuds.

Une autre généralisation du quorum majoritaire pour la solution du problème de la k-exclusion mutuelle distribuée a été proposée par **Agrawal, Egecioglu et ElAbbadi** dans [AEE97]. Ce schéma produit une famille de quorums de tailles variées et les disponibilités sont indexées par les diviseurs intégrants *r* et *k*. Les cas *r* =1 et *r* =k correspondent aux algorithmes connus de génération de quorums basés majorité MAJ et DIV, alors que les valeurs intermédiaires de *r* interpolent entre ces deux extrêmes. Dans cet algorithme [AEE97], ils partitionnent les *n* nœuds en *k* classes avec chaque classe utilisant toutes approches traditionnelles pour mettre en vigueur la 1-exclsuion mutuelle. Quand l'approche traditionnelle utilisée est la stratégie du quorum majoritaire, les quorums seront appelés DIV des quorums majoritaires. La taille du quorum est (n+k)/2k, et l'algorithme peut tolérer jusqu'à n-k[(n+k)/2k] pannes de nœuds.

Pour réduire l'overhead en supportant la tolérance aux fautes pour la k-exclusion mutuelle, **Chang et Chen** [CHC97] ont proposé deux stratégies appelées quorum de l'arbre binaire

généralisé et quorum de l'arbre binaire étendu pour la k-exclusion mutuelle, qui imposent une structure logique sur le réseau. Les deux stratégies proposées sont basées sur une structure d'arbre binaire logique. La taille du quorum construit à partir des deux stratégies est $[\log_2 n/k]$ dans le meilleur cas et [n+k/2k] dans le mauvais cas. De plus, les deux stratégies peuvent tolérer jusqu'à $(n-k[\log_2 n/k])$ fautes dans le meilleur cas et $k([\log_2 n/k]-1)$ fautes dans le mauvais cas. De leur analyse de performances, ils ont montré que la stratégie de l'arbre binaire étendu fournit presque tout le temps une plus haute disponibilité que les stratégies, k-majorité, cohortes, et DIV. De plus, la taille du quorum de l'arbre binaire étendu est toujours la plus petite parmi ces quatre stratégies quand n>12. Alors que dans le mauvais cas, la taille du quorum est toujours plus petite que la taille du quorum dans les stratégies 4-majorité et DIV. Cependant, la stratégie du quorum de l'arbre binaire étendu n'est pas complètement distribuée, elle assigne la plus importante charge à un certain nombre de nœuds.

2.3 La k exclusion mutuelle dans les réseaux mobiles ad hoc

La k-exclusion mutuelle est un problème fondamental dans les systèmes distribués, il est aussi important dans les réseaux mobiles ad hoc puisque les utilisateurs de ce système sont appelés à partager des ressources communes qui, souvent, existent en plusieurs exemplaires ; par exemple le partage de bande passante ou d'une base de données répliquée.

Dans la section précédente, nous avons vu qu'il y a trois algorithmes de la k-exclusion mutuelle basés jeton [SRR91, MBB92, BUV95].

L'algorithme de [SRR91] utilise *k* jetons et égale la possession de tout jeton avec la capacité d'accéder à la section critique. Cet algorithme exige au nœud qui veut entrer en section critique d'envoyer (n-1) requêtes aux autres nœuds afin d'allouer l'accès à la ressource, s'il ne possède pas déjà le jeton. Cette inondation de requêtes le rend pauvrement convenable aux environnements ad hoc qui ont une faible énergie.

L'algorithme de [MBB92] utilise un seul jeton avec un compteur pour garder la trace du nombre de nœuds qui sont actuellement en section critique. Sous une haute charge, ce système se comporte comme un système à une seule ressource. De plus, dans certaines étapes de l'algorithme, la communication entre tous les nœuds est exigée en le rendant inefficace pour les environnements ad hoc.

Bulgannawar et Vaidya [BUV95] présentent un algorithme qui utilise k jetons séparés dans le système. Les requêtes pour un jeton suivent un ou k arbres couvrants dynamiques. Comme les requêtes traversent un arbre de la feuille vers la racine, la compression de chemin est utilisée pour garder l'arbre assez profond que possible. Bien que les nœuds demandent les jetons par des identificateurs uniques, l'algorithme permet la substitution de tout jeton pour allouer l'accès en section critique.

Chacun de ces algorithmes basés jeton suppose que le réseau est fiable et entièrement connecté, en permettant à tout processus de communiquer directement avec tout autre processus. Ces suppositions les rendent pauvrement adaptés à l'environnement ad hoc, où les liens se forment et se défaillent en conséquence de la mobilité.

Nous présentons dans cette section le seul algorithme connu de k-exclusion mutuelle dans les réseaux mobiles ad hoc existant dans la littérature [WCM01] et qui a été modifié dans [WCM01a] en vue d'obtenir de meilleures performances en temps de réponse aux requêtes en

attente. Cet algorithme est une généralisation de l'algorithme RL [WWV98] présenté dans le chapitre précédent.

2.3.1 Travaux antérieurs

Dans [WCM01], Walter présente un algorithme de la k-exclusion mutuelle basé jeton pour les réseaux mobiles ad hoc qui provoque un graphe dirigé acyclique (DAG) sur le réseau, la structure logique est modifiée dynamiquement pour l'adapter aux changements physiques de la topologie dans l'environnement ad hoc. Cet algorithme est une généralisation de l'algorithme RL [WWV98] de la 1-exclusion mutuelle présenté dans le chapitre précédent, il est appelé justement l'algorithme **KRL** (pour k-Reverse Link). Il maintient *k* jetons dans le système ; quand k =1, le nœud qui a le plus faible identificateur va prendre le jeton et quand $k>1$, il y a k processus de 0 à $k-1$ que chacun d'eux va prendre un jeton.

L'algorithme de Walter [WCM01] combine des idées de plusieurs travaux. La technique d'inversement partiel de [GAB81], utilisé pour maintenir une destination orientée DAG dans un réseau radio par paquet lorsque la destination est statique, elle est utilisée dans son algorithme pour maintenir un jeton orienté DAG avec k destinations dynamiques. Comme l'algorithme de [RAM89], [CSL90], et [DHK94], chaque processus dans l'algorithme de Walter [WCM01] maintient une file de requêtes qui contient les identificateurs des processus voisins de qui il a reçu des requêtes pour le jeton. Comme [DHK94], l'algorithme KRL ordonne totalement les nœuds. Il maintient k jetons dans le système comme dans [BUV95]. Quand k=1, le nœud inférieur est toujours actuellement le détenteur du jeton, lui rendant un « évier » vers lequel toutes les requêtes sont envoyées. Quand k>1, il peut y avoir plusieurs éviers dans le système. Cependant, Walter a montré que tous les nœuds qui ne possèdent pas de jeton ont toujours un chemin vers un détenteur de jeton.

Comme dans l'algorithme RL de la 1-exclusion mutuelle [WWV98], chaque nœud choisit dynamiquement sont plus petit voisin comme son lien préféré vers le possesseur du jeton. Les nœuds sentent les changements de liens vers les voisins immédiats et déroutent les requêtes en se basant sur l'état du lien préféré précédent pour le possesseur du jeton et les contenus actuels de la file de requêtes locale. Toutes les requêtes arrivant au détenteur du jeton sont traitées systématiquement afin que ces requêtes soient entretenues continuellement pendant que le DAG est réorienté et les requêtes bloquées sont déroutées. Dans cet algorithme à jeton multiple [WCM01], il est possible que les processus reçoivent des requêtes quand ils sont en section critique. Si cela arrive, les processus peuvent satisfaire ces requêtes immédiatement s'ils possèdent plusieurs jetons, en augmentant l'accès concurrent à la section critique.

Les résultats de simulation comparent la performance de cet algorithme à l'algorithme de la k-exclusion mutuelle distribuée statique de Bulgannawar et Vaidya [BUV95] s'exécutant au-dessus d'un protocole de routage ad hoc idéal qui fournit toujours les plus courts chemins entre les nœuds, cette combinaison est appelée BVR pour (Bulgannawar/Vaidya avec routage). L'algorithme [BUV95] est utilisé parce qu'il exige moins de communications que les autres algorithmes de la k-exclusion mutuelle basés- jeton. Aussi, cet algorithme ne prévoit pas de défaillances et de récupération de lien et doit compter sur la couche de routage pour maintenir les chemins logiques s'il s'exécute dans un réseau dynamique. La simulation montre que l'algorithme KRL est meilleur que la combinaison BVR dans quelques scénarios, alors l'algorithme KRL est aussi meilleur que l'algorithme de Bulgannawar et Vaidya dans ces scénarios quand n'importe quel algorithme de routage ad hoc réel est utilisé.

a. L'algorithme KRL

i) Exemples de scénarios de fonctionnement de l'algorithme KRL

Les exemples de l'opération de l'algorithme KRL dans un réseau statique et dynamique sont très similaires aux exemples de l'algorithme RL, présenté précédemment. La différence principale, causée par la présence de plusieurs jetons dans le système, est que les nœuds qui possèdent le jeton peuvent réduire leur taille quand ils n'ont pas de liens entrants.

Premièrement, le cas d'un réseau statique est discuté, suivi par un réseau dynamique. Une illustration de l'algorithme dans un réseau statique (dans lequel les liens ne faillent et ne se forment pas) est présentée en figure 6. Des snapshots de la configuration du système durant l'exécution de l'algorithme sont montrés, avec incrément du temps de 6(a) à 6(f). Les liens sans fil directs sont représentés par des lignes de tirets connectant les nœuds sous forme de cercles. La flèche dans chaque lien sans fil pointe du nœud avec la plus grande taille vers le nœud avec la plus petite taille. La file de requêtes dans chaque nœud est représentée par un rectangle, la taille est montrée par 3-tuple, et les détenteurs de jetons (k=2) par des cercles ombrés. Les pointeurs *next* sont montrés par des flèches solides. Notons que quand un nœud possède un jeton, son pointeur *next* est dirigé vers lui-même.

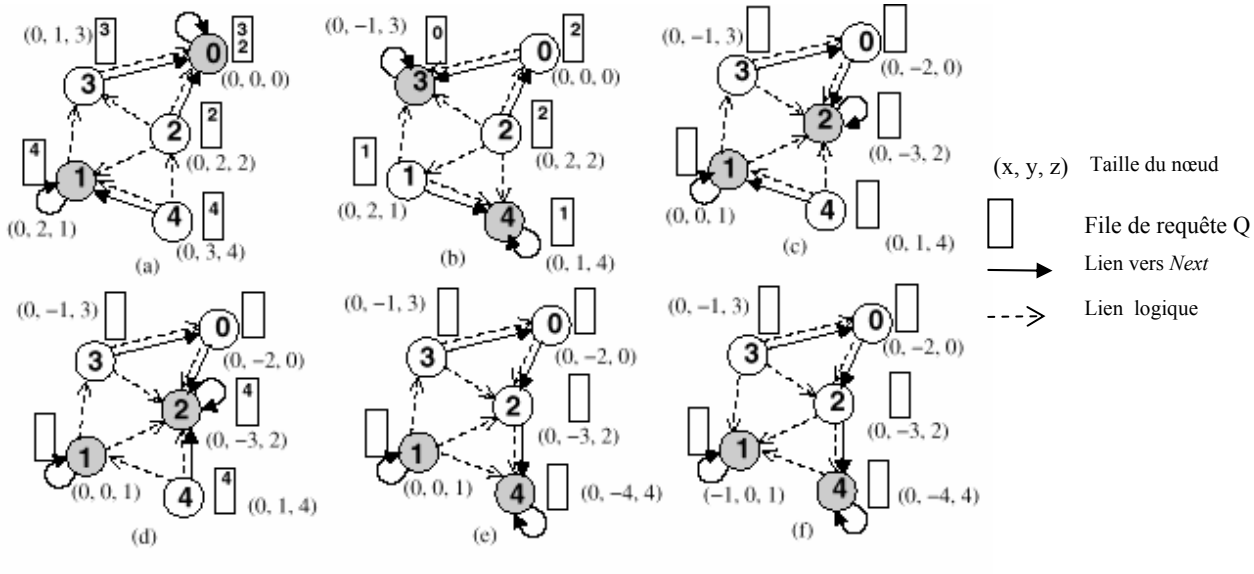


Figure 6 : Scénario de fonctionnement le l'algorithme KRL dans un réseau statique.

Dans la figure 6(a), les nœuds 2, 3, et 4 ont demandé l'accès à la section critique (notons que les nœuds 2, 3, et 4 ont enfilé leurs identificateurs dans respectivement, Q_2, Q_3, et Q_4) et les nœuds 2 et 3 ont envoyé des messages *Request* au nœud 0, qui enfile leurs identificateurs dans Q_0 dans l'ordre dans lequel les messages *Request* sont reçus. Le nœud 4 envoie une requête au nœud 1, puisque le nœud 1 est le nœud voisin qui a la plus petite taille. La partie (b) représente le système après un moment, quand le nœud 1 envoie un jeton au nœud 4 et demande aussi l'accès à la section critique en envoyant un message *Request* au nœud 4 (notons que 1 est enfilé dans Q_1 et Q_4). Le nœud 0 envoie un jeton au nœud 3 suivant le jeton avec une requête de la part du nœud 2 (notons que 0 est enfilé dans Q_3). Observons que la direction logique des liens entre le nœud 0 et le nœud 3 et entre le nœud 1 et le nœud 4 changent pour être dirigée loin des nœuds 3 et 4 dans la partie (a), pour être dirigée vers les nœuds 3 et 4 dans la partie (b), quand les nœuds 3 et 4 reçoivent les messages *Token* et réduisent leurs tailles. Remarquons aussi que les nœuds 0 et 3 ayant les pointeurs *next* dirigés

vers le nœud 0 et les nœuds 1 et 4 ayant les pointeurs *next* dirigés vers le nœud 1 dans la partie (a), changent vers le nœud 3 pour les deux nœuds 0 et 3 et vers le nœud 4 pour les deux nœud 1 et 4 dans la partie (b).

La figure 6(c) montre la configuration du système après que le nœud 4 ait libéré la section critique et envoie un message *Token* au nœud 1. Le nœud 3 a libéré aussi la section critique et a envoyé un message *Token* au nœud 0. Le nœud 0 envoie alors le jeton au nœud 2. A ce point de l'exécution, il n'a pas de requêtes en suspens, comme peut être vu par les files de requêtes vides.

La partie (d) montre la configuration du système après que l'application hôte au niveau du nœud 4 ait émis une requête pour entrer en section critique et que le nœud 4 ait choisit son plus petit voisin, le nœud 2, comme *next* et lui envoie *Request*.

Dans la partie (e), le nœud 4 reçoit un message *Token* du nœud 2, réduit sa taille et entre en section critique. Le nœud 1 a reçu un message *LinkInfo* du nœud 4 et sente qu'il n'a pas de liens entrants.

Dans la partie (f), le nœud 1 a réduit sa taille pour être plus petite que celles des voisins, afin que les requêtes futures aient une chance d'être dirigées vers le nœud 1. Observons que l'élément du milieu, h2, de chaque tuple *myHeight* de nœud est décrémenté de 1 à chaque saut traversé par le jeton, afin qu'un détenteur de jeton ai toujours la plus petite taille dans le système.

Maintenant, nous considérons l'exécution de l'algorithme KRL dans un réseau dynamique. L'information taille permet à chaque nœud *i* de garder la trace de la direction logique actuelle des liens aux nœuds voisins, particulièrement au nœud choisi pour être *next*. Si le lien vers *next* change et $|Q|>0$, le nœud *i* doit re-router ses requêtes en appelant *ForwardRequest*().

La figure 7(a) montre le même snapshot d'exécution du système montré dans la figure 6(a), avec incrément de temps de 7(a) à 7(e). La figure 7(b) représente la configuration du système après que le nœud 3 se déplace en relation des autres nœuds dans le système, résultant en un réseau qui est temporairement non orienté jeton, puisque le nœud 3 n'a pas de liens sortants. Le nœud 0 s'est adapté à la perte du lien avec le nœud 3 en enlevant 3 de sa file de requêtes. Le nœud 2 ne prend aucune action après la perte de son lien vers 3, puisque le lien vers next2 n'est pas affecté et le nœud 2 possède encore un lien sortant. Dans la partie (c), le nœud 3 s'est adapté à la perte de son lien vers le nœud 0 en augmentant sa taille et en envoyant un message *Request* au nœud 1. Les parties (d) et (e) montrent le système après que le nœud 0 ait envoyé un jeton au nœud 2 et le nœud 4 ait envoyé un jeton au nœud 1, qui l'envoie ensuite au nœud 3.

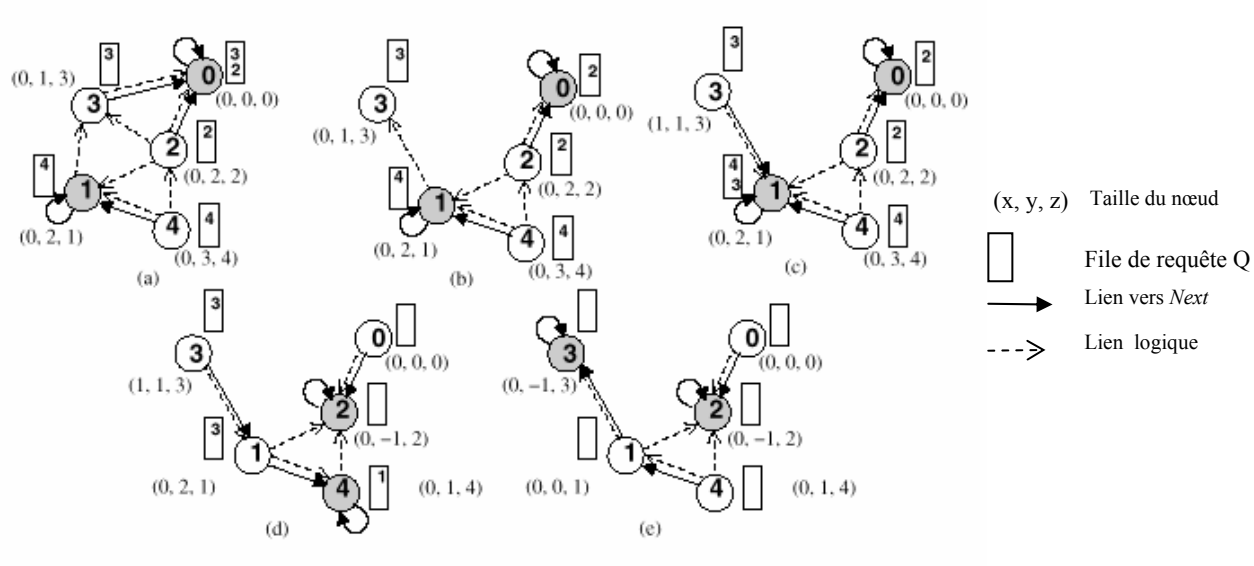


Figure 7 : Scénario de fonctionnement le l'algorithme KRL dans un réseau dynamique.

b. L'algorithme KRL avec envoi du jeton (KRLF) [WCM01a]

La figure 8 montre l'exécution de l'algorithme dans lequel il y'a deux nœuds qui possèdent le jeton (3 et 5). Les nœuds 4 et 6 font leurs requêtes au nœud 5, le nœud 5 est dans la section critique et il a mis 4 et 6 dans sa file de requêtes Q_5.

Le nœud 3 quitte la section critique et n'envoie le jeton que lorsqu'il reçoit un message de requête, mais les nœuds 4 et 6 doivent attendre leur tour pour avoir le jeton utilisé par le nœud 5.

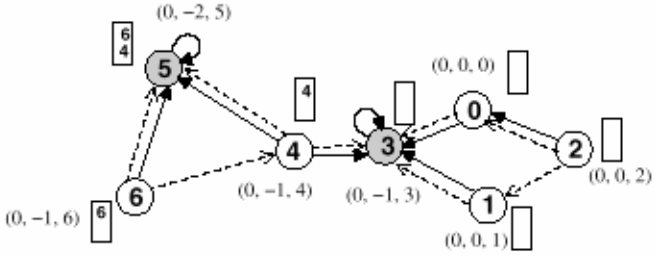


Figure 8 : Exécution de l'algorithme KRL avec envoi du jeton.

Pour remédier à ce problème, le nœud envoie le jeton à d'autres parties du réseau. Pour cela, il choisit le voisin qui a la plus faible taille et lui envoie le jeton à ce voisin.

Le choix du plus faible voisin est dû au faible nombre de liens inversés. Les nœuds gardent une trace de leurs voisins qui ont envoyé le jeton et qui ont une file de requêtes vide en marquant le lien comme visité.

La figure 9 illustre ces modifications durant l'exécution de l'algorithme. Dans la figure 9(a), le nœud 3 a un jeton, mais aucun voisin ne veut accéder en section critique. La figure 9(b) montre l'état du système après que 3 ait quitté sa section critique, et le jeton est envoyé à travers 0, 2, 1 et 3 (partie gauche du réseau), La lettre V sur chaque lien signifie que ce lien est marqué comme visité par le nœud émetteur et le nœud récepteur du jeton. Quand le nœud

3 reçoit le jeton et il a une file de requêtes vide, il marque tous ses liens comme non visités et recommence le processus d'envoi du jeton.

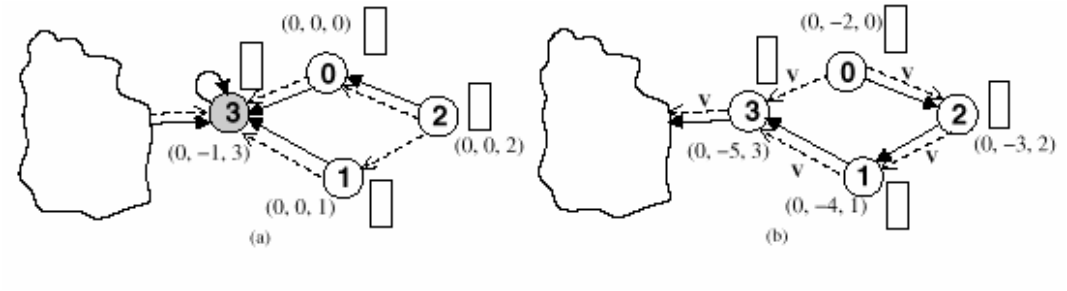


Figure 9 : Modifications durant l'exécution de l'algorithme KRL avec envoi du jeton.

2.4 Tableau comparatif

L'étude des algorithmes de la k-exclusion mutuelle existants a donné naissance à la comparaison présentée dans le tableau 4.

Environnement	Algorithme	Approche	Structure			Robustesse (Tolérance aux pannes)	Performances		Avantages	Inconvénients
			Type	Technique	Politique		NME	Taille Mémoire		
SD	Raymond **1989**	Consensus total	Diffusion	HL	Fifo	Non	Entre 2n-k-1 et 2(n-1)		1- Simple à concevoir.	1- Génère beaucoup de messages 2- N'est pas tolérant aux fautes.
	Fujita, Yamashita et Ae **1991**		Introduction des k-coterie			Un certain degré			1- Génère moins de messages. 2- Tolérant aux fautes	L'implémentation est plus complexe que celle des algorithmes basés-coterie.
	Srimani et Reddy **Janvier1992**	K jetons				Non	Entre 0 et N+K-1		1- Génère moins de messages, surtout au meilleur cas.	1- Le nombre de messages reste comme même élevé. 2- Si un jeton est perdu du a une panne de nœud ou de lien, quelques entrées ne seront pas disponibles, et des algorithmes complexes de détection et de régénération de jeton doivent être exécutés.
	Makki, Banta, Been, Pissinou et Park **Décembre 1992**	Un seul jeton		Compteur, (garde trace du nombre de nœuds qui sont en SC)		Non				1- Sous une haute charge, le système se comporte comme un système à une seule ressource. 2- Dans certaines étapes, l'algorithme exige la communication de tous les nœuds.
	Hiang, Jiang et Kuo **1993**	Quorum (consensus partiel)	K-coterie	Cohortes	Fifo	Oui (haute)	En utilisant $C^{k,n}(S)$, S étant la taille d'une cohorte, leNME est entre 2(S-k+1) et 2(N-1)		1- Sa tolérance aux fautes. 2- Dans le meilleur cas, il a un nombre constant de messages.	1- Génère le même nombre de messages que l'algorithme de Raymond dans le mauvais cas.
	Kakugawa, Fujita, Yamashita et Ae **1994**	Quorum (consensus partiel)	k-Coterie structure de forêt dynamique pour chaque jeton (stratégie du quorum k-majoritaire)	HL + Maekawa + (Chois dynamique du quorum)	Fifo	Un certain degré Tolère entre n-k[(n+1)/2k] et 0 pannes de nœud.	Entre 3c et 6n, où c est la taille maximale du quorum et n le nombre de processus, si c= $\sqrt{n}\log_n$, alors la CM devient égale a $3(\sqrt{n}\log_n)$ La taille du quorum est (n+1)/(k+1)		1- Dans le meilleur cas, la complexité de messages est bien meilleure que celle des deux algorithmes précédents. 2- Il a certain degré de tolérance aux fautes dans le sens où il y a une possibilité qu'un processus peut entrer en SC même s'il y a une panne de processus et/ou de lien. 3- La stratégie qu'il utilise (k-majorité) est complètement distribuée.	1- Dans le mauvais cas l'algorithme génère beaucoup de messages, même en bornant le nombre de tentatives de resélection de quorums, les processus peuvent attendre un temps plus long que celui de l'algorithme original.

	Naimi **1993**									
	Bulgannawar et Vaidya **Novembre 1995**	K jetons	Structure de forêt dynamique pour chaque jeton	Les nœuds demandent les jetons par des identificateurs uniques de jeton, l'algorithme permet la substitution de tout jeton.	Fifo			1- N tableaux de pointeurs de taille k, 2- N files d'attente des nœuds. 3- K files de jetons	1- Plus petit délai pour enter en SC. 2- Plus petit nombre de messages.	1- La taille des messages est plus grande. 2- La structure de forêt utilisée est parfois violée par la formation de cycles. 3- Les règles ou les méthodes pour choisir un jeton ne sont pas définis (spécifiées).
	Manabe et Aoyagi **1993 [MAA93]**	Quorum (consensus partiel)	k-Coterie				O((2h+3)|q|+3)			Dans une k-coterie, lorsque le nombre de ressources augmente, |q| devient grand.
	Baldoni et Ciciani **1994 [BAC94]**	Consensus total			Exe mple Short job First		Entre $3[n^{k/(k+1)}-1]$ et $5[n^{k/(k+1)}-1]$			
	Neilsen et Mizuno **1994 [NEM94]**	1- Ils ont montré que les k-coteries ND sont les plus résistants et fournissent un haut niveau de disponibilité que les k-coteries qu'elles dominent. 2- Ils ont présenté un ensemble de méthodes pour construire des k-coteries nondominées.								
	Jiang et Huang **1994 [JIH94]**	1- Ils ont introduit un théorème pour vérifier la nondomination des k-coteries. 2- Ils ont définit une classe de k-coteries nondominées spéciale, à savoir, les k-coteries fortement nondominées, (SND, pour Strongly NonDominated k-coteries). 3- Ils ont proposé deux opérations, union et jointure, pour générer de nouvelles k-coteries SND à partir de k-coteries SND connues. 4- Ils ont montré que la k-coterie singleton est SND est que sous certaines conditions spéciales, la k-coterie majoritaire est SND.								
	Jiang, Huang et Kuo **Février 1997**	Quorum (consensus partiel)	Cohortes			Oui, Tolère entre n-ks+(k(k-1))/2 et s-k+1 pannes de nœud.	Taille du quorum est entre 2 (quand k=1) ou k (quand k>1) et n,		1- Dans le meilleur cas les quorums sont de tailles constantes. 2- Tolérant aux fautes. 3- La stratégie qu'il utilise (cohortes) est complètement distribuée.	Quand il y a des sites inaccessibles, la taille du quorum augmente progressivement et peut être aussi grande que O(n).
	Agrawal, Egecioglu et ElAbbadi **Avril 1997**	Quorum (consensus partiel)	DIV			Oui, Tolère n-k[(n+k)/2k] pannes de nœud.	Taille du quorum est (n+k)/2k		3- La stratégie qu'il utilise (DIV) est complètement distribuée.	

	Chang et Chen **1997**	Quorum (consensus partiel)	Structure d'arbre binaire étendu (QABE)			Oui, Tolère entre n-k[lg_2n/k] et k[lg_2n/k]-1 pannes de nœud.	Taille du quorum est entre lg_2 (n/k) et (n+k)/2k		1- La taille du quorum du QABE est la plus petite parmi les trois stratégies (4-majorité>cohortes>DIV>QABE), quand n>12. 2- La taille du quorum est la plus petite que la 4-majorité et DIV quand les pannes de nœuds se produisent dans le mauvais cas. 3- Fournit la plus haute disponibilité que la k-majorité, cohortes et DIV.	1- La stratégie du quorum de l'arbre binaire étendu n'est pas complètement distribuée, elle assigne la plus importante charge à un certain nombre de nœuds.
Ad hoc	Walter **2001**	K jetons	DAG	Inversement partiel de liens	fifo			1- Chaque nœud maintient une file qui contient les identificateurs des nœuds voisins demandant le jeton.	1- La structure logique de la topologie du réseau correspond aux liens physiques en traitant les cas de destruction et formation de liens. 2- Ne dépend pas d'un protocole de routage spécifique. 3- Les nœuds ne contiennent des informations que sur leurs voisins directs.	1- Quelques hypothèses ne sont pas réalistes comme le partitionnement du graphe ne peut pas se produire. 2- Un nœud qui a envoyé une requête et perd son lien vers son voisin *Next*, renvoie la requête vers un autre voisin, ce qui augmente le temps d'attente pour un nœud pour avoir le jeton et peut produire une situation de famine si les changements de la topologie ne s'arrête pas. 3- Des nœuds qui n'ont pas demandé l'accès en section critique participe à l'exécution de l'algorithme. 4- Un nœud privilégié qui n'est pas en tête de sa propre file envoie le jeton pour le demander une nouvelle fois.

Tableau 4. Comparaison entre les différents algorithmes distribués de la k-exclusion mutuelle.

2. 5 Conclusion

Dans cette section, nous avons étudié le problème de la k-exclusion mutuelle distribuée. Nous avons introduit et définit le concept de k-coterie comme extension de celui de coterie. Nous avons présenté aussi plusieurs algorithmes pour la k-exclusion mutuelle distribuée et nous les avions comparés à la base des critères de comparaisons des algorithmes de l'exclusion mutuelle.

Nous avons présenté l'algorithme KRL de la k-exclusion mutuelle dans les réseaux mobiles ad hoc. Cet algorithme est une adaptation de l'algorithme RL. Il est classé dans la catégorie des algorithmes qui règlent l'accès à k exemplaires d'une même ressource. Il a les mêmes avantages et inconvénients de l'algorithme RL. Il ne donne pas de bonnes performances en termes de messages et d'énergie car il introduit dans son exécution d'autres nœuds qui n'ont pas demandé l'accès à la section critique. Les résultats obtenus par la simulation [WCM01] et [WCM01a] montrent que l'algorithme KRLF possède une meilleure performance en terme de temps de réponse par entrée en section critique ; en particulier lorsque la charge (les demandes d'accès) augmente. Par contre, lorsque le système est sous une charge réduite l'algorithme KRLF est plus coûteux en terme de nombre de messages générés.

Chapitre 3

La h-k exclusion mutuelle

3.1 Introduction

Dans ce chapitre nous présentons le problème de la h-k exclusion mutuelle dans les systèmes distribués et dans les réseaux mobiles ad hoc.

Le problème de la h-k exclusion mutuelle, aussi connu par le problème d'allocation de *h* parmi *k* ressources est défini comme suit [RAY91a] :
Il existe k ressources identiques partagées par les processus dans U. Chaque ressource ne doit pas être assignée à plus d'un processus à la fois. Un processus demande h ($1 \leq h \leq k$) ressources toutes à la fois, et pour éviter l'interblocage, le processus est bloqué jusqu'à ce qu'il obtienne toutes les ressources demandées. Le processus peut alors commencer à utiliser les ressources ; une fois fini, il les libère toutes à la fois. Si h=1 pour chaque requête, le problème de la h-k exclusion mutuelle correspondra à celui de la k-exclusion mutuelle, de plus, si k=1 nous obtenons la 1-exclusion mutuelle.

Comme nous l'avons déjà vu, un réseau mobile ad hoc est un ensemble de nœuds mobiles qui sont dynamiquement et arbitrairement éparpillés tels que l'interconnexion entre les nœuds peut changer à tout moment. Les unités mobiles dans les réseaux mobiles ad hoc sont caractérisées par leurs fréquentes déconnexions et leurs sources d'énergie limitées. Elles échangent des messages dans un environnement où les liens de communication sont non fiables ; ces limitations vont imposer d'autres problèmes lors de la conception des algorithmes d'exclusion mutuelle, comme le traitement du partitionnement du réseau, les déconnexions, la perte des messages, la consommation d'énergie…etc.

Dans ce qui suit, nous présentons quelques algorithmes de la h-k exclusion mutuelle conçus pour un environnement statique. Ensuite, un algorithme de la h-k exclusion mutuelle pour les réseaux ad hoc proposé par J-R Jiang [JIA02] sera étudié. Cet algorithme est le seul trouvé dans la bibliographie.

3.2 La h-k exclusion mutuelle dans les systèmes distribués

Dans les systèmes distribués, plusieurs travaux sont effectués quant à l'exclusion mutuelle. Raynal propose le premier algorithme dans [RAY91a] et ensuite quatre algorithmes utilisant les ensembles d'arbitres, les k-arbitres, les coteries locales et les (h-k)-arbitres sont proposés dans [BAL94], [BMR95], [KAY96], [MAT99], respectivement. Par la suite, Kuo a introduit deux nouvelles opérations pour la construction de nouveaux k-arbitres à partir de k-arbitres connus qui soient meilleures et peuvent êtres non-dominés, à savoir l'opération de jointure et l'opération de composition de k-arbitres, dans [KUO01] et [KUO01a], respectivement. En 2002, Wada et Cheng ont proposé un algorithme de la h-k exclusion mutuelle qui repose sur une technique d'allocation de ressources appelée « technique de serrage » afin d'améliorer l'efficacité de l'utilisation des ressources et de réduire le temps d'attente des processus [WAC02]. Ensuite, deux algorithmes qui utilisent les k-coteries et les coteries de cohortes ont été proposées dans [JIA04] et [JIA04a] respectivement. Parmi tous ces algorithmes, seulement les algorithmes de Jiang sont résistants aux défaillances. Ils peuvent tolérer les pannes de nœuds et/ou de liens de réseau même quand les pannes mènent au partitionnement du réseau. Les algorithmes basés-quorums [BAL94, BMR95, KAY96, MAT99, JIA04 et JIA04a] sont des algorithmes type-Maekawa [MAE85] qui reposent sur le concept de l'horloge logique de Lamport [LAM78] et sur cinq type de messages, à savoir, *Request*, *Grant*, *Release*, *Inquire* et *relinquish* pour éviter l'interblocage et la famine.

3.2.1 Le Modèle du système

Tous les algorithmes qui seront présentés par la suite suivent un même modèle du système distribué. Un système distribué est un ensemble $U=\{P_1, P_2,\ldots,P_n\}$ de n processus. Ces processus communiquent en échangeant des messages et chaque processus possède une file de longueur infinie qui garde les messages arrivés. Chaque message est délivré en un temps fini et le délai de livraison des messages est imprévisible mais l'ordre des messages reste inchangé durant la livraison. Chaque paire de processus est reliée par un canal logique. De plus, chaque canal est supposé avoir une capacité infinie, être sans erreur, bidirectionnel et FIFO. Les processus ne partagent pas une horloge commune ou une mémoire commune. Cependant, chaque processus a sa propre horloge locale, mais les horloges peuvent indiquer des temps différents, et aucun processus ne peut donner le temps global. Aucune limite n'existe sur les vitesses relatives des processus. Les processus et les liens de communication acceptent l'erreur et faillent selon le modèle échec-arrêt (Le processus stoppe et reste hors-service). Les autres processus peuvent détecter cet état défaillant), et un échec peut être détecté par les autres processus. Finalement, La topologie du réseau est supposée complète et il y a m ressources $R=\{r_1,r_2,\ldots r_m\}$ partagées par les processus.

3.2.2 Travaux antérieurs

Dans [RAY91a], **Raynal** propose le premier algorithme distribué de la h-k exclusion mutuelle. L'algorithme de Raynal est une extension de l'algorithme de Ricart et Agrawala [RIA81]. Il demande à un nœud u d'envoyer des messages de requêtes à tous les autres nœuds et attend les réponses pour estimer le nombre de ressources disponibles. Un nœud *v* répond qu'il y a *k* ressources disponibles s'il n'utilise et ne demande pas de ressources partagées, ou s'il a la priorité inférieur que celle du demandeur (en terme d'ordre de l'horloge logique [LAM78]). De l'autre côté, le nœud *v* répond qu'il y a (k-h) ressources libres si *v* utilise h ressources ou si *v* les a demandé avec la plus haute priorité. Dans un tel cas, le nœud *v* doit retarder sa réponse jusqu'à ce qu'il libère les *h* ressources après sa sortie de la section critique. De toutes les réponses, si le nœud u trouve que le nombre estimé de ressources disponibles est supérieur au nombre de ressources demandées, il peut entrer en section critique. La complexité de messages de l'algorithme de Raynal est entre 2(n-1) et 3(n-1), où n est le nombre de nœuds. L'algorithme de Raynal est simple à implémenter, cependant, un processus doit envoyer des messages à chaque autre processus; ce qui n'est pas efficace. De plus, il n'est pas tolérant aux fautes puisqu'un nœud ne peut pas rassembler les réponses de tous les autres nœuds s'il y a un nœud qui est tombé en panne.

Dans [BAL94], **Baldoni** a proposé le concept « d'ensembles d'arbitres » pour résoudre le problème de la h-k exclusion mutuelle. Chaque nœud est associé a un ensemble d'arbitres, et tous k ensembles d'arbitres devraient avoir au moins un membre commun. Un nœud doit envoyer les messages de requête avec le paramètre h ($h\leq k$) à tous les membres de son ensemble d'arbitre pour rassembler des permissions et accéder aux h ressources. Chaque nœud garde le nombre de ressources disponibles, qui est initialement k et est décrémenté de h après avoir accordé une demande pour accéder à h ressources. Le nombre total de ressources qui sont accédées concurremment ne peut pas dépasser k ; ceci est garanti parce que le membre commun de tous les (k+1) ensembles d'arbitre peut servir comme arbitre, qui accorde sa permission seulement lorsque le nombre de ressources libres n'est pas moins que le nombre de ressources demandées. Cet algorithme a une complexité de messages O(q), où q est la taille des ensembles d'arbitres. Baldoni prouve que les ensembles d'arbitre ont la taille inférieure à $O(n^{k/k+1})$ si tous les ensembles d'arbitres ont la même taille et chaque nœud

apparaît dans le même nombre d'ensembles d'arbitres. La méthode de construction des ensembles d'arbitres proposé par Baldoni a le problème qu'ont les k-coteries ; à savoir, lorsque le nombre de ressources augmente, la taille du quorum devient grande, en effet, lorsque k augmente, la taille du quorum approche O(n).

Le concept d'ensembles d'arbitres a été ensuite formalisé par la notion « k-arbitres » par **Baldoni** et **al** dans [BMR95]. Un k-arbitres est une collection d'ensembles d'arbitres minimaux (appelé quorums) où tous les k+1 quorums ont au moins un membre commun. Deux k-arbitres sont proposés dans [BMR95] : le k-arbitre (k+1)-cube et le k-arbitre uniforme, avec les tailles de quorum $(k+1).n^{k/(k+1)}$ et [k.n/k+1]+1, respectivement. Leur algorithme est une extension de l'algorithme de Maekawa, en utilisant son approche, chaque processus peut agir comme un processeur demandeur et un arbitre afin de résoudre les conflits de l'acquisition des ressources partagées. Cependant, leur algorithme a besoin de beaucoup de messages car la taille des quorums dans un k-arbitres est beaucoup plus grande que celle des quorums dans la 1-coterie. La complexité de messages des algorithmes basés k-arbitres, est entre 3q et (3h+3)q.

Dans [KAY96], **Kakugawa** et **Yamashita** ont introduit le concept de Coteries locales. L'idée de base d'une coterie locale est que les processus qui ne demandent et ne partagent aucune ressource n'interfèrent pas avec les autres processus. Dans leur algorithme, les processus maintiennent ensemble une base de données distribuée qui détient des paires (processus, ressource), chacune consiste en un processus et une ressource à laquelle il accède. Ils ont utilisé les coteries locales pour implémenter justement cette base de données distribuée. Un processus u qui veut accéder à k ressources, envoie une requête pour demander si h ressources sont disponibles pour lui, si la réponse est oui, alors h ressources sont alloué à u. La complexité de messages de l'algorithme de Kakugawa et Yamashita est entre 4*q et (7+h)q, où q est la taille du plus petit quorum de la coterie locale. Cet algorithme est tolérant aux fautes si une coterie singleton n'est pas utilisée comme coterie locale, les quorums sont conçus de telle sorte qu'ils ont la moindre intersection entre eux et la taille du quorum est plus petite que celle d'un k-arbitre. Cependant, l'algorithme est approprié aux cas où le nombre de processus est petit, car quand n est petit q est petit. De plus, l'algorithme est moins efficace dans le mauvais cas, lorsque h est grand, malgré que ce cas semble ne pas se produire souvent.

Les quorums dans les k-arbitres sont beaucoup plus grand que ceux de la 1-coterie, donc l'algorithme basé k-arbitres génère beaucoup de messages. Pour cette raison, **Manabe** et **Tajima** ont introduit dans [MAT99] une nouvelle solution afin de réduire la taille des quorums. Ils ont généralisé les k-arbitres et ont introduit un nouveau concept qui est le (h-k)-arbitre. Il ont proposé le (h-k)-arbitre (k+1)-cube et le (h-k)-arbitre uniforme avec les tailles de quorum $(k+2-h).n^{(k+1-h)/(k+1)}$ et [k.n/(k+h)]+1, respectivement. Les quorums dans chaque (h,k)-arbitre sont plus petits que leurs correspondants dans le k-arbitre. Cependant, il est difficile d'obtenir un (h,k)-arbitre de taille minimale, et un (h,k)-arbitre non-dominé. La complexité de messages de leur algorithme est la même que celle des algorithmes basés k-arbitre, elle est entre 3q et (3h+3)q.

Dans [KUO99], **Kuo** a introduit le concept de (K, M)-Coterie pour résoudre le problème de l'allocation de k ressources distribuées parmi M. Les algorithmes distribués basés sur les (K, M)-Coteries pourraient avoir les avantages de haute capacité de tolérance aux fautes et un bas coût de la communication. La (K, M)-Coterie nondominée est une classe importante des (K, M)-Coteries qui a habituellement la meilleure performance que les autres (K, M)-coteries

dominées sur la capacité de tolérance aux fautes et sur le coût de la communication. Il a proposé une théorie pour reconnaître les (K, M) -Coteries nondominées. Il a présenté des méthodes simples de construire les (K, M)-Coteries et montre comment les rendres non-dominées en appliquant cette théorie. Il a introduit aussi le concept de (q, k)-arbitres Binomiaux avec des quorums uniformes dans [KUO01a].

Les études de recherches de [RAY91a, BAL94, BMR95, KAY96, MAT99] de la h-k exclusion mutuelle qu'on a vue jusqu'ici concentrent surtout sur la complexité de message et ont négligées l'efficacité de l'utilisation des ressources, car dans leur algorithmes, la distribution des ressources est basée sur un niveau de priorité absolu (estampillage). En 2002, **Wada** et **cheng** ont introduit dans [WAC02], une nouvelle méthode d'allocation de ressources qui emploie une technique appelée « serrage dégagé ». Par cette méthode, un processus ayant une priorité inférieure peut acquérir des ressources avant un autre processus qui a une plus haute priorité a condition que l'allocation de ce dernier ne sera pas différée. Ainsi, l'efficacité de l'utilisation des ressources est améliorée et le temps d'attente des processus est diminué. La communication entre les processus pour résoudre la concurrence pour la même ressource est exécutée à travers un manager de ressources (chaque ressource a un manager et peut avoir plusieurs unités). La complexité de messages de leur algorithme est O(P), où P et un ensemble de processus qui demandent des unités de ressources. Il est clair que la complexité de messages de leur algorithme est inférieure à celle de l'algorithme de Manabe et Tajima. Cependant dans un environnement distribué, il n'est pas désiré que la charge soit imposée à un manager de ressources car, la panne de ce nœud (le manager) arrête l'algorithme. Néanmoins, leur technique peut être améliorée en distribuant le calcul concernant le serrage de la ressource entre les processus qui demandent la ressource.

Les algorithmes de la h-k exclusion mutuelle [BAL94, BMR95, MAT99] qui utilisent les ensembles d'arbitres ne sont pas tolérants aux fautes dans le sens où un nœud choisit juste un quorum et attend la réponse de tous les membres du quorum. Si n'importe quel membre du quorum choisi tombe en panne, un nœud peut échouer à rassembler les permissions pour entrer en section critique.

Dans [JIA04], **Jiang(1)** a proposé un algorithme distribué de la h-k exclusion mutuelle qui est tolérant aux pannes et utilise la k-coterie. Une k-coterie [HJK93] est une collection d'ensembles (appelées quorums) qui satisfait les propriétés suivantes :

1- Propriété d'intersection : Il y a au moins k paires de quorums disjointes.
2- Propriété de non-intersection : Pour tous h(< k) paire de quorums disjointes Q_1, …Q_h, il existe un quorum Q_{h+1} tel que Q_1, …Q_{h+1} est une paire disjointe.
3- Propriété de minimalité : Tout quorum n'est pas un super-ensemble d'un autre quorum.

Dans l'algorithme de Jiang, un nœud u demandant l'accès à h ressources doit arbitrairement choisir h paires disjointes de quorums et envoie les messages de requête à tous les membres des h paires de quorums. En recevant un message requête, un nœud v accorde sa permission en répondant par un message d'accord. Le nœud u peut entrer en section critique après avoir rassembler les permissions de tous les membres des h paires de quorums disjointes. La sûreté de la h-k exclusion mutuelle est garantie puisque chaque nœud accorde sa permission seulement à un nœud à la fois et il y a au plus k paires disjointes de quorums. L'algorithme de Jiang est tolérant aux fautes dans le sens où un nœud peut rechoisir h paires disjointes de quorums pour leur envoyer d'autres messages de requête quand le nœud ne

rassemble pas assez de permissions après un timeout. Cependant, l'algorithme de Jiang a les problèmes suivants : premièrement, il ne spécifie pas explicitement comment choisir et rechoisir efficacement les h paires de quorums disjointes. Deuxièmement, il est difficile de déterminer la valeur du timeout. De plus, il utilise la k-coterie, et l'inconvénient de la k-coterie est que la taille du quorum devient grande lorsque le nombre de ressources augmente. La complexité de messages de l'algorithme de Jiang est entre 3h.q et 6e.n, où q est la taille du quorum de la k-coterie et $0<e \leq 1$.

Dans [JIA04a], **Jiang(2)** a proposé un autre algorithme qui est basé sur la k-coterie de cohortes définit dans [JHK97], qui est construite à l'aide des structures de cohortes. Une structure de cohortes Coh(k,m)=(C1,…Cm), m>=k, est une liste d'ensembles, où chaque ensemble Ci est appelé une cohorte.
La structure de cohortes Coh(k,m) doit satisfaire les trois propriétés suivantes :

1- $|C1| = k$
2- (quelque soit) i : $1 \leq i \leq m$: $|Ci| > 2k-2$, pour $k>1$
   $|Ci| > 1$, pour $k=1$
3- (quelque soit) i,j : $1 \leq i,j \leq m$, $i \neq j$, $Ci \cap Cj = \emptyset$

En d'autre termes, une structure de cohortes Coh(k,m) a m paires de cohortes disjointes avec la première cohorte qui a k membres et les autres cohortes qui ont plus que 2k-2 membres (ou plus qu'un membre quand k=1).

Cet algorithme n'utilise pas de mécanisme de timeout et n'exige pas aux nœuds de rechoisir *h* paires de quorums disjoints. Le cœur de l'algorithme est une procédure de collection de permissions, appelée Get_Quorum, pour un système distribué avec n nœuds organisés comme structure de cohortes $Coh(k,m) \equiv (C_1,\ldots,C_m)$, le nœud demandant h parmi k ressources doit invoquer $Get\_Quorum(h,k,(C_1,\ldots,C_m))$. Le nœud peut accéder à *h* ressources après les *returns* de Get_Quorum.

La procédure Get_Quorum n'utilise pas de mécanisme de timeout, et peut retourner efficacement un ensemble de nœuds de h paires de quorums disjoints. Il est clair que deux nœuds ne peuvent pas rassembler simultanément les permissions de h_1 et h_2 paires de quorums distincts en invoquant Get_Quorum si $h_1+h_2 > k$. Donc l'algorithme de Jiang garanti la propriété de sûreté de la h-k exclusion mutuelle. Pour assurer la propriété de vivacité, (c-à-d : interblocage et famine), l'algorithme utilise le mécanisme de résolution de conflits de l'algorithme de Maekawa [MAE85].

Cet algorithme est résistant aux pannes de nœuds et/ou des liens de communication et a un coût constant de messages dans le meilleur cas. Cependant, il génère beaucoup de messages dans le mauvais cas. Dans le meilleur cas, un nœud a besoin de 3c*h messages pour accéder à h ressources, où c>2k-2. Dans le mauvais cas, il a besoin de 6f*n messages, où $0<f \leq 1$.Cet algorithme garantit la plus haute disponibilité parmi tous les algorithmes qui utilisent les k-coteries puisque la coterie de cohortes est non-dominée (ND).

3.3 La h-k exclusion mutuelle dans les réseaux mobiles ad hoc

Nous allons présenter le scénario du problème de la h-k-exclusion mutuelle dans les réseaux ad hoc. Considérons un réseau mobile ad hoc qui se compose de *n* nœuds et *k* ressources partagées. Les nœuds sont supposés faire un cycle à travers une section non critique (***NCS***), une section d'entrée (***ES***), et une section critique (***CS***). Un nœud *i* peut

accéder à la ressource partagée seulement à l'intérieur de la section critique. Chaque fois qu'un nœud *i* souhaite accéder à *h*, $1 \leq h \geq k$, ressources partagées, le nœud *i* se déplace de sa section non critique à une section d'entrée, en attendant d'entrer en section critique.

3.3.1 Algorithme de J-R Jiang

Comme nous l'avons vu précédemment, il y a plusieurs travaux [RAY91a, BAL94, MBR98, MAN99, …] proposés pour résoudre le problème de la h-k exclusion mutuelle dans les systèmes distribués. Tous ces travaux utilisent le concept de coterie est de quorum, les quorums sont des collections d'ensembles de nœuds. Un nœud doit contacter tous les membres de son quorum pour accéder à la ressource partagée. Les algorithmes utilisant le concept de structure de quorum sont généralement tolérants aux pannes et ils ont toujours un coût de communication très réduit. Cependant, le concept de structure de quorum n'est pas du tout approprié pour les réseaux mobiles ad hoc puisque la topologie du réseau mobile change continuellement et ainsi, il est très difficile de contacter tous les membres d'un quorum.

Dans les réseaux mobiles ad hoc, l'algorithme proposé par J-R Jiang essaye de résoudre le problème de la h-k-exclusion mutuelle par adaptation de l'algorithme de l'exclusion mutuelle RL (Reverse Link) [WWV98] basé sur le jeton vu précédemment. L'algorithme proposé par J-R Jiang est supposé s'exécuter dans un système composé de n nœuds mobiles et de k ressources partagées. Les nœuds sont étiquetés par *0, 1,..., n-1*. On suppose l'existence d'un jeton unique détenu initialement par le nœud *0*. Cet algorithme a les mêmes hypothèses que celles de l'algorithme RL.

3.3.1.1 Structures de données

Les variables utilisées dans cet algorithme pour le nœud i sont listées ci-dessous :

- *State* : Indique si le nœud *i* est en *ES*, *CS* ou *NCS*. Initialement, *state=NCS*.
- *N* : L'ensemble de tous les nœuds (voisins) qui ont un contact sans fil direct avec le nœud *i*. Initialement, *N* contient touts les nœuds voisins du nœud *i*.
- *Height* : Un triplet (h_1, h_2, *i*) qui représente la taille du nœud *i*. Les liens sont dirigés depuis les nœuds de plus haute taille vers les nœuds de taille inférieure, basés sur l'ordre lexicographique. Par exemple, si la taille du nœud 1, $height_1$, est (2, 3, 1) et la taille du nœud 2, $height_2$, est (2, 2, 2), alors $height_1 > height_2$ et le lien sera dirigé depuis le nœud 1 au nœud 2. Initialement, $height_0$ = (0, 0, 0), et $height_j$, $j \neq 0$, est initialisé de sorte que les liens sont dirigés selon un DAG où chaque nœud a une trajectoire dirigée vers le nœud 0.
- *hVector* : Un tableau de triplets représentants la vue qu'a le nœud *i* de la taille du nœud *j*, $j \in N$. Initialement, *hVector*[j]=taille du nœud *j*.
- *next* : Indique l'emplacement du jeton depuis le point de vue du nœud *i*. Quand le nœud *i* possède le jeton, *next=i*, sinon *next* est le nœud sur un lien sortant. Initialement, *next*=0 si *i =0*, autrement *next* est un voisin sortant.
- *tokenHolder* : Une variable booléenne qui indique si le nœud *i* possède le jeton ou non.
- *Q* : Une file qui contient les requêtes des voisins. Initialement, Q=∅.

Ce qui suit sont les messages utilisés dans l'algorithme :

- REQUEST : Lorsque *i* souhaite entrer en *CS* pour accéder à *r* copies des ressources, il envoie REQUEST au nœud voisin indiqué par *next*.
- RELEASE(*s*) : Lorsque *i* quitte la *CS* pour libérer *s* copies des ressources, il envoie RELEASE(*s*) à un nœud voisin.

- TOKEN(t) : Un message unique qui représente le privilège d'accès aux ressources. Le champ de donnée t, $0 \leq t \leq k$, du message indique le nombre de ressources disponibles.
- LINK : Un message utilisé par les nœuds afin d'échanger la valeur de leur taille avec leurs voisins.

Comme l'algorithme RL, l'algorithme proposé par J-R Jiang est dirigé par les événements. Un événement au nœud i consiste en la réception d'un message d'un autre nœud, ou une indication d'une défaillance de lien ou une formation de lien, etc. Chaque événement déclenche une procédure qui est supposée être exécutée d'une manière atomique.

3.3.1.2 Fonctionnement

- Demander r copies de ressources : lorsque le nœud i demande l'entrée en *CS* pour accéder à r ressources, il enfile son identificateur dans Q et met state à *ES*. Si le nœud i ne possède pas le jeton et i est le seul élément dans la file, il appel *forwardRequest()* pour envoyer le message REQUEST au nœud voisin indiqué par *next*. Si le nœud i possède TOKEN(t), alors i voit si $t \geq r$. Si oui, i met $t = t\text{-}r$, supprime i de Q et met state à *CS* pour accéder à r ressources. Autrement, si $t<r$, alors le nœud i continu à attendre que la condition $t \geq r$ soit vraie pour entrer en *CS*.
- Libérer s copies de ressources : Lorsque le nœud i quitte la *CS* pour libérer s ressources, il met *state=NCS*. Si le nœud i ne possède pas le jeton, il appel *sendRelease(s)* pour envoyer le message *RELEASE(s)* à son voisin indiqué par *next*. D'un autre coté, si i possède le jeton TOKEN(t), i met $t=t+s$.
- Recevoir un message de requête : lorsqu'un message REQUEST est envoyé par un nœud voisin j est reçu par le nœud i, i change *hVector[j]* et enfile j dans Q si le lien entre i et j est entrant vers i. Si i possède le jeton et i n'est pas entrain d'attendre la *CS* (*state*≠*ES*), i appel *giveTokenToFrontOfQ()* pour passer le jeton. Sinon, le nœud i appel *forwardRequest()* si $|Q|=1$ ou si Q n'est pas vide et le lien vers *next* est inversé.
- Recevoir un message de libération : Supposons que le nœud i possède le jeton, alors quand un message RELEASE(s) envoyé par un nœud voisin j est reçu par le nœud i, i met $t=t+s$. Notons que si *state=ES*, cela veut dire que i attend $t \geq r$ (à l'intérieur de la procédure *giveTokenToFrontOfQ()*) pour enter en *CS*, avec r le nombre de ressources demandées par i. Maintenant, si $t \geq r$, le nœud i peut entrer en *CS* ; sinon, le nœud i attend la procédure *giveTokenToFrontOfQ()* pour que la condition $t \geq r$ soit vraie. Supposons que le nœud i ne possède pas le jeton, alors quand le nœud i reçoit un message RELEASE(s), il appel juste *sendRelease(s)* pour transmettre le message de libération.
- Recevoir le message du jeton : Lorsque le nœud i reçoit un message TOKEN(t) de son voisin j, i met *tokenHolder* à vraie. Ensuite, i décrémente sa taille pour être inférieure à celle du dernier possesseur du jeton (c-à-d, le nœud j) et informe tous ses voisins de sa nouvelle taille en envoyant des messages LINK, et appel *giveTokenToFrontOfQ() si* $|Q|>0$.
- Recevoir un message d'information de lien : Lorsqu'un message d'information de lien LINK depuis le nœud j est reçu par le nœud i, j est ajouté dans N et la taille de j est enregistrée dans *hVector[j]*. Si le message de requête de j est dans Q et j est un lien sortant, alors le message de requête de j est enlevé de Q. Si le nœud i n'a pas de liens sortants et il n'est pas le possesseur du jeton, i appel *raiseHeight()* pour qu'un lien sortant soit crée. D'un autre côté, si Q n'est pas vide alors le lien vers *next* est inversé, i appel *forwardRequest()* puisqu'il doit envoyer une autre requête (re-route la requête) pour le jeton.
- Défaillance de lien : Lorsque le nœud i détecte la défaillance d'un lien vers un nœud voisin j, il enlève j de N. Et si un message de requête de j est dans Q, la requête est

supprimée de *Q*. Ensuite, si *i* n'est pas le possesseur du jeton et *i* n'a pas de lien sortant, *i* appel *raiseHeight*(). Si le nœud *i* n'est pas le possesseur du jeton, *Q* n'est pas vide, et le lien vers *next* manque, *i* appel *forwardRequest()* puisqu'il doit envoyer une autre requête (re-route la requête) pour le jeton.

- Formation de lien : Lorsque le nœud *i* détecte un nouveau lien vers le nœud *j*, *i* envoie un message LINK à *j*.

Ci-dessous les procédures appelées par *i* mentionnées précédemment :

- La procédure *giveTokenToFrontOfQ*() : Le nœud *i* défile le premier élément de *Q* et met *next* au premier élément. Si *next=i*, alors *i* vérifie si $t \geq r$, où *t* est le champ du message TOKEN dénotant le nombre de ressources disponibles et *r* dénote le nombre de ressources demandées. Si oui, *i* met *t=t-r* et ensuite entre en section critique. Sinon, *i* attend que la condition $t \geq r$ soit vraie. Après, *i* entre en section critique, si *Q* n'est pas vide alors *i* appel récursivement la procédure *giveTokenToFrontOfQ*() pour passer le message TOKEN au nœud qui est en tête de file. Maintenant, considérons le cas où $next \neq i$. Dans ce cas, *i* décrémente *hVector*[*next*] à (-∞,-∞,*next*), pour que tout message REQUEST entrant soit passé à *next*. Le nœud i met aussi *TokenHolder* à faux, et ensuite envoie un message TOKEN à *next*. Si *Q* n'est pas vide après l'envoie du message TOKEN à next, un message REQUEST est envoyé à *next* immédiatement après le message jeton pour que le jeton retourne éventuellement à *i*.
- La procédure *raiseHeight*() : Appelée par le nœud *i* non détenteur du jeton quand *i* perd son dernier lien sortant ; le nœud *i* augmente sa taille en utilisant la méthode d'inversement de liens de [GAB81] et informe tous ses voisins du changement de sa taille avec les messages LINK. Chaque nœud *l* est supprimé de *Q* si *i* a un lien sortant vers *l*. Si *Q* n'est pas vide à ce point, *forwardRequest*() est appelée puisque *i* doit envoyer une autre requête (déroute la requête) pour le jeton.
- La procédure *forwardRequest*(): Choisit le nœud voisin avec la plus basse taille pour être *next* et envoie un message REQUEST à *next*.
- La procédure *SendRelease*(s) : Un nœud *i* qui ne possède pas le jeton appel *raiseHeight*() quand *i* perd son dernier lien sortant. Après l'appel de *raiseHeight*(), *i* choisie son voisin qui a la plus petite taille pour être *next* et lui envoie un message RELEASE(s). Notons que la procédure *sendRelease(s)* n'est jamais appelée par un nœud détenteur de jeton.

3.3.1.3 Avantages de l'algorithme

1- La structure logique de la topologie du réseau correspond aux liens physiques en traitant les cas de destruction et formation de liens.
2- L'algorithme d'exclusion mutuelle ne dépend pas d'un protocole de routage spécifique.
3- Les nœuds ne contiennent des informations que sur leurs voisins directs.

3.3.1.4 Inconvénients de l'algorithme

1- Il utilise un jeton unique qui collecte les requêtes et doit les satisfaire ; ce qui centralise en quelques sortes le traitement.
2- Le jeton circule constamment tant qu'il y'a des ressources disponibles et il est suivit des nouvelles requêtes ; ce qui engendre plus de messages.
3- A défaut de ressources, le jeton devient stationnaire au niveau d'un nœud qui risque de recevoir plusieurs requêtes ; ce qui engendre autant de transit du jeton vers ce même nœud (concentration à son niveau) et autant de changements de la structure logique (charge supplémentaire en terme de messages et d'énergie) qu'il y a de demandes dans sa file.
4- Un nœud privilégié qui n'est pas en tête de sa propre file envoie le jeton pour le demander une nouvelle fois.

3.4 Tableau récapitulatif

L'étude des algorithmes de la h-k exclusion mutuelle existants a donné naissance à la comparaison présentée dans le tableau 5.

Environnement	Algorithme	Approche	Structure			Robustesse (Tolérance aux pannes)	Performances		Avantages	Inconvénients
			Type	Technique	Politique		NME	Taille Mémoire		
SD	Raynal **1991**	Consensus total		HL	FIFO	Non	Entre 2(n-1) et 3(n-1)		Sa simplicité d'implémentation.	1- Non tolérant aux pannes. 2- Génère beaucoup de messages.
	Baldoni **1994**	Quorum (consensus partiel)	Ensembles d'arbitre	PPL+HL	FIFO	Non	O(q) et q=O($n^{k/k+1}$)		Génère moins de messages que l'algorithme de Raynal.	1- Lorsque k augmente la taille du quorum approche O(n). 2- Non tolérant aux fautes.
	Baldoni, Manabe, Raynal et Aoyagi **1995**	Quorum (consensus partiel)	k-arbitre	PPL+HL	FIFO	Non	Entre 3q et 3(h+3)q	Chaque processus maintient une liste de priorité 2^x	Génère moins de messages que l'algorithme de Baldoni.	1- La taille des quorums reste beaucoup plus grande que celle des quorums de la 1-coterie, donc l'algorithme a besoin de beaucoup de messages. 2- Non tolérant aux fautes.
	Kakugawa et Yamashita **1996**	Quorum (consensus partiel)	Coterie locale	HL + PPL Accès aux noms de ressources	FIFO	Oui Au moins un quorum survit, si une coterie singleton n'est pas utilisée comme coterie locale	$\geq 4\|q\|$ $\leq(7+\|\alpha(u)\|)\|q\|$ $\|\alpha(u)\|=h$	1-BD distribuée de taille 2^R 2- chaque processus a une file de taille indéfinie.	1-La taille du quorum est plus petite que celle d'un k-arbitre. 2-les processus qui ne partagent aucune ressource n'interfère pas avec les autres processus. 3-Tolérant aux pannes.	Dans le mauvais cas, lorsque h est grand, l'algorithme est moins efficace.
	Manabe et Tajima **1999**	Quorum (consensus partiel)	(h,k)-arbitre	PPL+HL	FIFO	Non	$\geq 3\|q_{h,k}\|$ $\leq(3h+3)\|q_{h,k}\|$	Chaque processus a une file de priorité 2^X	1-La taille du quorum dans le (h,k)-arbitre est plus petite que celle d'un k-arbitre.	1-Difficulté d'obtenir un (h,k)-arbitre de taille minimale. 2-difficulté d'obtenir un (h,k)-arbitre non-dominé.
	Kuo **1999**	Quorum (consensus partiel)	(K,m)-coterie nondominée							
	Kuo **2001**	Ce n'est pas un algorithme mais plutôt une nouvelle structure de k-arbitres. Kuo a introduit l'opération de jointure de k-arbitres, qui consiste à construire de nouveaux k-arbitres qui soient meilleures et qui peuvent être non-dominés en joignant des k-arbitres connus.								
	Kuo **2001**	Il s'agit d'une autre opération qu'a introduit Kuo, c'est l'opération de composition de k-arbitres, qui consiste à construire de nouveaux k-arbitres qui soient meilleures et qui peuvent être non-dominés en composant des k-arbitres connus.								

	Kuo **2001**	Quorum (consensus partiel)	(q, k)-arbitres Binomials avec des quorums uniformes							
	Wada et Cheng **2002**	Gestionnaire (manager) de ressource		Serrage +HL	FIFO	Non	O(P), Où P est le nombre de process qui demandent les ressources		1- L'efficacité de l'utilisation des ressources est améliorée. 2- le temps d'attente des processus demandeurs est réduit 3- génère moins de messages.	1- la charge de la distribution des ressources est imposée à un manager de ressource, s'il tombe en panne, l'algorithme s'arrête. 2- Non tolérant aux fautes
	Jiang **2004**	Quorum (consensus partiel)	k-coterie	PPL+HL	FIFO	Oui Resélection de quorums +Mécanisme de Timout	Entre 3h.q et 6e.n, où q est la taille du quorum de la k-coterie utilisée et 0<e≤1		Tolérant aux pannes	1- Il ne spécifie pas explicitement la manière de choisir et rechoisir les h paires de quorums 2- c'est difficile de déterminer la valeur du timeout.
	Jiang **2004**	Quorum (consensus partiel)	Coterie de cohortes	PPL+HL	FIFO	Oui	Entre 3c.h et 6f.n, où c>2k-2 et 0<f≤1		Tolérant aux pannes	Génère beaucoup de message dans le mauvais cas.
Ad hoc	J-R Jiang **2002**	Jeton unique	DAG	Inversement partiel de liens	FIFO	Non			1- La structure logique de la topologie du réseau correspond aux liens physiques en traitant les cas de destruction et formation de liens. 2- Ne dépend pas d'un protocole de routage spécifique. 3- Les nœuds ne contiennent des informations que sur leurs voisins directs.	1- Il suppose que les pannes de nœuds ne se produisent pas. 2-Il suppose que le partitionnement du réseau ne peut pas se produire. 3- Un nœud qui a envoyé une requête et perd son lien vers son voisin *Next*, renvoie la requête vers un autre voisin, ce qui augmente le temps d'attente pour un nœud pour avoir le jeton et peut produire une situation de famine si les changements de la topologie ne s'arrête pas. 4- Un nœud privilégié qui n'est pas en tête de sa propre file envoie le jeton pour le demander une nouvelle fois.

Tableau 5. Comparaison entre les différents algorithmes distribués de la h-k exclusion mutuelle.

3.5 Conclusion

L'exclusion mutuelle simple et la k-exclusion mutuelle ne prennent pas les cas ou le total des ressources peut différer d'une requête à l'autre. Dans cette section, nous avons formalisé ce problème comme le problème de la h-k exclusion mutuelle, dans lequel chaque requête concerne un certain nombre h ($1 \leq h \leq k$) d'unités de ressources partagées et aucune unité n'est allouée à plusieurs processus en même temps. Nous avons aussi présenté presque tous les travaux antérieurs et tous les algorithmes de la h-k exclusion mutuelle publiés jusqu'à présent.

Dans ce chapitre nous avons introduit le problème de la h-k exclusion mutuelle dans les réseaux mobile ad hoc, nous avons présenté l'algorithme proposé par J-R Jiang ; cet algorithme est une adaptation de l'algorithme RL pour le problème d'allocation de h parmi k ressources ; donc, il hérite de ses avantages et de ses inconvénients.

A travers cette étude, on a constaté que les protocoles de la h-k exclusion mutuelle en particulier et d'allocation de ressources dans les réseaux mobiles ad hoc en général, présentés jusque là, utilisent tous l'approche à jeton, ce qui implique l'existence d'une tendance orientée jeton pour ce type de réseaux. En effet, les solutions basées sur les jetons semblent être les plus appropriées pour le problème de la h-k exclusion mutuelle en environnement mobile ad hoc. En outre, l'utilisation des (k-arbitre, coteries de cohortes,…) est à écarter vu le coût de leur construction et maintien (jusqu'à preuve du contraire) qui s'ajoute au coût du service de la h-k exclusion mutuelle lui même.

Partie 2

CONTRIBUTION

Chapitre 4

Conception du protocole

4.1 Introduction

Dans ce chapitre, nous allons présenter un nouvel algorithme de la h-k exclusion mutuelle dans les réseaux mobiles ad hoc. Rappelons que dans l'exclusion mutuelle généralisée du type h parmi k (ou h-k exclusion mutuelle), il existe k ressources identiques et chaque processus peut en demander, à tout instant, un nombre quelconque; de plus, chaque exemplaire de la ressource ne peut être utilisé que par un processus à la fois. L'algorithme de la h-k-exclusion mutuelle doit satisfaire deux propriétés : la sûreté (chaque ressource peut être utilisée par au plus un processus à la fois à un moment donné et la vivacité (toutes les requêtes doivent être satisfaites à un moment ou à un autre). Plusieurs algorithmes de la h-k exclusion mutuelle ont été proposés dans la littérature pour les systèmes distribuées et très peu de travaux ont été réalisés pour la h-k exclusion mutuelle dans les réseaux mobiles ad hoc, dont un seul algorithme est trouvé dans la bibliographie, il s'agit de l'algorithme proposé par J-R Jiang [JIA02].

L'algorithme de J-R Jiang[JIA02] est une adaptation de l'algorithme RL (Reverse Link) [WWV98] de l'exclusion mutuelle basé sur le jeton, cet algorithme présente un certain nombre de caractéristiques. Il utilise un jeton unique. Ce jeton circule constamment tant qu'il y a des ressources disponibles et il est suivit des nouvelles requêtes. À chaque visite du jeton, même temporaire, son nouveau possesseur doit envoyer des messages " LINK " à tous ses voisins pour les informer de sa nouvelle taille. Cela veut dire qu'à chaque déplacement du jeton ou de nœud, la structure logique du réseau change, ce qui génère beaucoup de ré-acheminements des demandes et de libérations de ressources, en plus de la gestion de la valeur de la taille au niveau de chaque nœud. En outre, le jeton devient stationnaire au niveau du premier nœud demandant des ressources non disponibles. Ce nœud risque de recevoir plusieurs demandes avant qu'il soit en mesure de satisfaire sa première demande dans la file. Dans ce cas, le jeton va faire autant de transits vers ce même nœud et il y aura autant de changements de la structure logique du réseau qu'il y a de demandes dans la file, ce qui représente une concentration au niveau d'un même nœud et une charge supplémentaire en terme de messages et d'énergie pour les unités mobiles.

Le protocole que nous proposons adopte une approche différente de celle de J-R Jiang[JIA02]. Il est fondé sur deux principes : *racine* et *distribution*. En effet, notre solution utilise deux structures logiques d'arborescence basées sur la structure physique du réseau mobiles ad hoc baptisées racine et distribution. La première est utilisée pour récolter les requêtes des ressources ; l'autre est utilisée pour satisfaire ces requêtes en distribuant les ressources demandées et en récoltant les ressources libérées, ce qui permet un partage de charges. Ces structures logiques sont organisées selon le principe de niveaux. A un moment donné, le plus bas niveau constitue le sommet de l'arborescence racine ; il contient un seul nœud appelé le nœud racine et chaque nœud dans le réseau a un chemin vers chaque sommet de l'une ou l'autre des deux arborescences.

Chaque arborescence, que ce soit racine ou distribution est dotée d'un jeton qui est maintenu toujours par le nœud qui est au sommet de l'arbre correspondant. Donc, deux jetons sont utilisés, le jeton *racine* au sommet de l'arborescence racine et le jeton *clé* au sommet de l'arborescence distribution. Le nœud qui détient le jeton racine se charge de récolter les demandes de ressources ; le nœud qui détient le jeton clé se charge de distribuer les ressources demandées et de récupérer les ressources libérées. Les autres nœuds se chargent de transmettre les messages dans l'arborescence correspondante.

Etant donné le changement fréquent de la topologie physique, les structures logiques utilisées doivent être constamment mises à jour. Pour se faire, nous faisons recours à la technique d'inversement partiel de liens logiques pour garantir la continuité et la sûreté du protocole.

Les structures sur laquelle se base notre protocole sont similaires à celle du DAG (*Directed Acyclic Graph*) mais diffèrent sur le concept de poids affecté à chaque nœud. Ces structures permettent de s'adapter aux changements de la topologie du réseau et de minimiser le nombre de messages échangés pour transmettre le ou les jetons. L'utilisation des deux principes racine et distribution permet d'éviter le problème de centralisation, de réduire considérablement les déplacements de jetons tout en favorisant la distribution des ressources et par conséquent favoriser la réduction du nombre de messages échangés, ainsi que le délai de synchronisation et le temps d'attente des nœuds demandeurs. L'utilisation des deux jetons permet d'équilibrer la charge de distribution des ressources. Le protocole ne fait pas recours aux services de la couche de routage pour véhiculer les messages. Les requêtes sont servies selon leur ordre d'arrivée.

Dans ce qui suit, nous présentons notre solution en suivant les étapes suivantes :

En premier lieu et dans la section 2, nous décrivons le modèle du système objet de la conception. Nous présentons dans la section 3, l'approche globale du protocole selon laquelle on introduit globalement ses bases. La section 4 présente les détails sur les bases du protocole. Cette section présente plusieurs exemples qui expliquent au mieux les différents principes, techniques et mécanismes utilisés. Dans la section 5, le protocole sera présenté dans le détail en précisant les structures de données implémentées ainsi que les actions entreprises à chaque événement. Le texte de ces actions est également donné. Dans la section 6, une discussion sur les différents points cachés du protocole et les choix adoptés permettra de préciser certains éléments du protocole est donnée. Enfin, quelques directives liées à l'extension du protocole sont présentées. Ce chapitre est terminé par une conclusion.

4.2 Modèle du système

Nous considérons un système composé de *n* unités mobiles communicantes entre elles par envois de messages en utilisant des interfaces de communication sans fil. Chaque unité mobile exécute un processus d'application et un processus de la h-k exclusion mutuelle. Les primitives s'exécutent de manière atomique ; cependant, pendant l'exécution de la section critique les événements qui surviennent peuvent être traités mais un à la fois. Les hypothèses imposées sur les nœuds et sur le réseau par le problème traité sont :

1- Le système contient *n* nœuds mobiles et chaque nœud a un identificateur unique.
2- Le système contient k ressources (k valeur connue par tous les nœuds et $k > 0$).
3- Un nœud peut demander h ressources tel que $h \leq k$.
4- Un nœud ne peut pas lancer plus d'une requête à la fois.
5- Les liens de communication sont fiables, bidirectionnels et FIFO.
6- Aucun algorithme de routage n'est utilisé.
7- La couche de liaison (MAC) assure que chaque nœud connaît ses voisins en l'informant de la création ou la défaillance d'un lien.
8- Un nœud ne peut pas rester dans la section critique indéfiniment.
9- Pas de partitionnement permanent du réseau.
10- Pas de pannes de nœuds.

4.2.1 Les différents messages échangés avec le processus d'application

L'interface de communication entre le service de la h-k exclusion mutuelle et l'application est définie par un certain nombre de messages qui permettent d'enclencher des traitements de part et d'autres de ces deux correspondants. Ces messages sont :

Les messages d'application
RequestCS$_i$: Le processus d'application demande d'entrer dans sa section critique.
ReleaseCS$_i$: Le processus d'application quitte sa section critique.
Les messages du protocole de la h-k exclusion mutuelle
EnterCS$_i$: Le processus d'exclusion mutuelle informe le processus d'application qu'il peut entrer dans sa section critique.

4.3 Approche globale du protocole

L'algorithme se base sur des *structures logiques* d'*arborescences par niveaux* ; dans une arborescence donnée, le niveau le plus bas, initialement le niveau "0", contient un seul nœud qui est le nœud racine. Le niveau suivant "1" comporte un ensemble de nœuds qui ont un lien physique avec le nœud racine ; c-à-d, il contient tous les nœuds voisins du nœud racine (qui sont dans sa portée de transmission). Le niveau "2" est composé des nœuds qui ont au moins un voisin dans le niveau "1" et aucun voisin dans le niveau "0". Le niveau "3" contient les nœuds qui ont au moins un voisin au niveau "2" et aucun voisin dans les niveaux inférieurs à "2". Tous les niveaux sont définis de la sorte jusqu'à ce qu'on arrive au dernier niveau qui est constitué de nœuds qui représentent les feuilles de l'arborescence. Par définition, le nœud racine se situe dans le niveau le plus bas.

Tous les niveaux de la structure logique sont initialement adjacents mais au cours de l'exécution de l'algorithme, il se peut que les niveaux ne soient pas adjacents ; néanmoins, ils restent dans l'ordre décroissant du plus grand niveau jusqu'au plus petit.

Afin de définir formellement les niveaux associés aux structures d'arborescences, nous introduisons la relation de voisinage notée « $\leftrightarrow$ » tel que $x \leftrightarrow y$ signifie que x est voisin de y. Soit L, l'ensemble de couples de nœuds défini comme suit :

$L=\{(x, y) / x \leftrightarrow y\}$

La structure d'arborescence est définie par l'ensemble des (n+1) niveaux définis comme suit :

$Niv_0=\{x_0\}$
$Niv_1=\{x/(x,x_0) \in L\}$
$Niv_2=\{x/\exists y \in Niv_1, (x,y) \in L \ \& \ (x,x_0) \notin L\}$
.
.
.
$Niv_n=\{x/\exists y \in Niv_{n-1}, (x,y) \in L \ \& \ \forall m<n-1, \neg \exists z \in Niv_m/(x,z) \in L \}$; n et m sont des entiers.

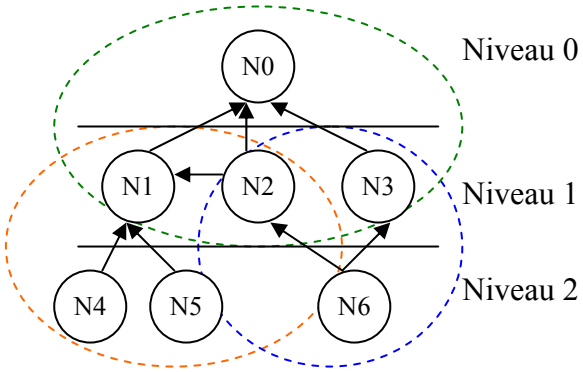


Figure 10 : Structure logique d'arborescence par niveaux.

Dans la figure10, tous les nœuds qui sont dans un même ensemble (représenté par un cercle en pointillé) ont un lien physique entre eux. Le niveau "0 " contient uniquement le nœud *N0*, le niveau "1 "contient les nœuds *N1*, *N2* et *N3* car ils ont un lien physique avec *N0*. Les nœuds *N4*, *N5* et *N6* appartiennent au niveau "3" car ils sont dans la portée de transmission des nœuds du niveau "2 ".

L'algorithme se base aussi sur deux concepts de jeton, *jeton racine* et *jeton clé*. Ainsi, à chacun est associée une structure d'arborescence, à savoir respectivement, *l'arborescence racine* et *l'arborescence distribution*. Dans l'arborescence racine, le nœud qui possède le jeton racine doit être forcément au niveau 0, c'est lui le nœud racine. Dans l'arborescence distribution, le nœud qui détient le jeton clé est aussi au niveau 0 et détient l'ensemble des ressources et se charge de les distribuer. Initialement, le nœud qui détient le jeton racine détient aussi le jeton clé ; par conséquent les deux structures d'arborescences sont confondues.

Donc, les différents messages circulants entre les nœuds suivent selon leur nature l'une ou l'autre arborescence. Les messages de demandes et d'allocation de ressources circulent dans l'arborescence racine alors que les messages de libération de ressources circulent dans l'arborescence distribution. Dans l'algorithme, des messages d'informations liés aux changements de la structure logique sont aussi utilisés, mais ceux là font abstraction des deux arborescences.

Chaque nœud dans le réseau hormis le nœud racine a un *lien logique* (virtuel) sortant correspondant au *lien physique*, dirigé vers un ou plusieurs nœuds du niveau inférieur ; De ce fait, tous les nœuds disposent d'un chemin dirigé vers le nœud racine. De même, chaque nœud dans le réseau hormis le nœud distributeur a un lien logique (virtuel) sortant, correspondant au lien physique, dirigé vers un ou plusieurs nœuds voisins. Ces liens suivent des chemins de telle sorte qu'ils mènent tous vers le nœud distributeur de ressources. Ainsi, chaque nœud entretient aussi des liens entrants pour chacune des deux arborescences.

Initialement, les liens dirigés vers le nœud racine et les liens dirigés vers le nœud distributeur sont identiques étant donné qu'il s'agit d'un même nœud qui détient les deux jetons. Ainsi, l'*arborescence racine* et l'*arborescence relative à la distribution* sont confondues en une seule.

Aussi, chaque nœud dans le réseau maintient une *file d'attente* contenant les identificateurs des nœuds voisins du niveau supérieur, qui ont envoyés des demandes de ressources, ainsi que le nombre de ressources demandées.

Notre algorithme fait recours à la technique d'*inversement partielle de liens (Partial Reversal technique)* lorsqu'un nœud perd son dernier lien sortant suite à une rupture de liens ou lors de la circulation des jetons. Le but de cette technique est de garantir l'existence permanente d'un chemin vers le nœud qui possède le jeton (que ce soit racine ou clé).

4.4 Principe du protocole

4.4.1 Structure par niveaux

Le principe de niveaux permet une organisation logique des nœuds en sous-ensembles. Cette organisation logique suit systématiquement la topologie physique du réseau.

Chaque niveau de l'arborescence est désigné par une valeur entière, elle est affectée comme attribue à chacun des nœuds de ce niveau ; initialement, cette valeur représente le nombre de sauts effectués pour atteindre la racine, par exemple (si la valeur du niveau est 3, alors le nombre exacte de sauts exigés pour que le message arrive au nœud racine est 3). Ultérieurement, cette valeur peut ne pas représenter exactement le nombre de sauts effectués pour atteindre la racine, mais elle est utilisée pour calculer le nombre de sauts à un moment donné. Le nombre de sauts permet de connaître la complexité de messages du protocole.

Soit *P* l'ensemble de tous les nœuds du réseau, alors :

1- Le niveau *l* (Initialement *l* = "0"), tel que *l* est la plus petite valeur de niveau à un moment donné, contient un sous-ensemble de *P*. Ce sous-ensemble comporte un seul élément qui est le nœud racine de l'arborescence.

2- On dit que le nœud *i* appartient au niveau *l+1*, si *l* est le plus bas niveau (dans l'arbre) avec lequel *i* a un lien physique.

3- Les différents nœuds d'un niveau *l* peuvent avoir des liens physiques les uns avec les autres. Ces liens restent symboliques tant que le réseau est dans un état stationnaire et ils ne sont pas considérés dans les différentes étapes du fonctionnement de l'algorithme. Néanmoins, il est très important de les garder dans la structuration de base du réseau logique car en cas de changement de la topologie du réseau, suite à des ruptures de liens ou d'un déplacement de jeton, ils peuvent devenir des liens reliant des nœuds de niveaux adjacents après avoir été des liens entre des nœuds d'un même niveau. Ces liens logiques sont bidirectionnels et chaque nœud regroupe tous ses nœuds voisins qui sont dans le même niveau que lui dans un seul ensemble.

Le rôle du mécanisme de niveaux est à peu près semblable au rôle du mécanisme de taille ou de poids décrit dans [GAB81], à la différence qu'une taille prend en charge trois valeurs, deux d'entre elles ne représentent que l'ordonnancement d'un nœud par rapport à son voisin, alors qu'un niveau a une valeur unique qui représente à la fois l'ordonnancement des nœuds et le nombre de sauts vers la racine.

La construction de la structure logique d'arborescence par niveaux est amorcée par le nœud "0" ; ce dernier envoie des messages d'initialisation à ses voisins. Ce message d'initialisation contient des informations sur l'identité et le niveau du nœud qui l'a envoyé. Ainsi, en recevant ce message, les voisins du nœud "0" qui est au niveau "0" deviennent au

niveau "1". Ils envoient à leur tour des messages de même type à leurs voisins qui incrémentent ou décrémentent leurs niveaux suivant les niveaux de leurs voisins et ainsi de suite, le message d'initialisation parcours tout le réseau et tous les nœuds se voient appartenir à un niveau. Il se peut qu'un nœud reçoive plusieurs messages d'initialisation avec différentes valeurs de niveaux, ce nœud, choisit toujours d'appartenir au niveau le plus petit (le plus proche du nœud racine). Lorsqu'un nœud acquière un niveau plus petit que le sien encours, il envoie un autre message d'initialisation à ses voisins afin qu'ils améliorent eux aussi leurs positions vis-à-vis de la racine.

Lorsque tous les niveaux sont établis, les feuilles de l'arborescence (les nœuds appartenant au plus haut niveau) envoient leurs accusés de réception à leurs voisins du niveau inférieur. Ces accusés de réception descendent d'un niveau à l'autre jusqu'à atteindre la racine. Ensuite, le nœud "0" envoie un message de terminaison de la construction de la structure logique qui sera diffusé dans tout le réseau.

Pour des raisons de simplicité, nous admettons que le réseau reste statique durant la construction de l'arborescence.

Exemple 1 : Construction et initialisation d'une arborescence logique.

Supposons un réseau mobile ad hoc (figure 11), composé de sept (07) nœuds mobiles et cinq (05) ressources partagées entre eux.

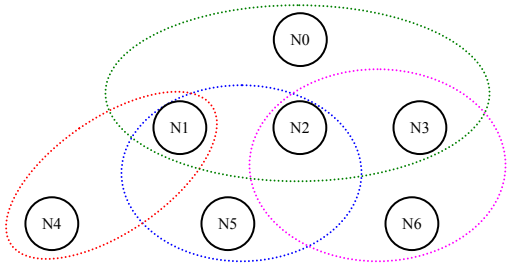


Figure 11 : Le réseau ad hoc initial.

Dans la figure 11, tous les nœuds qui sont dans un même ensemble (représenté par un cercle en pointillés) ont un lien physique entre eux. Le nœud *N0* commence la construction de la structure logique en envoyant des messages d'initialisation à ses voisins directs *N1*, *N2* et *N3*. À la réception de ces messages, les nœuds *N1*, *N2* et *N3* mettent à jour leurs niveaux et envoient à leur tour respectivement des messages d'initialisation respectivement à *N2, N4* et *N5* voisins de *N1*, à *N1*, *N3, N5* et *N6* voisins de *N2* et à *N2, N6* voisin de *N3*. Lorsque les nœuds *N4*, *N5* et *N6* reçoivent les messages d'initialisation de tous leur voisins et voient qu'ils n'ont pas de voisins à un niveau supérieur, ils seront considérés comme étant des feuilles dans les deux arborescences initiales (racine et distribution). Par conséquent, ils envoient leurs accusés de réception à leurs voisins du niveau inférieur. De même, en recevant tous les accusés de réception des voisins du niveau supérieur, les nœuds *N1*, *N2* et *N3* envoient leurs accusés de réception à *N0*. Quand *N0* collecte tous les accusés de réception de ses voisins, il diffuse un message de terminaison de l'initialisation de la structure. La figure 12 donne la structure construite :

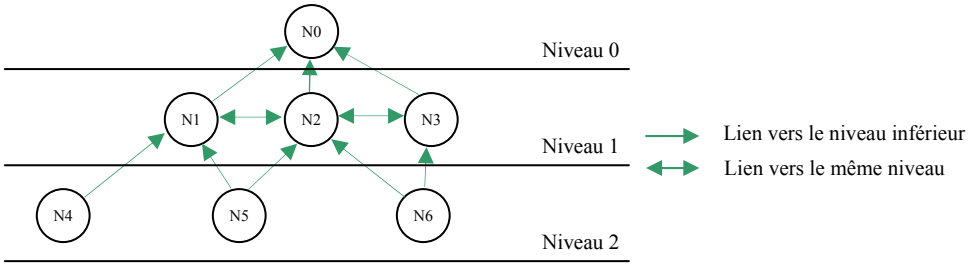


Figure 12 : Le réseau ad hoc après initialisation.

Dans la figure 12, afin de conserver la clarté des schémas représentant l'arborescence enracinée du réseau, on a préféré omettre les flèches des liens physiques, étant donné qu'ils sont confondus avec les liens logiques de cette même arborescence.

4.4.2 La file d'attente Q_i local à P_i

Rappelons que les requêtes qui arrivent au niveau d'un nœud sont celles des nœuds voisins appartenant à un niveau supérieur ; elles doivent être acheminées jusqu'à ce qu'elles arrivent au nœud racine. Pour que ces requêtes soient satisfaites, le chemin du retour vers les demandeurs de ressources doit être sauvegardé. Pour ce faire, nous avons doté chaque nœud d'une file d'attente, dans laquelle, il sauvegarde les requêtes dans l'ordre de leurs arrivées.

La file Q_i est constituée d'un ensemble d'éléments (P_j , r_j) où P_j est l'identité du nœud qui a demandé l'accès à la section critique et dont la requête est reçue au niveau de P_i et r_j le nombre de ressources demandées par le nœud P_j. Les éléments (P_j , r_j) sont ordonnés suivant une politique *FIFO* ; cette politique garantit qu'aucune requête n'est satisfaite avant que toutes les requêtes qui la précèdent ne le soient.

Exemple 2 : Mise à jour des files d'attentes.

La figure 13 reprend les mêmes structures que ceux de la figure 12. Dans la figure 13, le nœud *N6* lance une requête à *N0* pour avoir deux ressources via *N3*, puis *N4* qui veut accéder à trois ressources, envoie une première requête à *N0* via *N1* demandant les 3 ressources, puis *N5* lance une requête toujours à *N0* via *N1* demandant 4 ressources. Les files d'attentes respectives aux niveaux des nœuds concernés sont mises à jour. On aura alors le schéma suivant décrit dans la figure 13 :

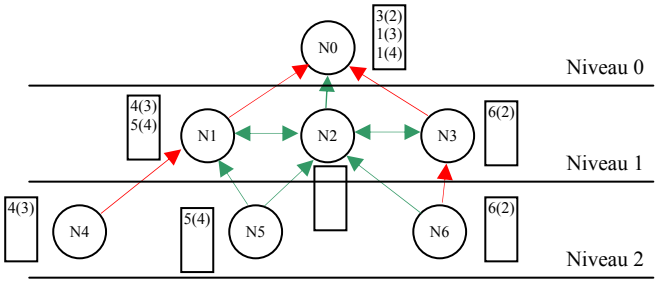


Figure 13 : Mise à jour des files d'attentes des requêtes.

4.4.3 Le concept de deux jetons

Le choix de deux jetons est motivé par un certain nombre d'avantages :

- L'utilisation de deux jetons permet d'éviter certains problèmes tels que la centralisation ie, la collecte des requêtes et la distribution des ressources peuvent se faire à deux endroits différents.
- Avec deux jetons, on peut garantir une certaine tolérance aux fautes en cas de perte d'un des jetons (celui qui possède le deuxième jeton génère le jeton perdu). (Ce cas n'est pas pris en considération dans la solution actuelle).
- Ce choix favorise aussi la distribution des ressources. En effet dans le cas d'un seul jeton, le nœud qui le détient doit recevoir les requêtes et doit les satisfaire. Quand ce dernier sort de sa section critique, il doit envoyer le jeton au nœud qui est en tête de file et lui envoie aussi la file des requêtes restantes ce qui induit une complexité additionnelle en terme de nombre de messages. Par contre, l'utilisation de deux jetons va permettre d'éviter de faire suivre le jeton de messages de requêtes et d'équilibrer la charge de distribution des ressources. En effet, pour ce dernier point, un nœud qui détient le jeton racine se charge de récolter les demandes de ressources et un nœud distributeur qui détient le jeton clé se charge de distribuer les ressources et satisfait ainsi les demandes.

Au cours du fonctionnement du protocole, le nœud racine garde les deux jetons jusqu'à ce qu'il sorte de sa section critique (sauf dans le cas initial où ce nœud "0" n'est pas en section critique auquel cas, il envoie les deux jetons dès la réception d'une première requête). Lorsqu'il sort de sa section critique, il réagit selon les cas suivants :

1- Si sa file est vide, il garde seulement les deux jetons à l'état de repos ;
2- Si sa file contient une seule demande, alors il envoie au nœud correspondant à celle-ci les deux jetons ;
3- Si sa file contient au moins deux demandes, alors, il envoie le jeton racine au nœud situé en fin de file, celui-ci devient le nouveau nœud racine. Par conséquent, l'arborescence enracinée du réseau change et les demandes futures de ressources seront alors envoyées au nouveau nœud racine à travers cette nouvelle arborescence. Par contre, le jeton clé est gardé localement afin de satisfaire les demandes restantes, ce qui maintient l'arborescence distribution inchangée. Dès qu'il reste uniquement la demande du dernier nœud de sa file, le jeton clé lui est envoyé ; ce qui va lui permettre de rentrer à sa section critique. Il s'agit du nœud auquel le jeton racine a été déjà envoyé. De ce fait, la réception du jeton clé va lui permettre de satisfaire les requêtes déjà récoltées (éventuellement en attente).

La satisfaction d'une requête consiste à envoyer immédiatement au nœud correspondant les ressources nécessaires, s'il y a lieu. Autrement, la satisfaction est reportée au moment de réception d'assez de ressources libérées. Notons que plusieurs requêtes peuvent être satisfaites simultanément.

Il ressort que les messages de demandes suivent les chemins menant au détenteur du jeton racine et les messages de libération de ressources suivent les chemins menant au détenteur du jeton clé.

Par défaut, le nœud distributeur satisfait les demandes de ressources en envoyant le nombre de ressources demandées séparément pour chaque requête. Une optimisation possible est d'envoyer le cumul du nombre de ressources demandées afin de satisfaire plus d'une demande à la fois si le nombre de ressources disponibles le permet. Par exemple, si un nœud

envoie une première requête demandant k_1 ressources qui n'étaient pas encore disponibles, puis le même nœud envoie une autre requête demandant k_2 ressources ; dans ce cas, si par la suite plus que k_1+k_2 ressources sont disponibles, alors, le nœud distributeur lui envoie k_1+k_2 ressources à la fois.

Exemple 3 : Circulation de jetons.

Dans la figure 14, nous reprenons l'exemple lié à la figure 13. Le nœud *N0* détient les deux jetons. Dès la réception de la requête émise par *N6*, *N0* lui envoie les deux jetons et inverse ses liens vers *N3*. En recevant les deux jetons, *N3* décrémente sa valeur de niveau, envoie les deux jetons à *N6* et inverse ses liens vers ses voisins. *N6* fait la même chose que *N3* sauf qu'il garde les deux jetons et entre en section critique.

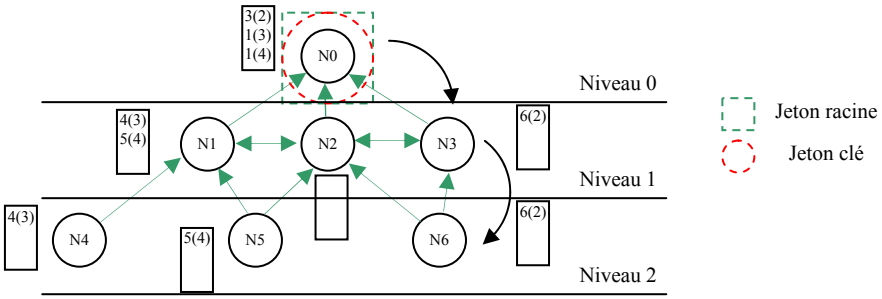


Figure 14 : Envoie des deux jetons.

4.4.4 Gestion des liens logiques entre les nœuds

4.4.4.1 Les chemins vers les détenteurs des jetons

Rappelons que chaque nœud dans le réseau maintient un chemin vers le nœud détenant le jeton racine et un chemin vers le nœud détenant le jeton clé. Ces deux chemins peuvent se confondre en un seul ; notamment dans le cas initial. Le long d'un chemin, les niveaux des nœuds sont décroissants. La figure 15 illustre les deux arborescences racine et distribution qui sont confondues :

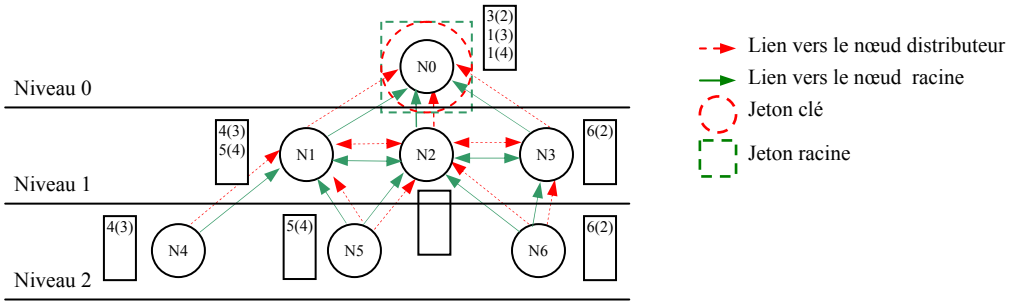


Figure 15 : Les deux arborescences racine et distribution.

Lors de l'envoi de requêtes, des critères de choix des chemins peuvent être définis pour améliorer et optimiser l'algorithme. Par exemple, lorsqu'un nœud a le choix entre deux nœuds voisins qui sont dans le même niveau, il sélectionne celui qui n'a pas la file vide car la probabilité que ce nœud devienne le nœud racine est supérieure à celle d'un nœud qui a la file vide. Autrement, il l'envoie à celui qui a le plus grand nombre de voisins au niveau supérieur ; dans ce cas, aussi, la probabilité que ce nœud devienne le nœud racine est supérieure à celle d'un nœud qui n'a que deux voisins. Afin d'éviter les échanges de messages pour avoir l'information concernant le nombre de voisins d'un nœud ou le nombre d'éléments de sa file, un nœud peut en échange envoyer sa requête à l'un de ses voisins à tour de rôle ou simplement il l'envoie à l'un d'eux aléatoirement ; celui-ci est considéré comme prochain sur le chemin du jeton.

Un nœud désirant entrer en section critique envoie une requête sur un lien sortant sélectionné ; chaque nœud recevant cette requête fait de même jusqu'à ce qu'elle arrive au nœud racine. A la réception d'une requête, chaque nœud va mettre dans sa file de requêtes l'identificateur du nœud qui a transmis cette requête ainsi que le nombre de copies demandées. La figure 6 illustre un arbre enraciné pour un système composé de sept nœuds ; si le nœud *N6* envoie une requête, alors elle va traverser le chemin *N6-N3-N0* ou le chemin *N6-N2-N0*.

4.4.4.2 Inversement de liens suite au déplacement de jetons

Lorsque le nœud racine envoie les deux jetons, il doit inverser ses liens (correspondants à chacune des deux arborescences) vers le nœud dont il vient de lui transmettre les jetons étant donné qu'il doit garder un chemin vers la nouvelle racine. En recevant les deux jetons, un nœud doit avoir un niveau plus bas que celui de l'émetteur de ces jetons. Il diminue donc son niveau et inverse ses liens en informant certains de ses voisins pour leur permettre de s'adapter à cette nouvelle situation. Chaque nœud intermédiaire recevant ces jetons fait de même jusqu'au nouveau nœud racine qui doit faire en sorte qu'il soit un nœud puit (ie sans liens sortants)

La figure 15 montre la topologie initiale dans laquelle le nœud *N0* possède les deux jetons. Quand il reçoit la demande de *N6*, le nœud *N0* lui envoie les deux jetons.
Les deux jetons passent par le nœud *N3* ; en recevant ces jetons, *N3* diminue son niveau à "-1" et lorsque le nœud *N6* reçoit les deux jetons, il diminue son niveau à "-2". La figure 16 montre l'état du réseau après réorganisation suite au déplacement des deux jetons. Dans cet état, *N6* devient le nœud racine.

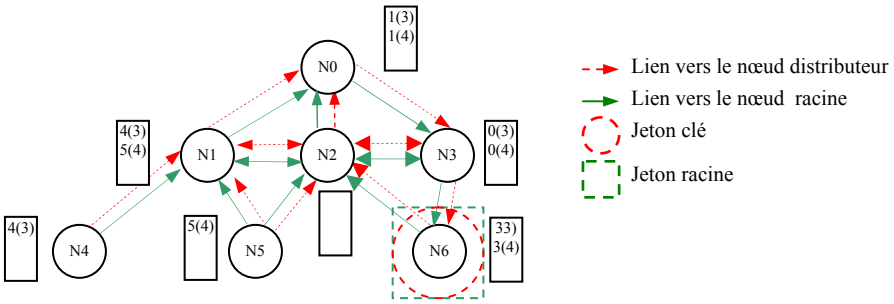


Figure 16 : Réorganisation des arborescences.

Lors du déplacement d'un ou des deux jetons, l'arborescence correspondante change ; des inversement des liens vers le nouveau détenteur d'un ou des deux jetons auront lieu. De plus, il y aura la réorganisation de la structure qui se fait au fur et à mesure afin de conserver la cohérence des liens entre les nœuds et de respecter le principe que les nœuds pointent sur leurs voisins de niveaux inférieurs. En plus, cette réorganisation permet d'optimiser les chemins vers la nouvelle racine ou le nouveau distributeur. Le réarrangement après le déplacement des deux jetons de *N0* vers *N6* engendre la nouvelle structure donnée en figure 17.

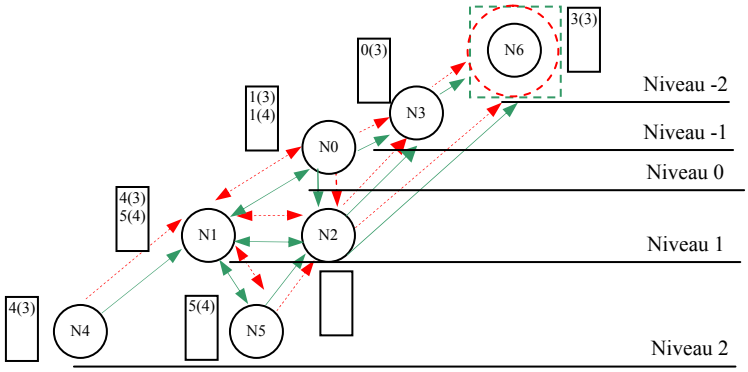


Figure 17 : Réorganisation des arborescences.

La figure 17 permet de mettre en évidence les différents niveaux. Dans celle-ci, le nœud *N6* entre en section critique en n'utilisant que 2 ressources ; Il peut donc satisfaire immédiatement le nœud *N3* en lui envoyant les 3 ressources qu'il demande. Les deux requêtes de *N4* et *N5* qui demandent respectivement 3 et 4 ressources de *N6* ne peuvent pas être satisfaites pour le moment car le nœud *N6* n'a pas assez de ressources. Quand *N6* sort de sa section critique, il envoie le jeton racine au nœud *N5,* mais il garde le jeton clé puisqu'il n'a pas terminé encore de satisfaire les demandes des autres nœuds. La figure 18 montre l'état du réseau après réorganisation suite au déplacement du jeton racine. Dans cet état, *N5* devient le nœud racine. La figure 18 montre aussi l'inversement de liens résultant.

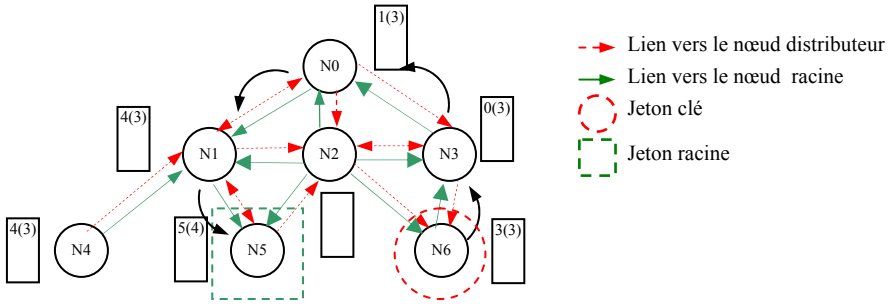


Figure 18 : Modification de l'arborescence racine suite au déplacement du jeton racine.

4.4.4.3 Utilisation de la technique d'inversement de liens suite aux changements de liens physiques

a. Rupture de liens physiques

Au cours d'un déplacement de nœud, il peut y avoir une rupture de lien physique de ce nœud avec un ou plusieurs de ses voisins. Si ce nœud détecte une rupture de lien avec un

voisin, il procède à un traitement lié aux mises à jour des structures d'arborescences. Ce traitement dépend de la relation de son niveau avec celui de son voisin comme suit :

- 1er cas : son voisin est de niveau supérieur, alors, il le supprime de l'ensemble de ses voisins entrants et s'il a des requêtes venant de ce voisin, il les supprime de sa file d'attente ;
- 2ème cas : son voisin est de même niveau, alors il le supprime de l'ensemble de ses voisins de même niveau.
- 3ème cas : son voisin est de niveau inférieur, alors il change éventuellement le chemin vers la racine s'il a encore au moins un lien sortant sinon il va restructurer ses liens logiques avec ses voisins entrants qui ont le plus petit niveau en les inversant et incrémente son niveau en conséquence. Si ce nœud avait des requêtes dans sa file d'attente alors, s'il s'agit de sa propre requête ou celles lui parvenant de nœuds qui sont encore ses voisins, il les relance.

Les trois situations engendrées par la rupture de lien sont illustrées dans l'exemple donné en figure 19.

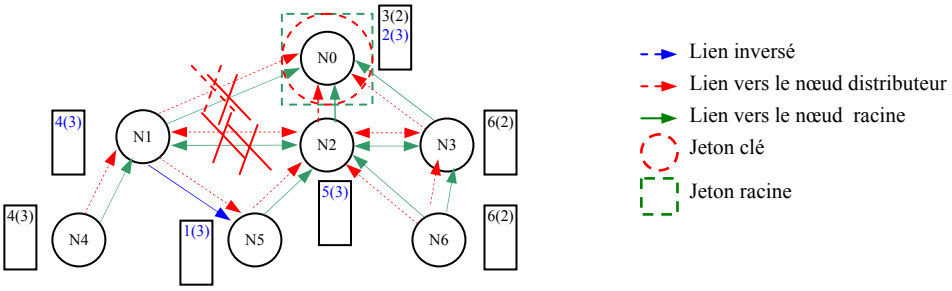


Figure 19 : Inversement de lien suite à des ruptures de liens.

La figure19 montre l'effet d'une défaillance d'un lien sur l'orientation des liens et sur la valeur du niveau des nœuds.

Reprenons l'exemple de la figure 14 sans la requête de *N5* et supposons que le nœud *N1* se déplace et perd en premier ses liens avec *N2* (2ème cas) puis avec *N0* (1er cas pour *N0* et 3ème cas pour *N1*). Suite à ce dernier événement, *N1* perd son dernier lien sortant. Par conséquent, il inverse les liens vers les voisins entrants qui ont le plus petit niveau (*N4* et *N5*). Dans notre cas, *N1* va inverser son lien avec *N5*, il devient au niveau "3" et renvoie une nouvelle requête de *N4* qui est maintenant au niveau "4".

b. Création de liens physiques

En se déplaçant, un nœud *i* peut rencontrer un nouveau voisin *j* et dans ce cas, il suffit simplement de mettre à jour leurs valeurs de niveau. Trois situations peuvent en découler :

1- Si le nœud *j* est de même niveau, alors il est mis dans l'ensemble des voisins de même niveau ;

2- Si le niveau de j est inférieur, alors il est mis dans l'ensemble des voisins sortants ;

3- Si le niveau de j est supérieur, alors il est mis dans l'ensemble des voisins entrants.

La figure 20 illustre les conséquences de la création d'un lien physique dans le réseau de la figure 3. Dans cette figure, le nœud *N5*, en se déplaçant, crée un lien physique avec le nœud *N0*. Par conséquent, *N5* est amené à créer un lien logique sortant avec le nœud *N0*.

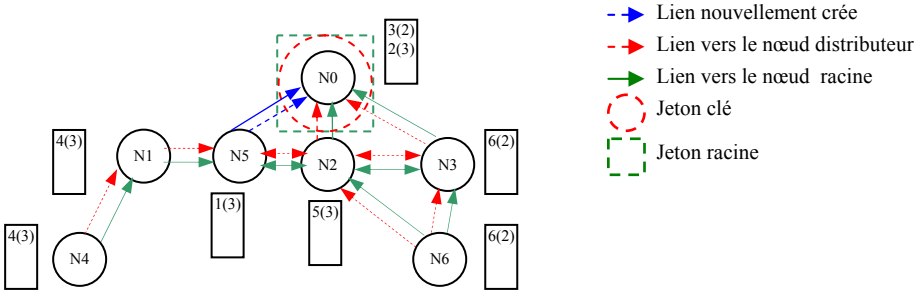


Figure 20 : Mises à jours suite à une création de lien physique.

4.5 Description du protocole

4.5.1 Structures de données

Les variables utilisées dans cet algorithme pour chaque nœud *i* se divise en trois catégories ; variables propres à l'arborescence racine, variables propres à l'arborescence distribution et variables générales, ces variables sont listées ci-dessous :

Variables générales :

- ***State*$_i$:** Indique l'état du nœud *i* : *Req*, *CS* ou *Idle*. Initialement, $state_i$=*Idle*.
- ***N*$_i$:** l'ensemble de tous les nœuds (voisins) qui ont un contact sans fil direct avec le nœud *i*. Initialement, N_i contient tous les nœuds voisins du nœud *i*.
- ***AckSet*$_i$:** L'ensemble des voisins de niveau supérieur qui n'ont pas encore envoyés leurs messages de terminaison de la construction de la structure logique.
- ***LevelVar*$_i$:** Indique le niveau où se trouve le nœud *i*.
- ***Q*$_i$:** Une file qui contient les requêtes des nœuds voisins du niveau immédiatement supérieur, elle est au format (P_j, r_j), où P_j est l'identificateur du nœud qui a demandé l'accès à la section critique et dont la requête est reçue par le nœud *i* et r_j le nombre de ressources demandées par le nœud P_j. Initialement, Q_i=∅.
- ***t*$_i$:** Le nombre de ressources utilisées ou gardées par le nœud *i*.

Variables propres à l'arborescence racine :

- ***OutSetRoot*$_i$:** L'ensemble de tous les nœuds *j* voisins de *i* et qui ont $LevelVar_j < LevelVar_i$; c-à-d les nœuds qui se positionnent juste au dessous du nœud *i* dans l'arborescence. $OutSetRoot_i$ peut changer dans le cas d'un déplacement de jeton racine ou d'un nœud.
- ***InSetRoot*$_i$:** L'ensemble de tous les nœuds *j* voisins de *i* et qui se positionnent juste au dessus du nœud *i* dans l'arborescence.
- ***SameSetRoot*$_i$:** L'ensemble de tous les nœuds *j* voisins de *i* et qui ont $LevelVar_j = LevelVar_i$, c-à-d les nœuds qui se positionnent au même niveau que le nœud *i* dans l'arborescence.
- ***NextRoot*$_i$:** Indique l'emplacement du jeton racine depuis le point de vue du nœud *i*. Quand le nœud *i* possède le jeton racine, *NextRoot* = *i*, sinon $NextRoot_i$ est le nœud sur un lien sortant vers le détenteur de ce jeton. Initialement, $NextRoot_i = 0$ si $i = 0$, autrement $NextRoot_i$ est un voisin sortant appartenant à l'ensemble $OutSetRoot_i$.
- ***TokenRootHolder*$_i$:** Une variable booléenne qui indique si le nœud *i* possède le jeton racine ou non.

Variables propres à l'arborescence distribution :

- ***OutSetKey$_i$:*** L'ensemble qui correspond à *OutSetRoot$_i$* dans l'arborescence distribution. Initialement *OutSetKey$_i$* = *OutSetRoot$_i$*. Il change lors du déplacement de nœuds ou lors du déplacement du jeton clé vers le détenteur du jeton racine et dans ce cas le nouveau *OutSetKey$_i$* = le nouveau *OutSetRoot$_i$*.
- ***InSetKey$_i$* :** L'ensemble qui correspond à *InSetRoot$_i$* dans l'arborescence distribution. Initialement *InSetKey$_i$* = *InSetRoot$_i$* et il ne change pas sauf lors du déplacement de nœuds ou lors du déplacement du jeton clé vers le détenteur du jeton racine et dans ce dernier cas le nouveau *InSetKey$_i$* = le nouveau *InSetRoot$_i$*.
- ***SameSetKey$_i$* :** L'ensemble qui correspond à *SameSetRoot$_i$* dans l'arborescence distribution. Initialement *SameSetKey$_i$* = *SameSetRoot$_i$*. Il change lors du déplacement de nœuds ou lors du déplacement du jeton clé vers le détenteur du jeton racine et dans ce dernier cas le nouveau *SameSetKey$_i$* = le nouveau *SameSetRoot$_i$*.
- ***NextKey$_i$* :** Indique l'emplacement du jeton clé depuis le point de vue du nœud *i*. Quand le nœud *i* possède le jeton clé, *NextKey$_i$* = *i*, sinon *NextKey$_i$* est le nœud sur un lien sortant vers le détenteur de ce jeton. Initialement, *NextKey$_i$* = 0 si *i = 0*, autrement *NextKey$_i$* est un voisin sortant appartenant à l'ensemble *OutSetKey$_i$*.
- ***TokenKeyHolder$_i$* :** Une variable booléenne qui indique si le nœud *i* possède le jeton clé ou non.

4.5.2 Les messages

Les messages utilisés dans l'algorithme sont :

- **Init(*LevelVar$_i$*)** : Message diffusé par le nœud *i* à l'ensemble de ses voisins pour initialiser la structure logique.
- **EndInit()** : Message diffusé par le nœud *i* à l'ensemble de ses voisins pour indiquer la fin de la construction de l'arborescence.
- **Ack()** : Message envoyé par le nœud *i* à ses voisins contenus dans l'ensemble *OutSetRoot$_i$* afin de terminer l'initialisation de la structure logique.
- **Request(*r*) :** Message envoyé par le nœud *i* au nœud voisin indiqué par *NextRoot$_i$*, lorsqu'il souhaite entrer en SC pour accéder à *r* copies de ressources.
- **Release(*s*) :** Message envoyé par le nœud *i* à *NextKey$_i$* lorsqu'il quitte la SC et libère *s* copies de ressources.
- **TokenRoot(*LevelVar$_i$*) :** Message symbolisant le jeton racine envoyé par le nœud racine quand il sort de sa section critique si sa file contient plus d'une requête.
- **TokenKey(*t*) :** Message symbolisant le jeton clé envoyé par le nœud distributeur au nouveau nœud racine où *t, 0* $\leq t \leq$ *k*, indique le nombre de ressources disponibles.
- **AllTokens(*LevelVar$_i$*,*t*) :** Message symbolisant les deux jetons envoyés dans le cas où le nœud racine n'a qu'une seule requête dans sa file d'attente où *t,0*$\leq t \leq$*k*, indique le nombre de ressources disponibles.
- **Resources(*r*) :** Un message envoyé pour satisfaire un nœud qui a demandé *r* copies de ressources.
- **ReverseLinkInfo(*c*, *LevelVar$_i$*) :** Un message utilisé par les nœuds afin d'échanger des informations concernant leurs voisins suivant la valeur de la variable *c* :
  - c=1 : le cas de déplacement des deux jetons racine et clé;
  - c=2 : le cas de déplacement du jeton racine ;
  - c=3 : le cas de déplacement du jeton clé ;
  - c=4 : le cas de perte du dernier lien logique sortant.

4.5.3 Les fonctions et les procédures

- **ReverseLink(*c , j*) :** Procédure exécutée par un nœud *i* dans des cas précis suivant la valeur de la variable *c* :
  - c=1 : quand *i* envoie les deux jetons, il inverse ses deux liens respectifs aux deux arborescences avec le nœud *j* qui devient son nouveau $NextRoot_i$ et son nouveau $NextKey_i$;
  - c=2 : quand *i* envoie le jeton racine (et garde le jeton clé), il inverse le lien avec *j* dans l'arborescence racine et son $NextRoot_i$ devient le nœud *j* ;
  - c=3 : quand *i* envoie le jeton clé, il met à jour ses ensembles de l'arborescence distribution et par conséquent inverse les liens correspondants ;
  - c=4 : quand *i* reçoit uniquement le jeton racine et devient le nouveau nœud racine, il met à jour ses ensembles de l'arborescence racine.
  - c=5 : quand *i* reçoit les deux jetons et devient le nouveau nœud racine et le nouveau nœud distributeur.
- **RearrangeRoot():** Procédure d'inversement de liens de l'arborescence racine en cas de rupture de lien physique.
- **RearrangeKey():**Procédure d'inversement de liens de l'arborescence distribution en cas de rupture de lien physique.
- **SetLevel(*j,*$Level_j$*,Set*) :** Fonction qui change la valeur du niveau du nœud *j* dans l'ensemble *Set* du nœud *i* qui l'exécute où $Level_j$ est la nouvelle valeur de $LevelVar_j$.
- **MinLevel(*Set*) :** Fonction qui retourne de l'ensemble *Set* le sous-ensemble des nœuds qui ont le plus petit niveau.
- **SameLevel(*Set*) :** Fonction qui retourne de l'ensemble *Set* le sous-ensemble des nœuds qui ont le même niveau.
- **Choose(*Set*) :** Fonction qui permet de choisir un nouveau *NextRoot* si *Set* est *OutSetRoot* ou un nouveau *NextKey* si *Set* est *OutSetKey*.

4.5.4 Fonctionnement du protocole

- **Processus d'application**

Le processus d'application du nœud *i* envoie un message $RequestCS_i$ au protocole de la h-k exclusion mutuelle lorsqu'il désire accéder à des ressources. Il attend jusqu'à ce qu'il reçoive le message $EnterCS_i$. Lorsqu'il termine l'utilisation des ressources, il les libère en envoyant le message $ReleaseCS_i$.

```
Begin
< non Critical Section >
     Send RequestCS_i to mutual exclusion process;
     Wait for EnterCS_i ;
< Critical Section >
     Send ReleaseCS_i to mutual exclusion process;
End
```

- **Service de la h-k exclusion mutuelle**

1- Initialisation du Protocole et construction de la structure logique

Initialement, le nœud " 0 " est le nœud racine et c'est lui qui commence la construction de la structure logique. Il initialise ses variables communes décrites dans le pseudo-code. Les autres nœuds initialisent aussi leurs contextes en mettant $state_i$ à *Idle*, $LevelVar_i$ à "–1",

$TokenRootHolder_i$ à False ; $TokenKeyHolder_i$ à False, $InSetRoot_i$ (Resp $InSetKey_i$) à N_i, $OutSetRoot_i$ et $SameSetRoot_i$ à ∅ (Resp $OutSetKey_i$ et $SameSetKey_i$ à ∅). Puis le nœud "0" démarre le processus de construction de l'arborescence en diffusant un message Init(0,0) à l'ensemble N_0 des nœuds voisins pour qu'ils continuent la construction de l'arborescence.

Remarque : On suppose que lors de la construction de la structure logique, le réseau reste statique.

Pseudo-code **(Démarrage du processus de la h-k EM au niveau du nœud *i*)**

```
When i initializes the h-k mutual exclusion process
Do //initialise all variables
    State_i := Idle ;
    InSetRoot_i := {j / ∀ j ∈ N_i};
    InSetRootLevel_i := {(j,-1) / ∀ j ∈ N_i}; // set of pairs (neighbour, level);
    InSetKey_i := InSetRoot_i ;
    AckSet_i := InSetRoot_i ;
    OutSetRoot_i := Φ ; OutSetKey_i := Φ ; OutSetRootLevel_i := Φ ;
    SameSetRoot_i := Φ ; SameSetKey_i := Φ ; SameSetRootLevel_i := Φ ;
    If (i=0) then  // if node designated to be the root
        t_i := k;
        LevelVar_i := 0 ;
        TokenRootHolder_i :=  True  ;
        TokenKeyHolder_i := True  ;
        NextRoot_i := i ; NextKey_i := i ;
        Send Init(0) to N_i ;
    Else
        t_i := 0;
        Q_i := Φ ;
        LevelVar_i := -1 ;
        TokenRootHolder_i :=  False  ;
        TokenKeyHolder_i := False  ;
    EndIf
EnDo
```

Après l'initialisation, un nœud *i* autre que le nœud "0" peut recevoir une première fois un message Init*(j,LevelVar$_j$)* d'un nœud voisin *j*, son niveau étant égal à "-1". Dans ce cas, *i* met $LevelVar_i = (LevelVar_j + 1)$, et son ensemble $OutSetRoot_i$ *à j*.

Le nœud *i* peut recevoir d'autres messages Init*(j,LevelVar$_j$)* d'un nœud voisin *j,* ce voisin peut être :

a) De même niveau : ie $LevelVar_i$ est égale à $LevelVar_j$, alors il rajoute simplement *j* dans son ensemble $SameSetRoot_i$. Le nœud *i* n'a pas a envoyer une autre fois le message Init*(i,LevelVar$_i$)*.

b) De niveau supérieur : Dans ce cas, *i* ne fait rien et sort immédiatement de la procédure.

c) De niveau inférieur :

- D'un seul pas : ie $LevelVar_i$ est égale à $(LevelVar_j+1)$, alors il rajoute *j* dans son ensemble $OutSetRoot_i$. Le nœud *i* n'a pas a envoyer une autre fois le message Init*(i,LevelVar$_i$)*.
- De deux pas : ie $LevelVar_i$ est égale à $(LevelVar_j+2)$, alors *i* met $LevelVar_i=(LevelVar_j+1)$, rajoute $SameSetRoot_i$ *à* $InSetRoot_i$, *met* $SameSetRoot_i = OutSetRoot_i$ et écrase son ensemble $OutSetRoot_i$ par *j*.

- De plus de deux pas : ie $LevelVar_i$>($LevelVar_j$+2), alors *i* met $LevelVar_i$=($LevelVar_j$+1), rajoute les deux ensembles SameSetRoot et $OutSetRoot_i$ *à* $InSetRoot_i$, écrase son ensemble $OutSetRoot_i$ par *j* et met *SameSetRoot à* ∅.

Dans tous les cas ci-dessus, à part le cas ou son voisin *j* est de niveau supérieur, le nœud *i* supprime *j* de ses ensembles $InSetRoot_i$ et $AckSet_i$ et teste si $InSetRoot_i$ devient vide, si c'est le cas alors il envoie *Ack*(*i*) à $OutSetRoot_i$ sinon il envoie Init*(i,*$LevelVar_i$*)* à $InSetRoot_i$.

***Pseudo-code* (*i* reçoit *Init(LevelVarj)*)**

```
When i receives Init(LevelVarj) from j
Do
    SInit := False ;
    Case LevelVar_i

     -1   :    // LevelVar_i not initialised yet
          LevelVar_i := LevelVar_j + 1 ;
          OutSetRoot_i := {j} ;
      LevelVar_j  : // LevelVar_i = LevelVar_j
          Add(j, SameSetRoot_i) ;
          Sinit := True ;
      LevelVar_j + 1:
          Add(j, OutSetRoot_i) ;
          Sinit := True;
     LevelVar_j + 2:
          LevelVar_i := LevelVar_j + 1 ;
          InsetRoot_i := InSetRoot_i + SameSetRoot_i ;
          SameSetRoot_i := OutSetRoot_i ;
          OutSetRoot_i := {j} ;
     > LevelVar_j + 2:
          LevelVar_i := LevelVar_j + 1 ;
          InSetRoot_i := InSetRoot_i + OutSetRoot_i + SameSetRoot_i;
          AckSet_i:= InSetRoot_i;
          OutSetRoot_i:= {j} ;
          SameSetRoot_i := ∅;
     Default:
          Exit; //end sub
    EndCase
    Remove(j, InSetRoot_i) ;
    Remove(j, AckSet_i) ;
    If (InSetRoot_i = ∅) then // i is a leaf of tree
        Send Ack() to OutSetRoot_i;
    Else
        If (SInit = False) then
          Send Init(LevelVar_i) to InSetRoot_i;
        EndIf
    EndIf
End
```

2- Fin de la construction de l'arborescence :

Les nœuds du dernier niveau se caractérisent par un *InsetRoot* vide alors c'est eux qui doivent commencer à envoyer un message d'accusé de réception « *Ack* » à leurs voisins qui se trouvent dans *OutSetRoot*. Ainsi, lorsque les nœuds du niveau inférieur reçoivent tous les messages *Ack* de leurs voisins se trouvant dans *InSetRoot*, ils envoient à leur tour un accusé de réception à leurs voisins dans *OutSetRoot* jusqu'à ce que le nœud "0″ reçoive tous les

messages *Ack* de ses voisins ; à ce moment, l'algorithme d'initialisation se termine et le nœud "0" envoie un message de fin d'initialisation du protocole *EndInit()* à $InSetRoot_0$.

Quand un nœud *i* reçoit Ack(*j*) de *j*, il supprime *j* de son ensemble $AckSet_i$. Si son ensemble $AckSet_i$ devient vide alors, s'il est le nœud "0" il diffuse EndInit() à tous ces voisins ; sinon il envoie à son tour un message *Ack(i*) à ses voisins qui sont dans $OutSetRoot_i$.

***Pseudo-code* (*i* reçoit *Ack*())**

```
When i receives Ack() from j
Do
    Remove(j,AckSet_i);
    If AckSet_i = ∅ then
     If i ≠ 0 then
          Send Ack() to OutSetRoot_i;
     Else
          Send EndInit() to InSetRoot_0;
     EndIf
    EndIf
EnDo
```

Quand un nœud *i* reçoit EndInit() de *j*, il termine l'initialisation de ses structures de données restantes et il envoie un message *EndInit*() à ses voisins entrants. Ce message est envoyé une seule fois.

***Pseudo-code* (*i* reçoit *EndInit*())**

```
When i receives EndInit() from j
Do
    If (RunOnce = False) then
     RunOnce := True;
     ∀ k∈InSetRoot_i SetLevel(k,LevelVar_i+1,InSetRoot_i)/; //fonction qui change la
     OutSetKey_i := OutSetRoot_i ;                  // valeur de la variable Level
     InSetKey_i := InSetRoot_i ;
     SameSetKey_i:= SameSetRoot_i ;
     NextRoot_i := j ; NextKey_i := j ;
     Send EndInit() to InSetRoot_i;
    EndIf
EnDo
```

3- Demande d'accès en section critique

Quand un nœud *i* désire entrer en section critique, il met $State_i=Req$ et il enfile son identificateur ainsi que le nombre de ressources désirées dans sa file Q_i et envoie un message *Request(r)* à son voisin du niveau inférieur ($NextRoot_i$) s'il n'est pas le détenteur du jeton racine. S'il détient les deux jetons et n'a que sa demande en file alors il met $State_i$ à CS, supprime sa demande de sa file Q_i et entre en section critique.

***Pseudo-code* (*i* veut entrer en SC et demande *r* copies de ressources)**

```
When i requests to enter the CS to access r resources
Do
    State_i := Req ;
    Insert (Q_i, i, r) ;
    If (not TokenRootHolder_i) then
        Send Request(r) to NextRoot_i ;
    Else
     If(TokenKeyHolder_i and (|Qi|=1) and (r ≤ t_i)) then // i have Tokens and want
        State_i := CS ;                                    // to enter CS
        t_i := t_i –r ;
        Delete(QHeader(Q_i));
        Send EnterCS_i to Application process ;
     EndIf
    EndIf
EnDo
```

Quand un nœud *i* reçoit un message *Request(r)* d'un nœud *j*, il exécute la séquence suivante :

1- Il met identificateur de j avec le nombre de ressources demandées dans sa file Q_i ;

2- Il envoie un message *Request(r)* à $NextRoot_i$ s'il ne possède pas le jeton racine. Ainsi, le message *Request(r)* va suivre les liens orientés d'un niveau à l'autre jusqu'à ce qu'il arrive au nœud racine.

3- Si le nœud *i* est le nœud racine, alors il vérifie s'il détient ou non le jeton clé ; si oui (c-à-d *TokenKeyHolder* = True) alors s'il est en section critique, il vérifie si cette requête est la seule dans sa file ou s'il y a d'autres requêtes en attente. Si elle est unique alors le nœud *i* tente de la satisfaire en testant si *r* (l'ensemble des ressources demandées) est inférieur ou égale à *t* (l'ensemble des ressources disponibles). Si oui, alors il envoie *Resource(r)* à *j* et supprime cette demande de sa file Q_i. Si le nœud *i* détient les deux jetons, il n'est pas en SC et n'est pas demandeur de ressources alors il envoie les deux jetons à *j*, inverse ses liens vers lui et supprime cette demande de sa file Q_i.

***Pseudo-code* (*i* reçoit *Request*(r) de *j*)**

```
When Request(r) is received from j
Do
    Insert (Qi, j, r) ;
    If (TokenRootHolder_i )then // i is root node
     If (TokenKeyHolder_i)then
        If (State_i = CS)then // i is in CS
          If (r ≤ t_i and |Q_i| = 1 ) then
             Send Resources(r)to j ;
             t_i := t_i –r ;
             Delete (QHeader(Q_i)) ;
          EndIf
        Else
          If (State_i ≠ Req)then // i have tokens but do nothing
             TokenRootHolder_i := False ;
             TokenKeyHolder_i := False ;
             Send AllTokens(t) to j;
             Delete (QHeader(Q_i)) ;
             t_i := 0;
             NextRoot_i := j ; NextKey_i := j ;
             ReverseLink(1,j);
          EndIf
        EndIf
     EndIf
    Else //Not TokenRootHolder
        Send Request(r) to NextRoot_i ;
    EndIf
EnDo
```

Quand un nœud *i* reçoit un message *Resource(r)*, s'il est le nœud concerné par cette distribution (c-à-d si $QHeader(Q_i).id = i$), alors il met son état $State_i$ à CS et entre en section critique. Sinon, il envoie *Resource*(*r*) à $QHeader(Q_i).id$ et dans les deux cas, il supprime l'élément de tête de file Q_i. Si ni *i* ni sa tête de file n'est le destinataire, cela veut dire que la requête du destinataire a été supprimée suite à une rupture de lien ; dans ce cas, les ressources envoyées seront libérées par un message *Release(r)* envoyé à $nextKey_i$.

***Pseudo-code* (*i* reçoit *Resource*(r) de *j*)**

```
When i receives Resources(r)
Do
    If ( QHeader(Q_i).id = i ) then
        State_i := CS ;
        Delete(QHeader(Q_i));
        Send EnterCS_i to Application process ;
    Else
     If (QHeader(Q_i).r = r) then //requête du destinataire est elle supprimée ?
        Send Resources(r) to QHeader(Q_i).id;
        Delete(QHeader(Q_i));
     Else//i à perdu son lien entrant avec le destinataire et la requête de ce dernier a été supprimée
        Send Release(r) to NextKey_i ;
     EndIf
    EndIf
EnDo
```

4- Sortie de la section critique

Quand un nœud *i* sort de sa section critique et libère *s* copies de ressources, il met $State_i$ à *Idle* et s'il n'est pas le nœud racine ; il envoie seulement un message *Release(s)* à $NextKey_i$.
Si *i* est le nœud racine alors, il exécute la séquence suivante :

1- Il met $t_i = t_i + s$;

2- Si sa file contient un seul élément, alors il lui envoie les deux jetons accompagnés du nombre de ressources disponibles, inverse ses liens vers cet élément et supprime sa demande de sa file.

Si sa file contient plusieurs demandes alors, il envoie *TokenRoot* à l'élément de fin de file, il inverse son lien vers cet élément, puis appel la procédure *Satisfy()*. En appelant cette procédure, il entre dans une boucle de distribution afin de satisfaire les demandes. Dans cette boucle, il teste si sa file Q_i contient plus d'un élément et si *r* est inférieur ou égale à *t*, dans ce cas, il satisfait les demandes une après l'autre en envoyant *Resource(r)* jusqu'à ce que *r* devienne supérieur à *t* ou s'il arrive à l'élément de fin de file. S'il ne reste qu'un seul élément dans sa file, alors il lui envoie *TokenKey(t)* et inverse son lien vers lui. Il supprime bien sur chaque demande satisfaite de sa file Q_i.

Pseudo-code **(*i* sort de sa section critique et libère *s* copies de ressources)**

```
When i leaves the CS and releases s copies of resources
Do
    State_i = Idle ;
    If ( TokenRootHolder_i ) then // have certainly TokenKey
      t_i := t_i + s ;
      If (|Q_i|= 1) then
          Send AllTokens(LevelVar_i,t_i) to QHeader(Q_i).id ;
          TokenRootHolder_i := False ;
          TokenKeyHolder_i := False ;
          t_i := 0 ;
          ReverseLink(1,QHeader(Q_i).id );
          Delete(QHeader(Q_i));
     Else
          If (|Q_i|> 1) then
             Send TokenRoot(LevelVar_i) to Qtail(Q_i).id ;
             TokenRootHolder_i := False ;
             ReverseLink(2,Qtail(Q_i).id );
             Satisfy();
          EndIf
     EndIf
    Else // certainly TokenKeyHolder = False
     Send Release(s) to NextKey_i ;
    EndIf
EnDo
```

```
Procedure Satisfy()
Begin
    While (|Qi| > 1 and QHeader(Qi).r ≤ ti)
        Send Resource(r)to QHeader(Qi).id ;
        ti := ti – QHeader(Qi).r ;
        Delete(QHeader(Qi));
    EndWhile
    If (|Qi| = 1)then
        If (not TokenRootHolderi ) then
            Send TokenKey(ti)to QHeader(Qi).id ;
            TokenKeyHolderi := False ;
            ti := 0 ;
            ReverseLink(3,null) ;
            Delete(QHeader(Qi)) ; //Qi ={∅}
        EndIf
        If(TokenRootHolderi and QHeader(Qi).r ≤ ti)then //i is still in CS
          Send Resource(r) to QHeader(Qi).id ;
          ti := ti – QHeader(Qi).r ;
          Delete(QHeader(Qi));
        EndIf
     EndIf
End
```

Quand un nœud *i* reçoit un message *Release(s)*, il exécute la séquence suivante s'il détient le jeton clé ; autrement, il envoie seulement *Release(s)* à $NextKey_i$:

1- Il ajoute *s* à l'ensemble de ressources disponibles ($t := t+s$) ;

2- Il appel la procédure *Satisfy()*. Rappelons que cette procédure satisfait les requêtes présentent dans la file Q_i en envoyant *Resource(r)* tant que les ressources disponibles suffisent. Ensuite, si la file de *i* contient un seul élément et si le nœud *i* ne détient pas le jeton racine alors, il envoie *TokenKey(t)* à cet élément. Sinon s'il détient le jeton racine (il est encore en section critique), il satisfait les requêtes jusqu'à ce que la file devienne vide ou les ressources disponibles ne suffisent plus. A chaque fois qu'il satisfait une requête, il supprime de sa file Q_i l'identificateur du nœud demandeur.

***Pseudo-code* (*i* reçoit *Release(s)*)**

```
When i receives Release(s)
Do
    If (tokenKeyHolderi ) then
        ti := ti + s ;
        Satisfy() ;
    Else
        Send Release(s) to NextKeyi ;
    EndIf
EnDo
```

5- <u>Déplacements des jetons</u>

<u>Quand un nœud *i* reçoit un message *AllTokens(LevelVarj, t)*</u>, il décrémente son niveau, met $TokenRootHolder_i$ et $TokenKeyHolder_i$ à Vrai et suivant le contenue de sa file de requêtes, il décide :

- Si sa file Q_i est vide (cela peut arriver dans le cas de rupture de liens), le nœud *i* garde les deux jetons en attendant de nouvelles requêtes, il envoie le message *ReverseLinkInfo(*1*, $LevelVar_i$)* à tous ses voisins hormis *j* pour les informer de sa nouvelle valeur de niveau et inverse ses liens vers ces voisins.
- Sinon, deux cas se présentent :
  - le nœud *i* est le nœud concerné par ce message (c-à-d si *QHeader*(Q_i).*id*=*i*) alors il inverse ses liens vers ses voisins et leur envoie le message *ReverseLinkInfo(*1*, $LevelVar_i$)* pour les informer qu'il est devenue la nouvelle racine. Si les ressources disponibles suffisent, il met son état $State_i$ à CS, supprime l'élément de tête de file Q_i et entre en section critique. Ensuite, Il entre dans la boucle *Satisfy* pour satisfaire d'éventuelles requêtes dans la file Q_i.
  - Si *i* n'est pas le nœud concerné par le message *AllTokens*, il envoie *AllTokens*($LevelVar_i$, *t*) à *QHeader*(Q_i).*id*, inverse ses liens vers ses voisins hormis *j* et *QHeader*(Q_i).*id* et leur envoie le message *ReverseLinkInfo(*1*, $LevelVar_i$)*. Il supprime l'élément de tête de file Q_i.

***<u>Pseudo-code</u>* (*<u>i reçoit AllTokens($LevelVar_j$, t) de j</u>*)**

```
When i receives AllTokens(LevelVar_j, t) from j
Do
    LevelVar_i := LevelVar_j - 1 ;
    TokenRootHolder_i := True ;
    TokenKeyHolder_i := True ;
    If (Q_i ≠ ∅) then
     If (QHeader(Q_i).id = i) then // Tokens is for me
        t_i := t ;
        Send ReverseLinkInfo(1,LevelVar_i) to N_i-{j} ;
        ReverseLink(5);
        If (r ≤ t_i) then
           State_i := CS ;
           t_i := t_i - QHeader(Q_i).r ;
           Delete(QHeader(Q_i));
           Send EnterCS_i to Application process ;
           Satisfy() ;
        EndIf
     Else
        Send AllTokens(LevelVar_i,t) to QHeader(Q_i).id ;
        Send ReverseLinkInfo(1,LevelVar_i) to N_i-{j+QHeader(Q_i).id};
        ReverseLink(1,QHeader(Q_i).id );
        Delete(QHeader(Q_i));
        TokenRootHolder_i := False ;
        TokenKeyHolder_i := False ;
        t_i := 0 ;
     EndIf
    Else // La requête du destinataire est supprimée suite à un linkdown
         t_i := t ;
         Send ReverseLinkInfo(1,LevelVar_i) to N_i-{j} ;
         ReverseLink(5,null);
    EndIf
EnDo
```

Quand un nœud *i* reçoit *TokenRoot(LevelVarj)*, il diminue de 1 sa valeur de niveau de tel sorte qu'il devienne dans le niveau le plus bas. Il envoie un message d'information *ReverseLinkInfo(2, LevelVar$_i$)* à ses voisins concernés puis vérifie s'il est le dernier élément de sa file Q_i (c-à-d, le jeton racine lui est destiné). Si oui, il met *TokenRootHolder$_i$* à Vrai et inverse ses liens. Si le nœud *i* n'est pas le dernier de sa file, il envoie seulement *TokenRoot(i, LevelVar$_i$)* à l'élément de fin de file *(Qtail(Q_i)).id*.

Le nœud *i* ne va pas réenvoyer une autre requête au nouveau nœud racine puisqu'il est sûr que les demandes restantes dans sa file seront satisfaites par l'ancien nœud racine, celui qui a encore le jeton clé.

***Pseudo-code* (*i* reçoit TokenRoot)**

```
When i receives TokenRoot(LevelVar_j) from j
Do
    LevelVar_i := LevelVar_j - 1 ;
    If (Q_i = ∅ or Qtail(Q_i).id = i) then // TokenRoot is for me
      TokenRootHolder_i := True ;
      Send ReverseLinkInfo(2,LevelVar_i) to N_i-{j};
      ReverseLink(4,null);
    Else
      Send TokenRoot(LevelVar_i) to Qtail(Q_i).id ;
      Send ReverseLinkInfo(2,LevelVar_i) to N_i-{j+Qtail(Q_i).id};
      ReverseLink(2,Qtail(Q_i).id);
    EndIf
EnDo
```

Quand un nœud *i* reçoit *TokenKey(t)*, il inverse ses liens vers les voisins concernés et leur envoie un message d'information *ReverseLinkInfo*(3,*LevelVar$_i$*). Ensuite, s'il est le nœud racine, il garde le jeton clé, puis commence à satisfaire les demandes de sa file une après l'autre en commençant par la sienne, auquel cas il entre en section critique si le nombre de ressources disponible est suffisant et continue à satisfaire les autres demandes en testant toujours la valeur de la variable t_i et en supprimant la requête satisfaite. Autrement, il attend la libération de ressources. Sinon, s'il n'est pas le nœud racine alors il envoie *TokenKey(t)* à *NextRoot$_i$*.

***Pseudo-code* (*i* reçoit TokenKey)**

```
When i receives TokenKey(t)
Do
    Send ReverseLinkInfo(3,LevelVar_i) to Ni-{j} ;
    ReverseLink(3,null) ;
    If (TokenRootHolder_i )then // i is root node
        TokenKeyHolder_i = True ;
        t_i := t ;
        If (QHeader(Q_i).id=i and QHeader(Q_i).r ≤ t) then//My Request in queue header
          t_i = t_i – r ;
          State_i = CS ;
          Send EnterCS_i to Application process ;
          Delete(Qheader(Q_i));
          While |Q_i| ≥ 1 et QHeader(Q_i).r ≤ t
                Send Resource(r) to QHeader(Q_i).id ;
                t_i = t_i – QHeader(Q_i).r ;
                Delete(QHeader(Q_i));
          EndWhile
        EndIf
    Else
        Send TokenKey(t) to NextRoot_i ;
    EndIf
EnDo
```

6- Création et défaillance de lien

Quand un nœud *i* détecte que le lien vers *j* est rompu, il commence par supprimer *j* de l'ensemble de ses voisins puis :

- Si ce lien est entrant alors, il suffit de l'enlever de $InSetRoot_i$ (resp, $InSetKey_i$) et si *j* a émis des requêtes et qui sont dans Q_i, elles seront toutes supprimées.
- Si le nœud *j* est dans le même niveau que *i* alors, *i* enlève *j* de son ensemble $SameSetRoot_i$ (resp, $SameSetKey_i$)
- Si le lien vers *j* est sortant ; dans ce cas, il l'enlève de $OutSetRoot_i$ (resp, $OutSetKey_i$). Il vérifie ensuite si *j* est $NextRoot_i$ (resp, $NextKey_i$) ; dans ce cas, il faut qu'il choisisse un autre $NextRoot_i$ (resp, $NextKey_i$) et si sa file de requêtes Q_i n'est pas vide, il doit relancer de nouvelles requêtes en suivant l'ordre FIFO dans la file vers le nouveau *NextRoot*. Mais si le lien vers *j* est le dernier lien sortant, la procédure d'inversement de liens (*Partial Reversal procedure*) que nous avons nommé RearrangeRoot() (resp RearrangeKey()) est appelée ; ces procédures mettent à jour les ensembles $InSetRoot_i$ (resp, $InSetKey_i$), $OutSetRoot_i$ (resp, $OutSetKey_i$), $SameSetRoot_i$ (resp, $SameSetKey_i$) et la valeur de $NextRoot_i$ (resp $NextKey_i$). En plus de ces structures de données, la procédure RearrangeRoot() met à jour la variable $LevelVar_i$ comme suit :
  - Si le nœud *i* n'a aucun voisin de même niveau, alors il choisi parmi ses voisins entrants ceux qui ont la plus petite valeur de niveau pour être ses nouveaux voisins sortants en exécutant la fonction MinLevel($InSetRoot_i$). Ce choix est motivé par le fait que l'ensemble des nœuds qui ont la plus petite valeur de niveau est le plus près de la racine. Ensuite, *i* choisi sont *next* parmi ces voisins. Enfin, *i* envoie un message d'information à ses nouveaux voisins.
  - Si le nœud *i* possède encore des liens sortants vers des nœuds qui sont dans le même niveau que lui, alors il incrémente sa valeur de niveau, met ses voisins de même niveau dans son ensemble de voisins sortants, choisi un autre *next* et envoie un message d'information à tous ses voisins.

***Pseudo-code* (*i* détecte une rupture de lien avec *j*)**

```
When failure of link to j detected at node i
Do
    Ni := Ni − {j};
    If j ∈ InSetRooti then
        Remove(j,InSetRooti) ;
        Delete(Qi,j); //supprime de Qi toutes les requêtes émises par j
    Else
     If j ∈ OutSetRooti then
        Remove(j,OutSetRooti) ;
        If (NextRooti = j) then
          If (OutSetRooti = Φ) then RearrangeRoot() ;
          Else
            NextRooti :=Choose(OutSetRooti);//NextRooti:=OutSetRooti[1]
          EndIf
          If Qi≠ Φ then
            P := QHeader(Qi);
            While not end queue(Qi)
              Send Request(P.r) to NextRooti ;
              P := P.Next ;
            EndWhile
          EndIf
        EndIf
     Else
        If j ∈ SameSetRooti then Remove(j,SameSetRooti) ;
        EndIf
     EndIf
    EndIf
    If j ∈ InSetKeyi then
     Remove(j,InSetKeyi) ; // InSetKeyi=InSetKeyi-{i}
    Else
     If j ∈ OutSetKeyi then
        Remove(j,OutSetKeyi) ;
        If (NextKeyi = j) then
          If (OutSetKeyi = Φ) then RearrangeKey() ;
          Else
            NextKeyi := Choose(OutSetKeyi);
          EndIf
        EndIf
     Else
        If j ∈ SameSetKeyi then Remove(j,SameSetKeyi) ;
        EndIf
     EndIf
    EndIf
EnDo
```

***Pseudo-code* (*RearrangeRoot()*)**

```
When i executes RearrangeRoot()
Do
  If (SameSetRoot_i = Φ) then
     (OutSetRoot_i, LevelVar_i) := MinLevel(InSetRoot_i); //Fonction qui maj OutSetRoot_i et LevelVar_i
     InSetRoot_i := InSetRoot_i - OutSetRoot_i;
  Else
     LevelVar_i := LevelVar_i + 1;
     OutSetRoot_i := SameSetRoot_i;
  EndIf
  SameSetRoot_i := SameLevel(InSetRoot_i);
  InSetRoot_i := InSetRoot_i - SameSetRoot_i ;
  NextRoot_i := Choose(OutSetRoot_i);
  Send ReverseLinkInfo(4, LevelVar_i) to N_i;
EnDo
```

La procedure RearrangeKey() :

Afin d'optimiser le traitement et d'éviter les messages issues de la réorganisation de l'arborescence distribution, le nœud qui perd sont dernier lien sortant dans cette arborescence, empreinte les mêmes chemins pris par son voisin *NextRoot* de l'arborescence racine. Le fait de réorganiser cette branche pour avoir un chemin optimisé vers le détenteur du jeton clé ; par exemple, choisir le voisin avec la plus petite valeur de niveau pour être *NextKey*, coûte plus chère que si *NextRoot* qui devient *NextKey*, car le nœud distributeur, après avoir terminé de satisfaire les requêtes en sa possession va rejoindre le nœud racine et finalement le nouveau *NextKey* devient *NextRoot*.

***Pseudo-code* (*RearrangeKey()*)**

```
When i executes RearrangKey()
Do
    OutSetKey_i := OutSetRoot_i ;
    SameSetKey_i := SameSetRoot_i ;
    InSetKey_i = InSetRoot_i;
    NextKey_i := NextRoot_i;
EnDo
```

Quand un nœud *i* détecte la création d'un lien vers un nœud *j*, en recevant $LinkUp_i(j)$, *i* va comparer la valeur de son niveau par rapport au niveau de j :

- Si $LevelVar_i > LevelVar_j$, alors il ajoute *j* dans $OutSetRoot_i$ (resp, $OutSetKey_i$).
- Si *i* trouve que $LevelVar_i < LevelVar_j$ alors il ajoute *j* dans son ensemble $InSetRoot_i$ (resp $InSetKey_i$).
- Si $LevelVar_i = LevelVar_j$ alors il ajoute *j* dans son ensemble $SameSetRoot_i$ (resp, $SameSetKey_i$).

***Pseudo-code* (*i* détecte une formation d'un nouveau lien avec *j*)**

```
When LinkUp_i(j) is received from Network Layer
Do
    If LevelVar_i > (LevelVar_j) then
        Add(j, OutSetRoot_i); Add(j,OutSetKey_i) ;
        // la fonction Add mis à jour LevelVar_j dans les ensembles appropriés
    Else
        If LevelVar_i < (LevelVar_j) then
          Add(j,InSetRoot_i); Add(j,InSetKey_i) ;
        Else
          If LevelVar_i = LevelVar_j then
             Add(j, SameSetRoot_i) ;  Add(j, SameSetKey_i) ;
          EndIf
        EndIf
    EndIf
EnDo
```

Quand un nœud *i* exécute ReverseLink(*c*, *j*), le traitement suit la valeur de *C*:

- Si *C*=1 : *i* doit inverser ses liens vers *j* dans les deux arborescences, car *i* vient de lui envoyer les deux jetons. Dans ce cas, *j* devient son unique lien sortant et son *NextRoot*(resp *NextKey*) et tous les autres voisins sont dans $InSetRoot_i$(resp $InSetKey_i$).
- Si C=2 : seules les structures de données de l'arborescence racine sont mises à jour car *i* vient d'envoyer le jeton racine au nœud *j*.
- Si C=3 : *i* vient d'envoyer le jeton clé au nœud *j* ; dans ce cas, seules les structures de données de l'arborescence clé sont mises à jour.
- Si C=4 : *i* vient de recevoir le jeton racine, il doit inverser tous ses liens pour recevoir les nouvelles requêtes d'entrées en sections critiques.
- Si C=5 : *i* vient de recevoir les deux jetons, il doit inverser tous ses liens de tel sorte qu'il devienne un puit.

```
Procedure ReverseLink(c,j)
Begin
    Case c
        1: // i envoie les deux jetons au nœud j
            InSetRoot_i = InSetRoot_i+OutSetRoot_i+SameSetRoot_i-{(j,LevelVar_j)};
           InSetKey_i = InSetRoot_i; OutSetRoot_i := {(j, LevelVar_i-1)} ;
           SameSetRoot_i := ∅ ;
           OutSetKey_i := OutSetRoot_i;  NextRoot_i := j; NextKey_i := j;
        2: // i envoie le jeton racine au nœud j
            InSetRoot_i = InSetRoot_i+OutSetRoot_i+SameSetRoot_i-{(j,LevelVar_j)};
            SameSetRoot_i := ∅ ;
           OutSetRoot_i := {(j, LevelVar_i-1)} ; NextRoot_i := j;
        3: // i envoie le jeton clé au nœud j
            OutSetKey_i := OutSetRoot_i;    InSetKey_i = InSetRoot_i;
           SameSetKey_i := SameSetRoot_i ; NextKey_i := Nextroot_i;
        4: // i reçoit le jeton racine uniquement et devient le nouveau nœud racine
            InSetRoot_i = InSetRoot_i+ OutSetRoot_i+ SameSetRoot_i;
            OutSetRoot_i = {∅}; SameSetRoot_i = {∅};
            NextRoot_i = i;
        5: // i reçoit les deux jetons et devient le nouveau nœud racine et le nouveau distributeur
           InSetRoot_i := InSetRoot_i+ OutSetRoot_i+ SameSetRoot_i;
            OutSetRoot_i := {∅}; SameSetRoot_i := {∅}; OutSetKey_i := {∅};
            InSetKey_i := InSetRoot_i; SameSetKey_i := {∅};
           NextRoot_i := i; NextKey_i := i;
    EndCase
End
```

Quand un nœud *i* reçoit *ReveseLinkInfo*(*c*, *LevelVar*$_j$), le traitement dépend de la valeur de *C* :

- Si *C*=1 : Cela veut dire que *j* a reçu les deux jetons et il est devenu le nouveau nœud racine et le nouveau distributeur de ressources ; dans ce cas, *i* doit pointer sur *j* qui devient son nouveau *NextRoot* (resp *NextKey*). *i* met à jour les structures correspondantes.
- Si *C*=2 : *i* doit pointer sur *j* car c'est lui la nouvelle racine, *i* met à jour les structures de l'arborescence racine affectées par ce changement.
- Si *C*=3 : Les structures de données correspondantes à l'arborescence distribution sont mises à jour de tel sorte que *i* pointe sur *j*.
- Si *C*=4 : Quand un nœud *j* perd son dernier lien sortant, il inverse quelques liens pour avoir des chemins vers les détenteurs des deux jetons ; dans ce cas, il doit envoyer *ReverseLinkInfo*(4,*LevelVar*$_j$) à ses voisins. En recevant ce message, *i* compare son niveau avec celui de *j* :

    - S'ils sont les mêmes (resp, *i* se trouve dans un niveau inférieur à *j*), alors il rajoute *j* dans *SameSetRoot*$_i$ (resp, *InSetRoot*$_i$) et le supprime de *OutSetRoot*$_i$ (resp, *OutSetKey*$_i$) mais cela peut engendrer la perte du dernier lien sortant de *i* vers le nœud qui détient le jeton racine (resp, le jeton clé). Dans ce cas, *i* exécute la procédure d'inversement de liens RearrangeRoot(*i*) (resp RearrangeKey(*i*)). Il choisi un autre *NextRoot*$_i$ (resp *NextKey*$_i$) si son ancienne valeur était *j*. Le nœud *i* n'a pas à relancer des requêtes même si Q_i n'est pas vide car elles vont comme même être satisfaites (pas de rupture physique de lien).

- Dans le cas où *i* se trouve dans un niveau supérieur à celui de *j*, deux cas se présentent, soit *j* appartenait à $SameSetRoot_i$ et dans ce cas il suffit de changer son emplacement vers $InSetRoot_i$; soit *j* appartenait à $OutSetRoot_i$ et dans ce cas il suffit seulement de changer la valeur de son niveau.

***Pseudo-code* (*i* reçoit *ReverseLinkInfo* (c, *LevelVarj*))**

```
When i receives ReverseLinkInfo(c,LevelVar_j) from j
Do
    Case c
        1: //j a reçu les deux jetons
            Add (j, OutSetRoot_i); NextRoot_i := j;
            If j ∈ SameSetRoot_i then Remove(j,SameSetRoot_i);
            Else Remove(j,InSetRoot_i);
            EndIf
            InSetKey_i := InSetRoot_i ;OutSetKey_i := OutSetRoot_i ;
            SameSetKey_i:= SameSetRoot_i ;NextKey_i := Nextroot_i ;
     2: //j a reçu le jeton racine
            Add (j, OutSetRoot_i); NextRoot_i := j;
            If j ∈ SameSetRoot_i then Remove(j,SameSetRoot_i);
            Else Remove(j,InSetRoot_i);
            EndIf
     3: //j a reçu le jeton clé
          InSetKey_i := InSetRoot_i ; OutSetKey_i := OutSetRoot_i ;
          SameSetKey_i:= SameSetRoot_i ; NextKey_i := Nextroot_i ;
     4: //j a perdu son dernier lien sortant
            If LevelVar_i = LevelVar_j then
                Add(j, SameSetRoot_i) ;
                Remove(j,OutSetRoot_i) ;
                If (NextRoot_i = j) then
                   If (OutSetRoot_i = Φ) then   RearrangeRoot() ;
                   Else  NextRoot_i :=Choose(OutSetRoot_i);
                   EndIf // ne pas relancer des requêtes même si Q_i n'est pas vide (pas de rupture
                   physique de lien) car elles vont comme même être satisfaites.
                EndIf
            Else
               If LevelVar_i < (LevelVar_j) then
                  Add(j, InSetRoot_i) ;
                  Remove(j,OutSetRoot_i) ;
                  If (NextRoot_i = j) then
                     If (OutSetRoot_i = Φ) then   RearrangeRoot() ;
                     Else   NextRoot_i :=Choose(OutSetRoot_i);
                     EndIf
                  EndIf
               Else //forcément LevelVar_i > LevelVar_j
                  If j ∈ SameSetRoot_i then
                     Add(j, InSetRoot_i) ;
                     Remove(j,SameSetRoot_i);
                  Else
                     SetLevel(j,LevelVar_j, OutSetRoot);
                  EndIf
               EndIf
            EndIf
    EndCase
EnDo
```

4.6 Discussion

Nous essayons dans ce qui suit d'aborder les différents points cachés du protocole et de dégager certains aspects de force ou de faiblesse des solutions utilisées.

1- Lors du déplacement de jeton racine, hormis le cas de perte du dernier lien sortant, un nœud n'est pas amené à envoyer des requêtes pour les éléments de sa file étant donné que ces requêtes seront satisfaites par le détenteur actuel du jeton clé. D'autre par, le déplacement du jeton clé signifie que toutes les requêtes sont satisfaites et les nouvelles requêtes sont récoltées par le détenteur du jeton racine, elles vont être satisfaites lors de l'arrivée du jeton clé à ce nœud.

2- Les nœuds qui changent de position et changent en conséquence leur chemin vers la racine et qui ont demandés des ressources, doivent envoyés de nouvelles requêtes à leur nouveau *NextRoot*, car leur ancien *NextRoot*, en détectant la rupture de lien, supprime toutes les requêtes venant d'eux.
Donc, la question qui se pose est quel est le sort des anciennes requêtes ? Pour cela, deux solutions se présentent :

- Dans la première, (celle que nous avons adoptée), seul le nœud qui a détecté la rupture de lien supprime les requêtes. Mais il y aura un moment où ces requêtes seront satisfaites par la réception de ce même nœud d'un message *Resources*(*r*) ou *TokenRoot* ou bien *AllTokens*. Si ce nœud a reçu *Resources*(*r*) -rappelons que *Resources*(*r*) est normalement le message qui satisfait la requête supprimée- alors, il compare le paramètre "*r*" du message reçu avec le paramètre "*r*" de la requête qui est en tête de file, il trouve qu'elle n'est pas la même alors il libère ces ressources en envoyant *Release*(*r*). Il se peut que comme par hasard la valeur du "*r*" correspond à la valeur du "*r*" de la requête qui est en tête de file ; dans ce cas, le nœud envoie *Resources*(*r*) au nœud chanceux de tête de sa file. En recevant *TokenRoot,* ce nœud envoie le jeton racine à l'élément de fin de sa file sans se soucier du reste. En recevant *AllTokens* , il n'y aura aucune requête dans sa file (puisqu'il a supprimé la requête du destinataire de ce message) ; dans ce cas, il garde les deux jetons jusqu'à ce qu'il reçoive une nouvelle requête.

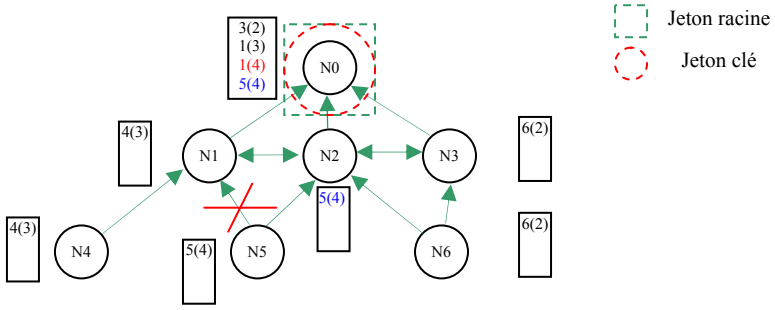


Figure 21 : Le sort des anciennes requêtes après un Linkdown .

Reprenons l'exemple de la figure 14 avec une rupture du lien reliant le nœud *N5* à *N1*.
En détectant la rupture de lien avec *N5*, le nœud *N1* supprime la requête de ce dernier de sa file, mais le nœud *N0* ne va pas supprimer cette requête et va la satisfaire à un moment donné. En recevant les quatre ressources demandées par *N5*, le nœud *N1*, consulte sa file, il ne trouve pas le destinataire ; alors, il va libérer ces ressources.

- Dans la deuxième solution, le nœud qui a détecté la rupture de lien et qui a supprimé les requêtes venant de ce voisin, envoie un message d'information à son *NextRoot* lui demandant de supprimer lui aussi les requêtes concernées et ce même *NextRoot* va envoyer ce message à son *NextRoot* et ainsi de suite jusqu'à ce que ce message arrive à la racine. Dans cette solution, un risque apparaît, lorsque ce message de demande de suppression croise le message de ressources ou de jetons en réponse à la requête supprimée.

3- Quand un nœud sort de la section critique et trouve plus d'une requête dans sa file, alors il envoie le jeton racine au dernier de sa file et garde le jeton clé ; dans ce cas, l'arborescence racine change alors que l'arborescence distribution reste la même. Quand le nœud distributeur termine de satisfaire les requêtes et atteint la requête du nouveau nœud racine, il lui envoie le jeton clé. Au fur et à mesure du déplacement du jeton clé les deux arborescences logiques deviennent identiques jusqu'à ce qu'il arrive à la racine, dans ce cas les deux arborescences racine et distribution se confondent.

4.7 Quelques directives liées à l'extension de la solution

1- Comportement du protocole en cas de panne de nœud : Si le nœud en panne n'est ni la racine ni le nœud distributeur, c-à-d, il ne possède aucun jeton, il n'y aura aucun problème et le protocole réagit comme si c'était le cas de ruptures de liens. Cependant, si le nœud en panne détient l'un des deux jetons, le nœud qui possède l'autre jeton peut le régénérer et une simple réorganisation de l'arborescence correspondante est faite. Toutefois, si le nœud en panne possède les deux jetons, alors, une procédure d'élection doit être exécutée et les deux jetons sont régénérés par le nœud élu. Une réorganisation des arborescences doit être faite.

2- Comment savoir que le nœud avec lequel le lien est rompu est détenteur du jeton ?
Nous pouvons supposer comme solution que le nouvel nœud racine (resp distributeur) envoie un message d'information à ses voisins leur informant qu'il détient le/les jetons (en cas de formation de lien, le nœud racine/distributeur envoie ce message à son nouveau voisin). Une fois son traitement terminé et doit transmettre le/les jetons à un autre nœud, il envoie un autre message d'information à ses voisins leur avisant qu'il n'est plus le nœud racine (resp distributeur). De cette manière, un nœud peut savoir si son voisin avec lequel le lien est rompu possède ou non le/les jetons. Notons que les nœuds intermédiaires (ceux chez lesquels le jeton n'est pas stationnaire) ne sont pas concernés par cette solution.

Cette solution peut aider à résoudre quelques problèmes tels que la panne du nœud racine (resp distributeur), la détection d'un partitionnement ou d'un fusionnement, néanmoins elle provoque un nombre supplémentaire de messages.

3- Comment détecter la perte du/des jeton(s) : Le problème de perte de jeton revient au problème de pannes de nœuds. Donc, si le nœud détenteur d'un jeton x tombe en panne, les nœuds voisins, en détectant la rupture de leurs liens avec lui, envoient un message d'alarme en direction du détenteur de l'autre jeton y. C-à-d, ces messages d'alarme circulent dans l'arborescence dans laquelle, le jeton y est encore présent dans le système. Au fur et à mesure que ces messages avancent, ils signalent la perte du jeton x et la réorganisation des deux arborescences se fait par le biais des messages d'informations diffusés par les nœuds qui reçoivent les messages d'alarme. Enfin, lorsque le nœud détenteur du jeton y reçoit le premier message d'alarme, il régénère le jeton x, il sera donc racine et distributeur en même temps et les deux arborescences deviennent confondus en une seule. Si le nœud détenteur des deux jetons tombe en panne, la détection et la signalisation de la perte des deux jetons se font de la

même manière que précédemment. Dans ce cas, plusieurs politiques d'élection peuvent être adoptées : par exemple, les voisins de l'ancienne racine sont les seuls candidats, ils peuvent négocier la responsabilité d'être le nouveau détenteur des deux jetons, du fait qu'ils étaient les plus proches de l'ancienne racine. Dans ce cas, il y aura moins de messages de réorganisation de la structure que si tous les nœuds soient candidats. Parmi ces voisins, le nœud élu peut être choisi soit :

- Par le critère du niveau, c-à-d, le voisin qui à la plus petite valeur de niveau. Remarquons qu'il était l'avant dernier détenteur de jetons.
- Par le critère du nombre de voisins, c-à-d, le nœud qui a le plus de voisins devient la nouvelle racine afin de minimiser le nombre de messages de réorganisation et d'optimiser le nombre de messages *Request, Resources* et *Release* futures.

4- Dans les circonstances de *partitionnement* du réseau, le réseau se trouve diviser en deus partitions ou plus, chaque partition doit continuer à exécuter normalement le protocole de la h-k exclusion mutuelle. Cette division du réseau est transparente à l'ensemble des nœuds, En effet, au sein d'une même partition, les différents nœuds du système voient le problème de partitionnement comme celui de pannes de nœuds (racine/distribution ou les deux).
Dans le phénomène de partitionnement deux cas se présentent :

1- Dans le premier cas : une partie du réseau contient les deux jetons et l'autre aucun des deux ; dans ce cas, la première partie continue d'exécuter le protocole normalement et la deuxième partie exécute d'abord une procédure d'élection afin de régénérer les deux jetons.
2- Dans le deuxième cas : chacune des deux parties prend un jeton, de ce fait, chaque nœud possédant un jeton, régénère le deuxième jeton manquant et chaque partie réorganise l'arborescence correspondante.

5- Comment détecter le fusionnement ?
En cas de formation de lien, les deux nœuds nouvellement connectée doivent donner l'un à l'autre l'identificateur du nœud racine. Si l'un des nœuds est racine ou voisin d'un nœud racine, il transmet directement cette information à son nouveau voisin, sinon, il envoie une requête à son *Next* lui demandant l'identificateur de la racine, cette requête suit le chemin jusqu'au voisin de la racine. L'information ainsi récupérée, elle va suivre le même chemin retournant vers le nouveau voisin. En échangeant cette information, deux cas se présentent :

- Les deux nœuds appartiennent à la même arborescence (pas de fusionnement) ;
- Les deux nœuds appartiennent à deux arborescences différentes et le fusionnement est détecté.

Dans le cas de *fusionnement* de deux réseaux : le réseau correspondant à ce fusionnement se voie détenir deux jetons racine et deux jetons clé et le double du nombre de ressources. Dans ce cas, chaque nœud diffuse un message de suspension du service de la h-k exclusion mutuelle le temps de reconstruire la nouvelle arborescence. Afin d'avoir le moins de messages possibles, la partition qui contient le moins de nœuds mobiles doit renoncer à ses deux jetons et s'intégrer à l'autre partition par la réorganisation de ses structures d'arborescences logiques.

6- Comment régénérer le nombre exact de ressources dans le système ?
Les problèmes de pannes de nœuds, de partitionnement et de fusionnement induisent un autre problème, à savoir, celui du nombre de ressources dans le système. Nous pouvons donner comme solution ce qui suit :
Le nouveau nœud distributeur ne procède pas à la satisfaction des requêtes avant de restituer toutes les ressources. Pour cela, on peut utiliser un *timer*. Le nouveau nœud distributeur

attend une durée de temps déterminée pendant laquelle il soit sûr que toutes les ressources sont à son niveau (cela s'applique aussi pour un ancien nœud distributeur qui a attendu cette même durée afin de satisfaire une requête pour laquelle des ressources manquent). Ensuite, il vérifie leur compte, trois cas se présentent :

- Le compte est manquant (le cas de pannes de nœuds ou de partitionnement) ;
- Le compte dépasse le nombre exacte de ressources du système (cas de fusionnement) ;
- Le compte est bon.

Dans les deux premiers cas, il restitue le nombre exact de ressources et continue à servir les requêtes.

A cause du *timer* utilisé, cette solution augmente le temps d'attente des nœuds pour une entrée en section critique.

4.8 Conclusion

Dans ce chapitre, nous avons présenté une solution au problème de la h-k exclusion mutuelle dans les réseaux mobiles ad hoc. En fait, mis à part des problèmes connus dans les réseaux statiques, les réseaux mobiles ad hoc se caractérisent par la mobilité, l'absence d'infrastructure de base, une bande passante et une autonomie énergétique limitées… et ceci ne fait que rendre plus difficile la conception des protocoles de la h parmi k exclusion mutuelle.

Le protocole que nous avons proposé dans ce chapitre se base sur une structure logique de niveaux divisée en deux arborescences logiques ; Racine et Distribution. Cette structure permet de s'adapter aux changements de la topologie du réseau et essaye de minimiser le nombre de messages échangés pour transmettre le ou les jetons. En outre, le principe de niveaux permet une organisation logique des nœuds en sous-ensembles et cette organisation logique suit systématiquement la topologie physique du réseau. Le protocole proposé ne fait pas appel à la couche de routage. Il est basé sur deux jetons ; Racine et Clé, chacun appartient à l'une ou l'autre arborescence. Le service de la h-k exclusion mutuelle entretient de multiples chemins vers le détenteur de jeton afin de minimiser la restructuration de l'arborescence. Le nœud qui possède le jeton racine collecte les requêtes qui sont mises dans une file d'attente suivant l'ordre de leurs arrivées. Le nœud qui possède le jeton clé détient les ressources disponibles dans le réseau et se charge de les distribuer. Le nœud racine ne peut entrer en section critique que s'il a le jeton clé et lorsqu'il sort de la section critique il envoie le jeton racine à l'élément de fin de sa file et garde le jeton clé jusqu'à ce qu'il termine de satisfaire toutes les requêtes de sa file. Entre temps, le nouveau nœud racine collecte les nouvelles requêtes sans qu'il puisse les satisfaire pour le moment, car le jeton clé ne lui est pas encore parvenu. Lorsque le détenteur du jeton clé termine de satisfaire toutes les anciennes requêtes et lui reste que la requête de la nouvelle racine, il lui envoie le jeton clé.

La mobilité des nœuds a fait partie intégrante de la solution proposée grâce au mécanisme d'inversement de liens en cas d'une défaillance de liens. Ce mécanisme d'inversement de liens est utilisé aussi en cas de déplacement de jeton, car la racine doit être dans le niveau le plus bas ; de plus, les chemins vers la racine doivent rester décroissants (du plus haut niveau jusqu'à la racine).

Enfin, la performance de tout protocole est conjointement liée à des contraintes relatives à l'environnement externe comme la mobilité, la charge du système, le nombre de nœuds participants… etc. Où seule une simulation avancée pourra l'évaluer. Ceci fera l'objet du prochain chapitre dans lequel on va essayer d'analyser le comportement du protocole élaboré en conjonction avec les paramètres précédemment énumérés.

Chapitre 5

Etude de simulation

5.1 Introduction

La preuve de correction d'un algorithme et l'évaluation de ses performances devrait se faire avec des outils formels tels que les outils mathématiques, les réseaux de pétri, les chaînes de Markov, etc… La simulation ne donne pas une preuve de correction formelle mais, permet non seulement de tester un algorithme sans aucun coût 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 tels 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 performance. 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.

Nous avons utilisé un outil de simulation de réseau appelé NS2 *(Network Simulator 2)* [TEM00]. Ce logiciel a été développé dans un contexte académique au sein du *LBL*, du *Xerox PARC*, de l'*UCB* et de *USC/ISI* (c'est cette dernière institution qui maintient le code actuellement). NS2 est un simulateur à événements discrets permettant l'étude et la conception de protocoles pour les réseaux d'ordinateurs, il permet d'exécuter tout type de scénario sur des topologies définies par l'utilisateur. Il propose plusieurs facilités pour la simulation des protocoles basés sur *TCP*, le routage et la multi-distribution dans les réseaux (avec ou sans fil). Ce simulateur est un logiciel libre à code source ouvert. NS2 se compose d'une interface de programmation en *tcl* et d'un noyau écrit en *C++* dans lequel la plupart des protocoles réseaux ont été implémentés. Le développement des protocoles s'effectue en *C++* et en *OTcl* (évolution objet de *TCL*) et les scénarios sont décrits en *OTcl*.

La simulation de notre protocole sous l'environnement NS2 passe par son implémentation et la réalisation des tests préliminaires afin de dégager toute anomalie liée à la programmation. La prochaine étape permet de préparer les différentes simulations en définissant les modèles de simulation et les scénarios de mobilité.

Ce chapitre présente les différentes simulations réalisées ainsi que les résultats obtenus. Il se compose de sept sections. La première, introduit l'étude de simulation. La seconde présente les différentes structures et classes qui constituent le programme. La section 3 expose les paramètres pris en considération dans ces simulations tels que la mobilité, la charge, etc… Ensuite, vient la section 4 qui donne la définition de chaque critère de performances à évaluer. Les résultats des différentes simulations sont présentés et interprétés dans la section 5 selon les données initiales et les scénarios d'exécution générés. Nous terminons se chapitre par la section 6 qui contient une discussion et une conclusion qui permettront de faire la synthèse du comportement du protocole et des résultats obtenus.

5.2 Implémentation du protocole de la h-k exclusion mutuelle

Notre protocole a été implémenté sous l'environnement NS2 version 2.30 avec le système d'exploitation Linux Suse version 10.1. Le protocole UDP a été employé pour la transmission des paquets de messages. Le langage *OTCL* a principalement été utilisé en tant que langage de programmation et a permis la création d'un ensemble de classes dont chaqu'une couvre un aspect de la simulation. Parfois, on fait recours à certaines classes prédéfinies afin de combler certains aspects de cette implémentation. Les classes développées dans cette simulation sont :

La classe eNode : Contient les variables de description d'un nœud mobile (par exemple : le status, la liste des voisins, le niveau, etc…) ;
La classe Node : Contient l'ensemble des méthodes relatives aux mouvements des nœuds en termes de formations et ruptures de liens et en termes de son activité locale tel que l'envoi d'une requête, l'entrée et la sortie de la section critique, l'insertion d'un nouveau voisin, etc… Nous pouvons citer par exemple :

- EnterCS : Entrer en SC ;
- LeaveCS : Sortir de la SC ;
- Request : Demande d'entrée en SC ;
- ReverseLink : Mise à jour des structures de données relatives aux nœuds mobiles suite à une formation ou une rupture de lien ou à une réception d'un des deux jetons.

La classe Msg_Agent : Définit un agent UDP qui permet la réception des paquets des messages suivants au niveau de la couche transports :

- Le message *Init* : Débute la construction de l'arborescence ;
- Le message *EndInit* : Détermine la fin de la structure logique et le commencement de l'exécution de l'algorithme de la h-k exclusion mutuelle ;
- Le message *ReqCS* : Demande de la SC ;
- Le message *Resources* : Satisfait une requête sans avoir de jetons ;
- Le message *Release* : Libère des ressources ;
- Le message *AllTokens* : Envoie des deux jetons à savoir, le jeton racine et le jeton clé ;
- Le message *TokenRoot* : Envoie le jeton racine ;
- Le message *TokenKey* : Envoie le jeton clé ;
- Le message *ReverseLinkInfo* : Mis à jour les structures de données suite à un ReverseLink.
- Le message *DeleteReq* : Supprime les requêtes inutiles existantes dans les files d'attente des requêtes (en cas de changement d'itinéraire vers le détenteur du jeton racine).

Les scénarios de mobilité ont été générés indépendamment du programme. Cependant, nous y avons injecté un code d'appel à une méthode '*LinkState'* à chaque mouvement d'un nœud afin de déterminer l'état des liens de ce nœud et mettre à jour ses nouveaux liens.

5.3 Environnement de la simulation

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

5.3.1 Paramètres de mesures

- La charge : Elle correspond au délai entre le moment où un nœud quitte sa section critique et celui de la formulation d'une nouvelle demande. Les requêtes d'entrée en section critique sont déterminées par un processus de poisson avec un taux d'arrivée λ. Par conséquent, la charge correspond à une variable aléatoire exponentielle avec une moyenne de $1/\lambda$ unités de temps. Pour notre simulation, les valeurs prises pour λ sont : 0.06, 0.10, 0.12, 0.16, 0.25, 0,33, 0.5, 1 correspondant respectivement aux intervalles $(1/\lambda)$: 15s, 10s, 8s, 6s, 4s, 3s, 2s, 1s, durant lesquels un nœud, après satisfaction, génère une nouvelle demande d'entrée en section critique.

- La mobilité : La définition de la mobilité que nous avons adoptée est celle proposée par Tony Larson et Nicholas Hedman [LAH98]. Elle est basée sur le mouvement relatif des nœuds. Cette définition donne une bonne estimation de la manière dont les nœuds se meuvent les uns par rapport aux autres. La définition est la suivante : Si les nœuds sont en mouvement pour une certaine durée de temps, alors la mobilité est la moyenne des changements de distances entre tous les nœuds.
  Les valeurs de mobilité retenues sont données par :
  - Mobilité nulle (la mobilité moyenne égale 0) : Dans ce cas, les nœuds sont soit stationnaires soit dans un état de mouvement qui n'affecte pas les liens ni en terme de formations ni en terme de ruptures.
  - Mobilité faible (la mobilité moyenne est supérieure à 0 et inférieure ou égale à 3): Dans ce cas, le nombre varie entre 2 et 11 changements de liens par unité de temps ;
  - Mobilité moyenne (la mobilité moyenne est supérieure à 3 et inférieure ou égale à 8) : Dans ce cas, le nombre varie entre 12 et 30 changements de liens par unité de temps ;
  - Mobilité élevée (la mobilité moyenne est supérieure à 8) : Ici, le nombre de changements de liens est supérieur à 30 changements par unité de temps.

  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 forte mobilité.

- Le nombre de ressources : Un autre paramètre intrinsèque au protocole, à savoir, le nombre de ressources du système, pour lequel une valeur doit être fixée a été introduit étant donné que sa variation influence considérablement les performances du protocole. Nous avons considéré 10, 20 et 30 ressources.

- La connectivité : Elle est calculée en terme de pourcentage des liens existant par rapport au nombre de liens possibles dans le graphe. Nous avons considéré les connectivités suivantes : 5%, 25%, 50%, 75% et 100%.

Les scénarios de mouvements que nous avons utilisés ont été générés avec l'outil « *setdest* » (générateur aléatoire de scénarios) 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 simulation seront exécutés suivant le fichier ainsi généré.

5.3.2 Paramètres généraux

Ce sont des paramètres nécessaires au déroulement de la simulation :
La surface de mouvement : 1000 x 500 mètres ;
Nombre de nœuds dans le réseau : 25 ;
Le nombre de ressources dans le réseau : 10 ;
Le rayon de communication entre nœuds mobiles : 250 mètres ;
Le temps de simulation : 100 secondes.

5.3.3 Ajustement de certains paramètres

- Durée de construction de l'arborescence : Notre algorithme débute par la construction de la structure logique qui est l'arborescence de niveaux. Par conséquent, une durée de temps

doit être considérée pour cette construction ; c-à-d, la première requête pour la section critique ne peut être envoyée qu'après cette durée. En réalisant plusieurs scénarios, on a opté pour la valeur de 0.5 secondes ;
- Durée d'exécution de la section critique : 0.1 secondes ;
- Durée de transfert de messages : 0.1 secondes.

5.4 Critères de performances

Ce sont les métriques que nous avons évalué en tenant compte des paramètres déjà cités.

5.4.1 Temps moyen d'attente par entrée en section critique : C'est le nombre moyen d'unités de temps écoulées entre la demande d'accès en section critique et l'accès lui-même. Cette métrique est calculée par :

$$\text{Temps d'attente / ESC} = \frac{\sum (\text{Temps d'ESC}_i - \text{Temps Req}_i)}{\text{Nombre d'entrées en SCs}}$$

C'est le cumul de tous les temps durant lesquels les nœuds demandeurs de la section critique ont attendus pour y entrer, divisé par le nombre total d'entrées en section critique et cela durant toute la durée de la simulation.

Les temps durant lesquels les nœuds demandeurs de la section critique ont attendus pour y entrer peuvent être obtenus par la différence entre le temps où un nœud *i* est entré en section critique (Temps d'ESC_i) et le temps durant lequel il a lancé sa demande (Temps Req_i).

5.4.2 Nombre moyen de messages générés par entrée en section critique : Cette métrique représente le nombre moyen de messages nécessaires pour la réalisation d'une section critique. Elle est calculée de la façon suivante :

$$\text{NbMsg / ESC} = \frac{\text{Nombre de tous les messages}}{\text{Nombre d'entrées en sections critiques}}$$

Le nombre moyen de messages par entrée en section critique (NbMsg / ESC) est le rapport du nombre total de messages avec le nombre total d'entrées en section critique.

5.4.3 Délai de synchronisation : C'est le temps moyen écoulé en termes de nombre moyen de messages générés entre deux entrées successives en section critique.
Dans notre protocole, trois cas se présentent pour qu'un nœud mobile puisse entrer en section critique, soit en recevant le message *Resources*, soit en recevant les deux jetons, soit en recevant le jeton clé.

Dans le premier cas, cette métrique est calculée de la manière suivante :

$$\text{DélaiSyn} = \frac{\text{Nombre de msg } \textit{Resources} + \text{ Nombre de msg } \textit{Release}}{\text{Nombre d'entrées en SCs par le msg } \textit{Resources}}$$

En quittant la section critique, un nœud libère les ressources utilisées. Le délai de synchronisation enregistré quand le prochain nœud entre en section critique par la voie des messages *Resources*, est calculé par la somme des deux types de messages (*Release* et *Resources*) divisé par le nombre total d'entrée en section critique au moyen des messages *Resources*.

Dans le cas de *AllTokens*, cette métrique est calculée comme suit :

$$\text{DélaiSyn} = \frac{\text{Nbr msg } \textit{AllTokens} + \text{Nbr msg } \textit{Release} + \text{Nbr msg } \textit{ReverseLinkInfo}}{\text{Nombre d'entrées en SCs par le msg } \textit{AllTokens}}$$

Le calcul du délai de synchronisation, dans le cas où le prochain nœud entre en section critique en recevant les deux jetons, prend en considération les messages d'inversement de liens liés aux déplacements des deux jetons. Donc, le délai de synchronisation est la somme des trois types de messages (*AllTokens, Release* et *ReverseLinkInfo*) divisé par le nombre total d'entrées en section critique au moyen du message *AllTokens*.

Dans le cas de *TokenKey*, le délai de synchronisation est calculé comme suit :

$$\text{DélaiSyn} = \frac{\text{Nbr msg } \textit{TokenKey} + \text{Nbr msg } \textit{Release} + \text{Nbr msg } \textit{ReverseLinkInfo}}{\text{Nombre d'entrées en SCs par le msg } \textit{TokenKey}}$$

Le calcul du délai de synchronisation, dans le cas où le prochain nœud entre en section critique en recevant le jeton clé, prend en considération les messages d'inversement de liens liés aux déplacements du jeton clé. Donc, le délai de synchronisation est la somme des trois types de messages (*TokenKey, Release* et *ReverseLinkInfo*) divisé par le nombre total d'entrées en section critique au moyen du message *TokenKey*.

Dans le cas général, le délai de synchronisation est calculé comme suit :

$$\text{DélaiSyn} = \frac{\text{msg } \textit{AllTokens} + \text{msg } \textit{TokenKey} + \text{msg } \textit{Resources} + \text{msg } \textit{Release} + \text{msg } \textit{ReverseLinkInfo}}{\text{Nombre d'entrées en SCs}}$$

Le calcul du délai de synchronisation dans le cas général, prend en considération les messages d'inversement de liens liés aux déplacements des deux jetons et aux déplacements du jeton clé plus la somme des deux types de messages *Release* et *Resources*. Donc, le délai de synchronisation global est la somme des cinq types de messages (*AllTokens, TokenKey, Resources, Release* et *ReverseLinkInfo*) divisé par le nombre total d'entrées en section critique.

5.5 Réalisation des tests et interprétation des résultats

Afin d'évaluer les performances de notre protocole, nous avons procédé à la réalisation d'un certain nombre de tests. Dans ce qui suit, nous allons présenter les résultats obtenus des

différentes simulations en considérant séparément chacun des cas de mobilité (cas statique, faible, moyenne et forte mobilité).

Pour chacun des critères de performances suscités, nous présentons le graphe des courbes correspondant aux mesures effectuées. L'interprétation de ces mesures permettra ensuite d'expliquer ou de justifier les valeurs obtenues et de montrer l'influence de la variation de certains paramètres sur ces mesures tout en se basant sur le comportement de l'algorithme.

Nous avons établi un graphe pour chacune des quatre mobilités, élevée, moyenne et faible et en prenant en considération bien sur le cas statique. A la première vue de ces graphes, il est important de souligner l'existence de quatre faisceaux différents de courbes chacun correspond à l'une des mobilités. Le premier (avec un petit losange) correspond au cas statique, celui avec un carré correspond à la faible mobilité, le faisceau avec un triangle représente la mobilité moyenne et enfin celui avec une croix, c'est la forte mobilité qui est représentée.

5.5.1 Influence de la variation de la charge sur les performances

Les différentes simulations ont permis de dégager un comportement général du protocole en fonction de la charge :

- **Temps d'attente moyen pour une entrée en section critique**

Parmi les mesures que nous avons réalisées, nous retrouvons celles permettant de mesurer le temps moyen d'attente d'un nœud pour avoir les ressources demandées par rapport à la charge de demandes (figure 22).

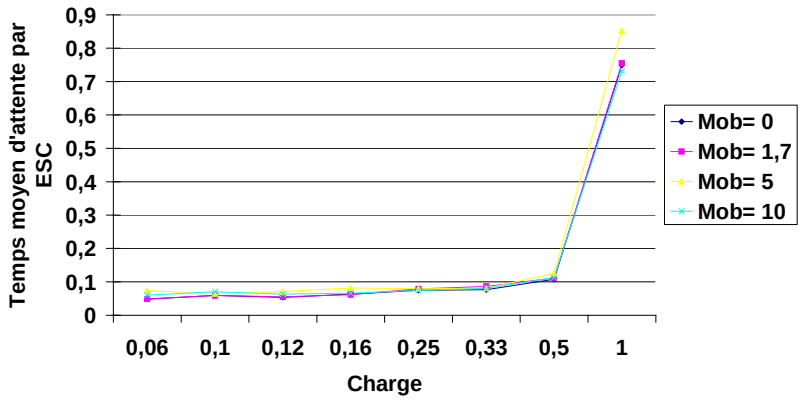


Figure 22 : Temps moyen d'attente pour une ESC.

Les graphes correspondants au temps d'attente moyen pour une entrée en section critique, obtenus à partir des mesures effectuées, ont une allure plutôt croissante ce qui exprime que plus la charge augmente plus les performances diminuent. Il est à constater que l'écart entre les courbes n'est pas très grand sauf le cas de forte charge qui a l'air de s'écarter lorsque la charge est de 1s ; A cette même charge, les trois courbes montrent un pic.

Comme le montre la figure 22, le temps d'attente moyen pour une entrée en section critique est en croissance continue durant les six premières valeurs de la charge (jusqu'à une demande chaque 3s), par la suite une croissance rapide de ce temps est enregistrée à partir de

la septième valeur de la charge (une demande chaque 2s). Cet accroissement du temps d'attente revient essentiellement aux manques de ressources disponibles, car avec une forte charge de demandes, un nœud doit attendre la libération des ressources utilisées par les autres nœuds pour que sa requête soit satisfaite. Cela exprime une forte activité du système et peut expliquer les valeurs de performances obtenues. Dans les cas de faible charge, les zones d'activités du système sont séparées. En outre, dans une même zone d'activité, chaque nœud n'a droit qu'à une et une seule requête à formuler ; ainsi, le système est "soulagé" ; ce qui peut expliquer que les performances soient meilleures.

L'impact du taux de mobilité, tel que montré par la figure 22, n'est pas du tout ressenti du fait que le temps d'attente augmente ou diminue suivant le nombre de ressources disponibles dans le système (Voir résultats en fonction du nombre de ressources).

- **Nombre de messages moyen pour une entrée en section critique**

Dans cette mesure, nous nous sommes intéressés au nombre moyen de messages générés pour une entrée en section critique dépendamment de la charge des demandes d'entrée en section critique.

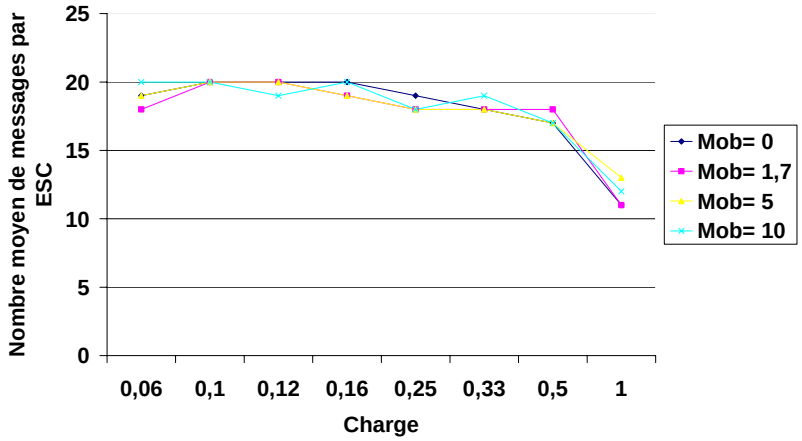


Figure 23 : Nombre moyen de messages par ESC.

Les quatre courbes représentant le nombre de messages moyen pour une entrée en section critique (figure 23) semblent avoir une allure décroissante avec un écart très réduit.

La réduction du nombre de messages est conjointement liée au temps moyen d'attente pour une entrée en section critique. Ceci dit, plus la charge des demandes est importante, plus le nombre de ressources ne suffit plus et plus le temps d'attente augmente, mais dès que les ressources se libèrent, les processus peuvent entrer en section critique. Dans ce cas, mis à part les messages de libération de ces ressources (*Release*) et ceux de satisfaction des demandes (*Resources*) qui sont enregistrés, on n'aura aucun autre type de message qui circule dans le réseau. Par conséquent, le nombre moyen de messages pour une entrée en section critique diminue et on conclue que ce nombre moyen de messages est "inversement proportionnel" au taux de charges.

Nous pouvons conclure aussi, que la mobilité a une influence minime sur le nombre de messages. Elle influe principalement sur le chemin des jetons, des messages (*Resources*) et

des messages (*Release*) par les ruptures des routes. Par ailleurs, l'influence de la charge est remarquable.

- **Délai de synchronisation**

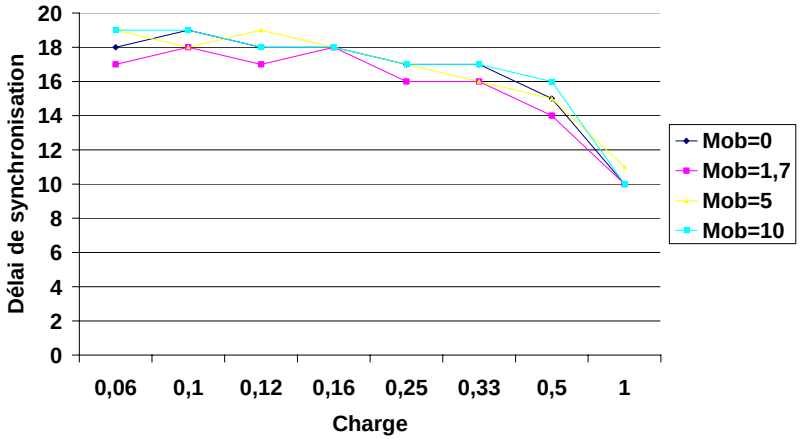


Figure 24 : Délai de synchronisation.

Les courbes obtenues des mesures effectuées concernant la métrique de délai de synchronisation (Figure 24) (quelque soit la mobilité) ont une allure décroissante, ce qui signifie que plus forte est la charge, meilleures sont les performances. Ceci peut être expliqué par l'augmentation du temps d'attente à défaut de ressources. En effet, l'insuffisance de ressources engendre une stagnation du système à ce moment, c-à-d, qu'on n'aura aucun message mis à part les messages d'information sur le voisinage. Dès la libération de ressources, le système marque une activité réduite en terme de messages, il sert seulement les nœuds en attente jusqu'à épuisement des ressources libérées puis marque "une pose" jusqu'à une nouvelle libération.

Par ailleurs, nous constatons que les performances dans le cas statique sont meilleures que celle du cas mobile, ceci s'explique par le fait de la mobilité elle-même qui, plus intense est-elle, plus elle génère de messages de type *ReverseLinkInfo*. Néanmoins, les performances obtenues sont acceptables.

Les cas de la mobilité moyenne et la forte mobilité n'ont pas modifié le comportement général du protocole. En effet, les courbes obtenues quelque soit la performance en question ont gardées la même allure avec une légère baisse des performances, ce qui est prévisible.

5.5.2 Influence de la variation du nombre de ressources sur les performances

Afin de montrer l'influence de la variation du nombre de ressources sur les performances, nous avons effectué des simulations avec un système respectivement à 10, 20, 30 ressources. Nous avons construis des graphes pour chacune des métriques à évaluer dans le cas statique et dans chaque cas de mobilité (faible, moyenne et forte mobilité). Nous tenons à préciser que le nombre de ressources demandées par les processus reste réduit (un nœud peut demander d'une à dix ressources pas plus). De se fait, l'influence de la variation du nombre de ressources est clairement ressentit. Autrement, c-à-d, si le nombre de ressources demandé suit le nombre de ressources présent dans le système, les résultats seront généralement semblables.

Les résultats qui vont suivre permettent de vérifier une propriété prévisible selon laquelle l'augmentation du nombre de ressources va servir le protocole de la h-k exclusion mutuelle dans le sens de l'amélioration des performances.

a- Cas de réseau statique

- **Temps d'attente moyen pour une entrée en section critique**

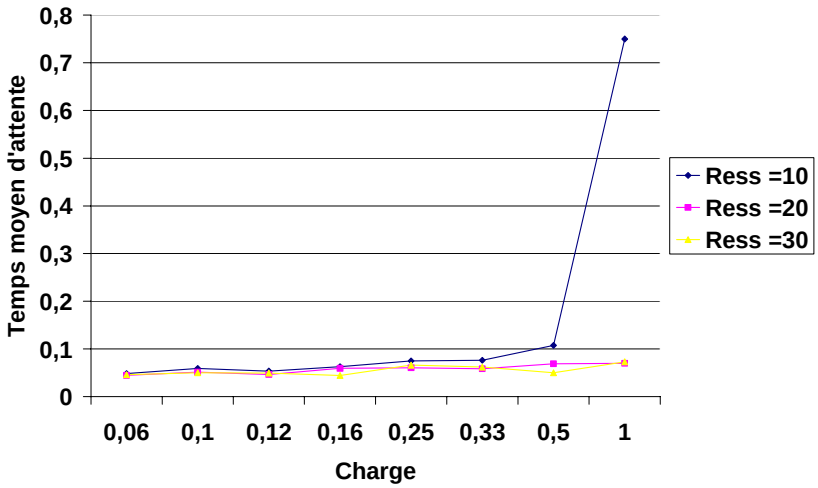


Figure 25 : Temps d'attente pour une ESC(avec variation du nombre de ressources).

Les courbes issues des simulations (figure 25) montrent une croissance du temps d'attente proportionnellement à l'augmentation de la charge et la diminution du nombre de ressources. En général, ces courbes ne présentent pas un très grand écart entre elles, même dans le cas où le nombre de ressources est 10 dont les performances ne sont pas si dégradées que prévisible ; elles sont même relativement bonnes, excepté le cas où la charge est élevé (de 0,33 à 1). Dans ce dernier cas, le manque de ressources est clairement ressenti par la divergence de la courbe correspondante et le pic enregistré quand la charge vaut 1. Autrement, dans le cas où le nombre de ressources est 20 ou 30, les performances augmentent considérablement.

En effet, hormis les valeurs correspondants aux fortes charges, les valeurs des mesures sont tellement proches que les courbes obtenues ont l'air de se confondre et de prendre une allure presque horizontale, ce qui peut laisser penser que certaines ressources, dans le cas où le système dispose d'un grand nombre de ressources (soit 20 ou 30) ne sont pas bien exploitées dans le sens où certaines ressources restent toujours disponibles. Cependant, cette dernière conclusion n'est pas vraie. En effet, en observant le graphe issu des mesures sur le nombre moyen de messages par entrée en section critique en fonction de la charge (figure26), nous remarquons que les courbes représentant les mesures effectuées avec 20 et 30 ressources s'écartent en présentant de meilleures performances.

En conclusion, dans le cas statique, le temps d'attente pour une entrée en section critique avec peu de ressources est plus important lorsque la charge est forte.

- **Nombre de messages moyen pour une entrée en section critique**

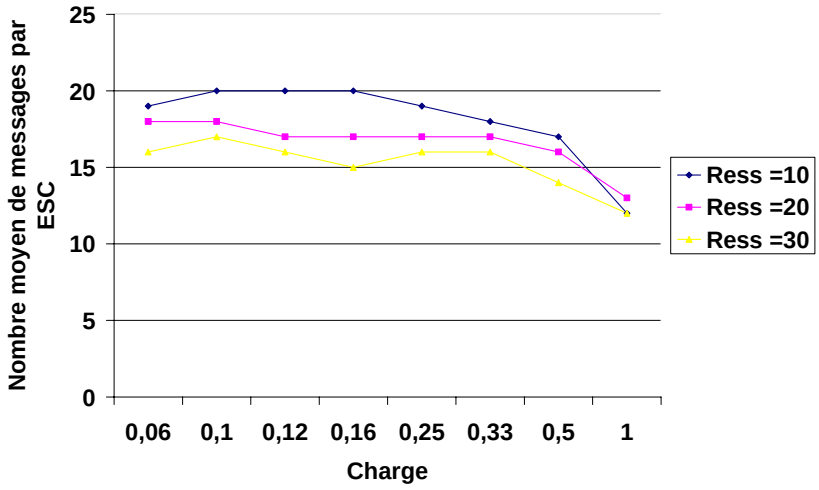


Figure 26 : Nombre de messages pour une ESC (avec variation du nombre de ressources).

Les courbes obtenues (figure 26) présentent une allure décroissante. En effet, avec suffisamment de ressources, la file des requêtes du nœud racine est presque toujours vide ; car chaque nouvelle requête est satisfaite immédiatement. De plus, lorsque le nœud racine sorte de sa section critique, il envoie les deux jetons (un seul message) au premier nœud lui demandant des ressources. Par contre, à défaut de ressources, plus de messages sont engendrés à cause des déplacements individuels de chaque jeton (jeton racine puis jeton clé).

L'écart entre les courbes enregistré pour les faibles et moyennes charges, peut être expliqué aussi par le fait qu'avec plus de ressources, on aura plus d'entrées en section critique, ce qui diminue le nombre moyen de messages.

Pour les forte charge, nous remarquons que les courbes se rapprochent. En réalité, c'est le rapport (nombre total de messages sur le nombre d'entrées en section critique) qui nous donne ces valeurs. Cela est expliqué par le fait d'avoir plus de ressources, on aura plus d'entrées en section critique, ce qui engendre plus de messages.

En conclusion, le nombre de messages par entrée en section critique est inversement proportionnel au nombre de ressources ce qui est prévisible.

- **Délai de synchronisation**

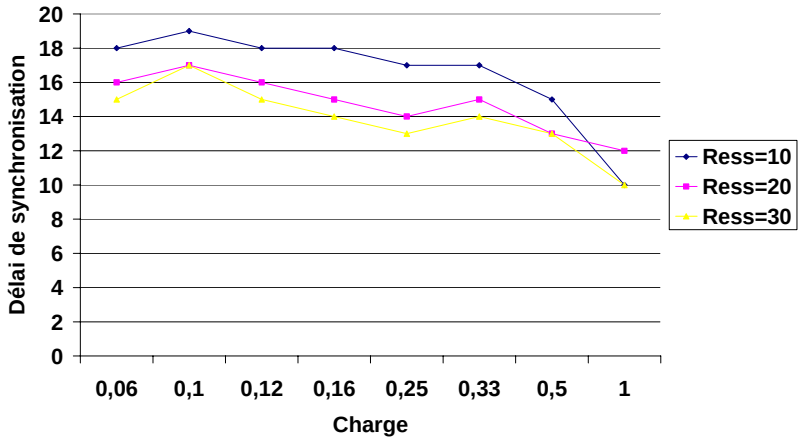
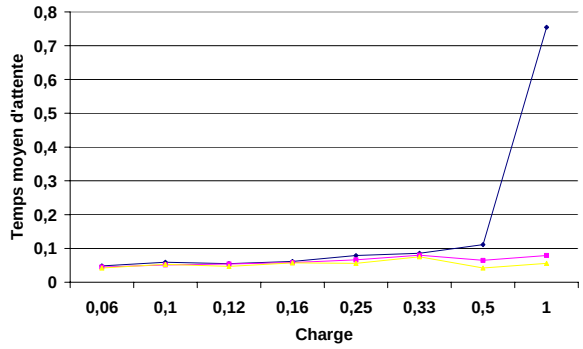


Figure 27 : Délai de synchronisation (avec variation du nombre de ressources).

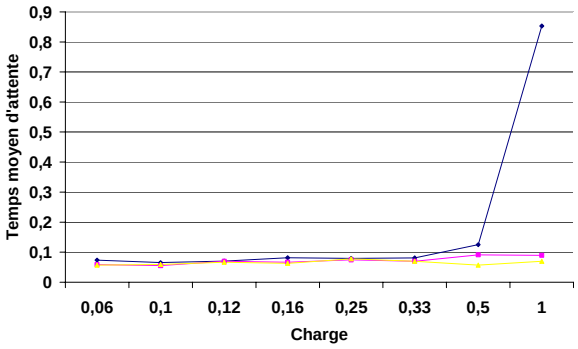
Les courbes donnant le délai de synchronisation en fonction de la charge (figure 27) présentent une allure décroissante et montre clairement que les performances du protocole deviennent meilleures lorsque le nombre de ressources augmente notamment pour le cas de forte charge.

b- Cas de réseau dynamique

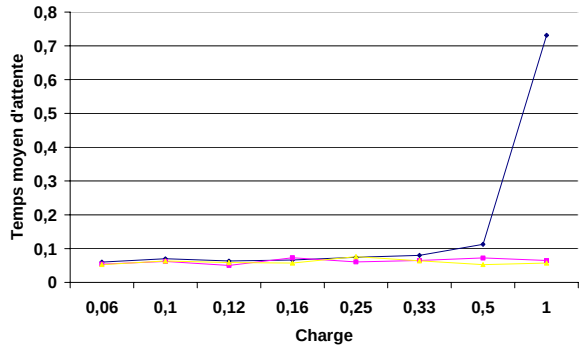
- **Temps d'attente moyen pour une entrée en section critique**



(a) Mobilité faible (Mob=1,7)



(b) Mobilité moyenne (Mob=5)



(c) Mobilité forte (Mob=10)

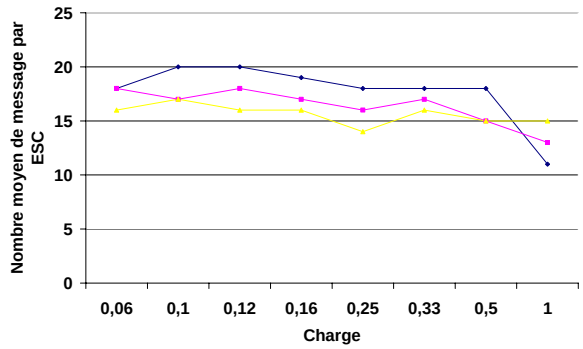
Figure 28 : Temps d'attente pour une ESC (avec variation du nombre de ressources).

L'influence de la variation du nombre de ressources est plus évidente dans le graphe des temps d'attente par entrée en section critique en fonction de la charge (figure 28), où plus le système dispose de ressources moins un nœud attend pour entrer en section critique, ce qui est prévisible. Dans le cas de 20 et 30 ressources, les performances sont bonnes de par le fait que le système est fortement actif avec disponibilité suffisante de ressources. En effet, en faisant varier le nombre de ressources, on remarque que les courbes restent relativement proches dans le cas de faible et moyenne charge. Néanmoins, on constate comme même une amélioration en terme de temps d'attente. Dans le cas de forte charge, l'augmentation du nombre de ressources donne des résultats satisfaisants.

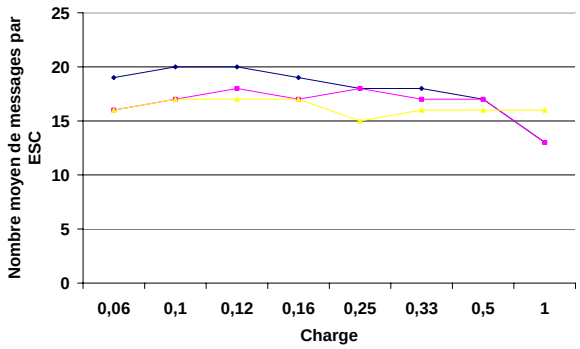
Les courbes issues des mesures effectuées dans le cas de faible et moyenne mobilité sont comparables à celles des mesures effectuées dans le cas de forte mobilité.

- **Nombre de messages moyen pour une entrée en section critique**

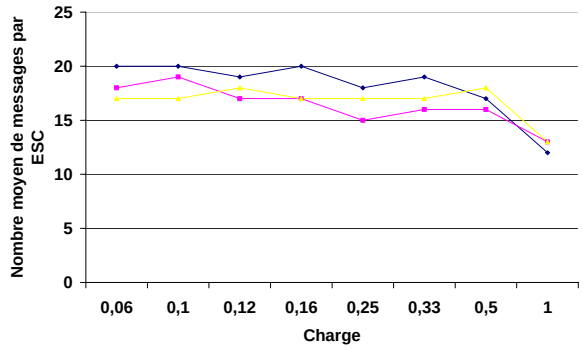
Concernant le calcul du nombre moyen de messages nécessaires pour l'acquisition de ressources et l'entrée en section critique, des mesures nous ont permis d'établir une évaluation concernent son évolution illustrée par les deux figures 29 et 30. Notons que dans la figure 29, pour chaque type de mobilité, un graphe est établit. Il représente le nombre moyen de messages par entrée en section critique en fonction de la charge et du nombre de ressources. Dans cette figure, nous voulons mettre en évidence l'influence du nombre de ressources sur les performances pour chaque type de mobilité. En ce qui concerne la figure 30, trois valeurs du nombre de ressources sont considérées, et pour chaque valeur, un graphe est établit, il représente le nombre moyen de messages par entrée en section critique en fonction de la charge et de la mobilité. La figure 30 permet de comparer les différentes courbes représentants les quatre types de mobilité issues de l'augmentation du nombre de ressources dans le système.



(a) Mobilité faible (Mob=1,7)



(b) Mobilité moyenne (Mob=5)



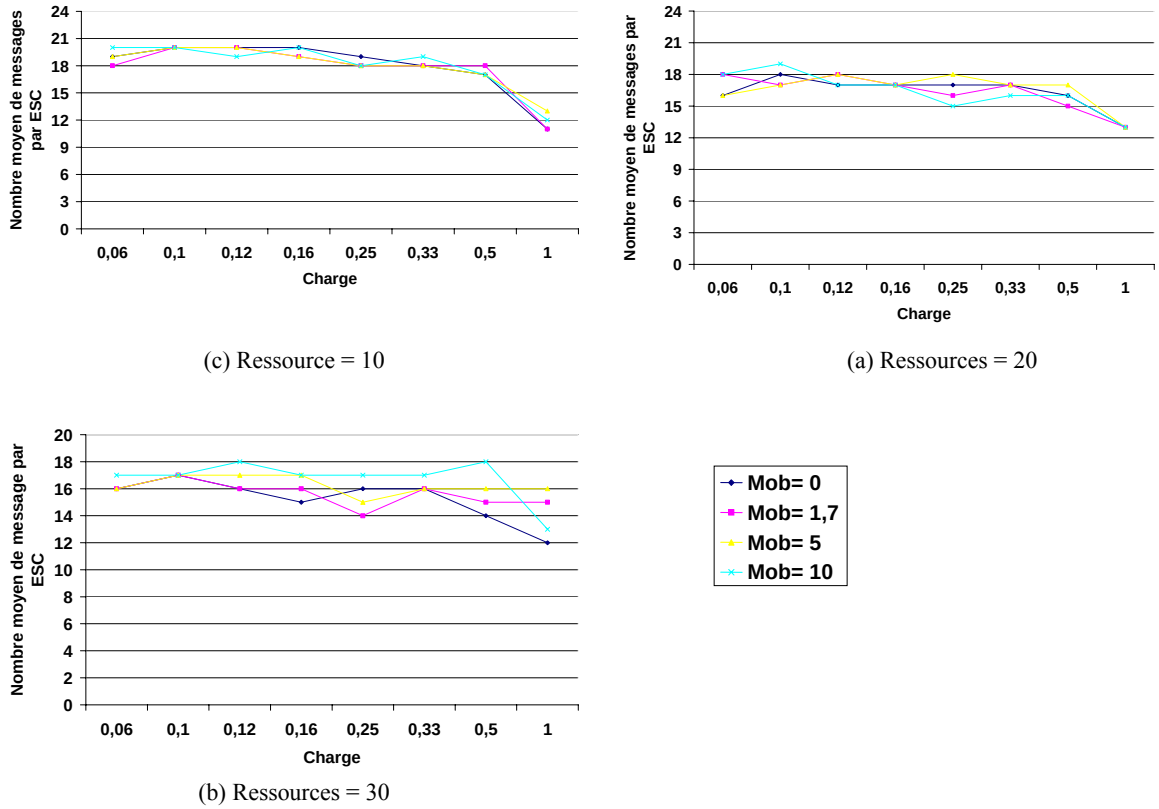
(c) Mobilité forte (Mob=10)

Figure 29 : Nombre de messages pour une ESC (avec variation du nombre de ressources).

L'allure des courbes de la figure 29 est décroissante. En effet, dans les trois cas de mobilité, les graphes ont la même allure décroissante. Dans les trois figures 29.a ; 29.b et 29.c nous constatons que plus le nombre de ressources augmente plus le nombre moyen de messages diminue, ce qui est prévisible. En outre, il est à noter que les mesures du nombre moyen de messages dans les cas de fortes charges sont meilleures en général que dans les cas de faibles et moyennes charges. Effectivement, les cas de 20 et 30 ressources dans le système offrent relativement de faibles performances dans le cas de faibles et moyennes charges, et de bonnes performances dans les cas de fortes charges. On remarque même que dans le cas de fortes charges, les performances offertes par le cas de 20 ressources dépassent celles des cas

où le système dispose de 30 ressources. Ceci s'explique par le fait qu'avec 30 ressources le système reste toujours actif et les nœuds consomment moins de temps pour attendre les ressources demandées, ce qui induit à plus de messages de redemandes de la section critique.

L'impact du taux de mobilité, tel que montré par la figure 30, a été ressenti dans l'accroissement du nombre de messages calculé qui accompagne l'augmentation de la mobilité. De cette manière, le nombre de voisins peut augmenter comme il peut diminuer avec un mouvement plus rapide des nœuds mobiles, ce qui permet d'adresser plus de messages de requêtes et d'information du voisinage pour la réorganisation de la structure logique.



(c) Ressource = 10

(a) Ressources = 20

(b) Ressources = 30

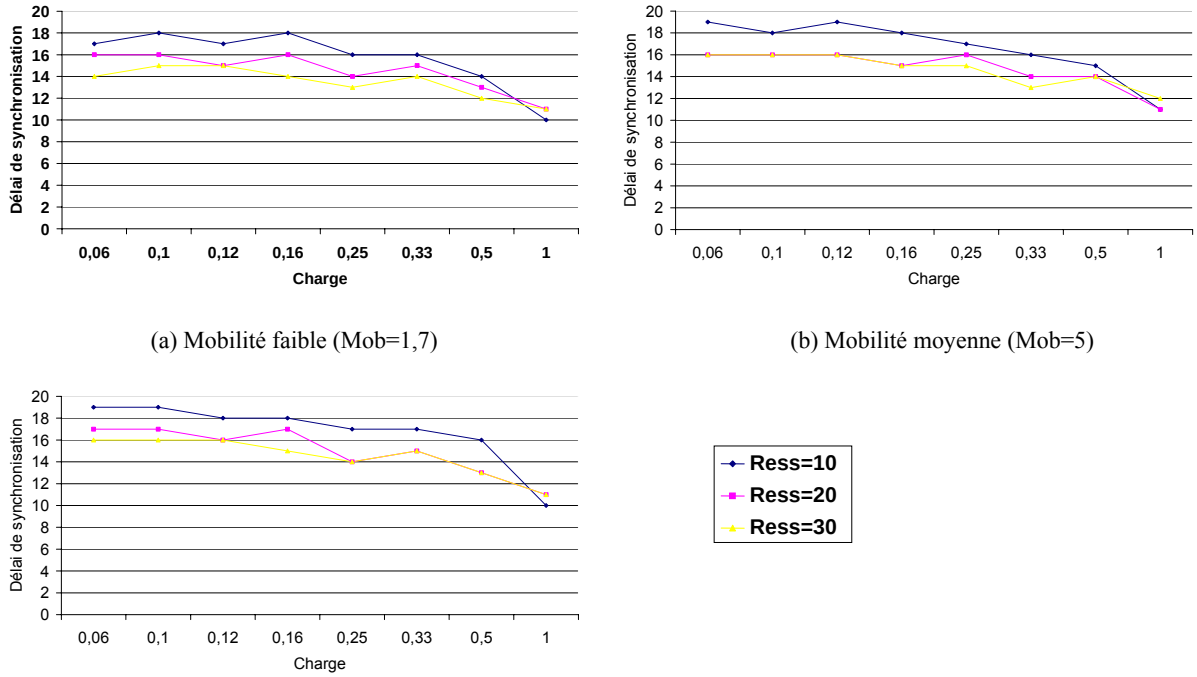
Figure 30 : Nombre moyen de messages pour une ESC en fonction de la mobilité.

Afin de comparer les différentes courbes représentant les quatre types de mobilité issues de l'augmentation du nombre de ressources dans le système, nous avons repris le graphe de la figure 23 où le nombre de ressources égale à 10.

Avant de comparer les trois graphes, notons qu'en terme de nombre de messages par entrée en section critique, il est important de souligner que les valeurs de mesures obtenues dans tous les cas appartiennent à un intervalle de valeurs réduit.

Dans le graphe représentant 10 ressources, il n'y a pas un grand écart entre les courbes en faisant varier le type de mobilité. L'allure de celles-ci est décroissante avec un léger avantage des conditions de fortes charges par rapport aux faibles et moyennes charges. Les graphes représentants 20 et 30 ressources donnent de meilleurs résultats que celui représentant 10 ressources, ce qui est prévisible, et fait ressentir l'impact de la mobilité sur les performances.

- **Délai de synchronisation**



(a) Mobilité faible (Mob=1,7)

(b) Mobilité moyenne (Mob=5)

(c) Mobilité forte (Mob=10)

Figure 31 : Délai de synchronisation (avec variation du nombre de ressources).

En faisant varier le nombre de ressources, nous remarquons qu'il y a un écart entre les courbes qui symbolise une amélioration du délai de synchronisation. L'allure de celles-ci est décroissante quelle que soit le type de mobilité avec un léger rapprochement des valeurs pour la forte charge (figure 31.a, 31.b et 31.c).

Au final, nous pouvons conclure que les performances du protocole deviennent meilleures en augmentant le nombre de ressources.

5.5.3 Influence de la variation de la connectivité sur les performances

Les différentes simulations ont permis de dégager un comportement général du protocole en fonction de la connectivité (charge fixe = 0,25 ; 4s) :

- **Temps d'attente moyen pour une entrée en section critique**

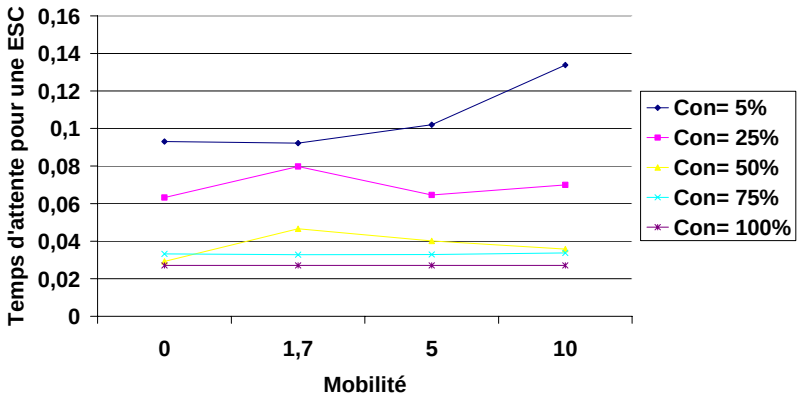


Figure 32 : Temps d'attente pour une ESC en fonction de la connectivité.

Il est à constater, à partir de la figure 32 que l'influence de la connectivité sur les temps moyens d'attente pour une entrée en section critique est évidente, car l'écart entre les courbes représentants ce dernier est bien apparent. Ce qui signifie que plus forte est la connectivité, meilleurs sont les délais d'attente. Nous remarquons aussi, que les temps d'attente ont plus ou mois des valeurs constantes quelle que soit la mobilité quand la connectivité est forte 75% et 100%. Ceci s'explique par le fait que malgré que les nœuds se déplacent, ils restent tous ou presque tous voisins.

- **Nombre de messages moyen pour une entrée en section critique**

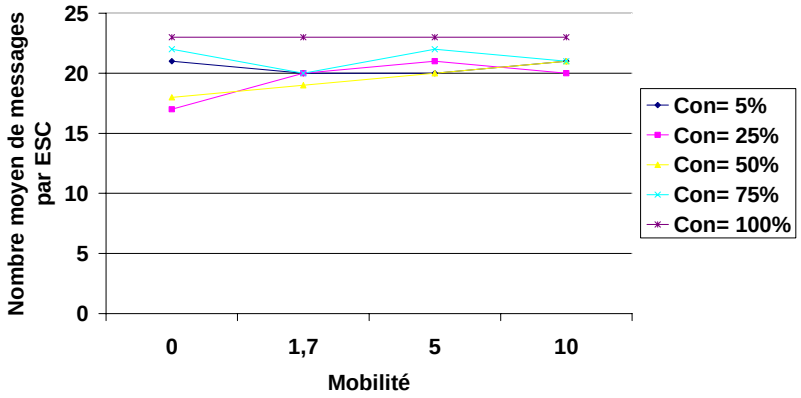


Figure 33 : Nombre moyen de messages pour une ESC en fonction de la connectivité.

La figure 33 montre que les performances diminuent en augmentant le taux de la connectivité à partir du taux 50%. En effet, en augmentant le taux de la connectivité nous aurons plus d'entrées en section critique, ce qui engendre plus de messages. De plus, La complexité de messages a une relation inversement proportionnelle avec le temps d'attente. Quand le délai d'attente diminue, l'augmentation du nombre de messages par entrée en section critique est enregistrée et cela malgré la forte connectivité. En outre, pour le taux de connectivité 100% par exemple, l'augmentation du nombre de messages moyen par entrée en section critique peut être expliquer par le fait que tous les nœuds du réseau sont connectés au nœud racine, et à chaque déplacement d'un des deux jetons, tous les (N-1) nœuds reçoivent des messages d'inversement de liens. D'après la figure 33, nous constatons aussi que le taux de connectivité 50% donne les meilleures performances.

L'impact du taux de mobilité, tel que montré par la figure 3, est ressenti dans l'accroissement du nombre de messages calculé qui accompagne l'augmentation de la mobilité. En effet, plus le taux de connectivité diminue, plus il y aura de messages de ruptures et de formations de liens.

- **Délai de synchronisation**

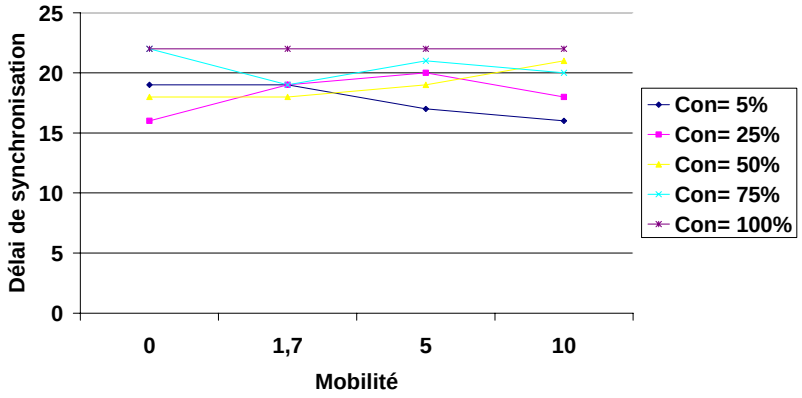


Figure 34 : Délai de synchronisation en fonction de la connectivité.

Il est à constater, à partir de la figure 34, que l'influence de la connectivité n'est pas ressentie du fait que le délai de synchronisation est plus important lorsque le temps d'attente est faible.

5.6 Discussion et conclusion

L'objectif de l'étude de simulation est d'étudier le comportement du protocole devant des situations variées et d'évaluer ses performances relativement à ses propres caractéristiques. Après toutes les mesures effectuées afin d'évaluer les performances du protocole, il ressort un certain nombre de constatations relatives aux comportements du protocole.

Les courbes représentant la métrique « délai d'attente » en fonction de la charge ont en général une allure plutôt stable pour tous les cas de mobilité dans les conditions de faible et moyenne charge ; cette allure s'élève progressivement pour s'accentuer en pic quand la charge atteint la valeur la plus élevée "1". Nous avons expliqué cette dégradation des performances par le manque de ressources. En effet, les courbes reprennent leur allure stable lorsque le nombre de ressources augmente.

Les courbes représentant la métrique « nombre de messages » en fonction de la charge ont en général une allure décroissante pour tous les cas de mobilité. Ceci montre que notre protocole réagit de manière efficace même dans les conditions de forte charge. Cependant, en augmentant le nombre de ressources, les résultats sont meilleurs, ce qui est prévisible. Cependant, on remarque que dans les conditions de forte charge, les résultats obtenus avec moins de ressources sont meilleures que ceux avec un nombre élevé de ressources. Ceci s'explique par le fait qu'avec un nombre suffisant de ressources le système reste toujours actif et les nœuds consomment moins de temps pour attendre les ressources demandées, ce qui induit plus de messages de redemandes de la section critique.

Les comportements observés durant les différents tests de simulation nous ont permis de déduire une dépendance immédiate du temps moyen d'attente, du délai de synchronisation et du nombre moyen de messages pour une entrée en section critique vis-à-vis du nombre de

ressources disponibles dans le système. En général et dans tous les cas, l'augmentation du nombre de ressources est en faveur du protocole en terme de performances.

Le protocole se montre bien adapté à la mobilité car cette dernière n'influe pas sur ses performances. Cependant, dans le cas de forte mobilité, nous constatons une légère dégradation des performances comparée à celles du cas de faible mobilité. Ces dernières étant moins bonnes que celles du cas statique. Il est à noter que : étant donné que notre protocole se base sur une structure d'arborescence logique, le maintien de cette structure génère des messages additionnels surtout dans le cas de forte mobilité mais les complexités de mesures restent acceptables.

Finalement, les résultats que nous avons obtenus sont généralement acceptables. Néanmoins, nous considérons qu'ils sont préliminaires et servent principalement à donner une idée globale sur le fonctionnement du protocole face à des situations réelles. En effet, une conclusion précise et objective ne peut être énoncée que si le protocole est testé avec une variété plus grande de situations. Il s'agit de confectionner des jeux d'essais combinant l'ensemble des paramètres qui influencent son fonctionnement. A titre d'illustration nous citons :

- Faire varier le nombre de nœuds pour se rendre compte des limites du protocole ;
- Faire en sorte que le nombre de nœuds soit variable : par exemple en ajoutant de nouveaux nœuds dans le système de manière dynamique ;
- Faire varier le nombre de ressources afin de trouver l'intervalle qui fait que le protocole donne les meilleures performances.

En outre, chaque combinaison doit donner lieu à plusieurs simulations moyennant des scénarios de mobilité variés.

Conclusion générale

Les réseaux mobiles sans infrastructure ou réseaux mobiles ad hoc se caractérisent par le changement rapide et fréquent de la topologie d'une manière imprévisible et par les ressources modestes des unités mobiles. De ce fait, le problème lié aux services d'allocations de ressources dans ce type de réseau est un problème complexe car les protocoles d'allocations de ressources dans un tel environnement doivent tenir compte de la variation de la topologie, le partitionnement éventuel du réseau et la limitation des sources d'énergie dans les unités mobiles. Ces contraintes font que les protocoles d'allocations de ressources doivent générer une charge faible en termes de nombre de messages et de quantité d'énergie consommée.

Notre intérêt a porté particulièrement sur le problème de la h-k exclusion mutuelle dans les réseaux mobiles ad hoc; ce problème est aussi connu par le problème d'allocation de h parmi k ressources. Il consiste à travers un certain nombre de mécanismes à contrôler les nœuds, dans un système distribué, lorsque chacun désire accéder à h ressources parmi la totalité des k ressources partagées, $1 \leq h \geq k$, avec la contrainte de ne pas accéder concurremment à plus de k ressources.

Pour mieux cerner le problème de la h- k- exclusion mutuelle, nous étions amené à présenter quelques algorithmes d'exclusion mutuelle qui ont été proposés pour résoudre le problème d'accès concurrent à une seule ressource, à k ressources et à h parmi k ressources dans les systèmes distribués. Nous avons décris leurs principales caractéristiques et fonctionnalités ainsi que leurs avantages et inconvénients. Nous avons donc présenté les principales solutions apportées à tous ces problèmes vu la relation qui existe entre certains d'entre elles (adaptation pour l'environnement et adaptation pour le nombre de ressources) et aussi vu les liens étroits qui existent avec la h-k exclusion mutuelle dans les réseaux mobiles ad hoc.

Les algorithmes présentés dans ce document donnent de bons résultats pour certains cas et moyens dans d'autres. De plus, on constate que tous les algorithmes proposés afin de résoudre les problèmes de la k-exclusion mutuelle et de la h-k exclusion mutuelle aussi bien dans les réseaux statiques que dans les réseaux mobiles ad hoc sont une adaptation d'autres algorithmes qui peuvent être considérés comme des algorithmes de base.

Notre solution s'écarte de toutes celles qui la précèdent ; elle ne généralise ni n'adapte aucune autre solution. Elle est fondée sur deux principes : *racine* et *distribution*. Cette solution utilise deux structures logiques d'arborescence basées sur la structure physique du réseau mobiles ad hoc baptisées racine et distribution. La première est utilisée pour récolter les requêtes des ressources ; l'autre est utilisée pour satisfaire ces requêtes en distribuant les ressources demandées et en récoltant les ressources libérées. Ces structures logiques sont organisées selon le principe de niveaux. A un moment donné, le plus bas niveau constitue le sommet de l'arborescence racine, il contient un seul nœud appelé le nœud racine et chaque nœud dans le réseau a un chemin vers chaque sommet de l'une ou l'autre des deux arborescences. Chaque arborescence, que ce soit racine ou distribution est dotée d'un jeton qui est maintenu toujours par le nœud qui est au sommet de l'arbre correspondant. Donc, nous avons deux jetons, le jeton *racine* au sommet de l'arborescence racine et le jeton *clé* au sommet de l'arborescence distribution. Le nœud qui détient le jeton racine se charge de récolter les demandes de ressources. Le nœud qui détient le jeton clé se charge de distribuer les ressources demandées et de récolter les ressources libérées. Les autres nœuds se chargent

de transmettre les messages dans l'arborescence correspondante. En outre, le service de la h-k exclusion mutuelle entretient de multiples chemins vers le détenteur de jeton afin de minimiser la restructuration de l'arborescence.

Etant donné le changement fréquent de la topologie physique, les structures logiques utilisées doivent être mises à jour. Pour se faire, nous avons utilisé la technique d'inversement partiel de liens logiques pour garantir la continuité et la sûreté du protocole de la h-k exclusion mutuelle proposé.

Notre protocole a été implémenté et simulé sous l'environnement NS2 version 2.30. Afin d'évaluer ses performances, nous avons définis quelques critères de performances ; à savoir, le nombre de messages générés par entrée en section critique, le temps d'attente et le délai de synchronisation.

L'observation des courbes issues des mesures sur le temps d'attente moyen pour une entrée en section critique ont une allure plutôt croissante; ce qui exprime que plus la charge augmente plus les performances diminuent. Le comportement général du protocole montre que dans le cas de forte charge de demandes d'accès en section critique, toutes les ressources ou presque sont utilisées, les nœuds qui utilisent ces ressources sont en section critique et les autres processus sont en attente et ce durant tout le temps de la simulation ; ce qui exprime une forte activité du système et peut expliquer les valeurs de performances obtenues. Dans les cas de faible charge, les zones d'activités du système sont séparées. En outre, dans une même zone d'activité, chaque nœud n'a droit qu'à une et une seule requête à formuler ; ainsi, le système est "soulagé" ; ce qui peut expliquer que les performances soient meilleures.

Les courbes obtenues des mesures effectuées concernant la métrique de complexité de messages (quelque soit la mobilité) ont une allure décroissante, ce qui signifie que plus forte est la charge, meilleures sont les performances. Ceci peut être expliqué par l'augmentation du temps d'attente à défaut de ressources.

Par ailleurs, nous constatons que les performances dans le cas statique sont meilleures que celle du cas mobile ; ceci s'explique par le fait de la mobilité elle-même qui, plus intense est-elle, plus elle génère de messages de type *ReverseLinkInfo*. Néanmoins, les performances obtenues sont acceptables. Les cas de la mobilité moyenne et la forte mobilité n'ont pas modifié le comportement général du protocole. En effet, les courbes obtenues quelque soit la performance en question ont gardées la même allure avec une légère baisse des performances, ce qui est prévisible. En outre, les courbes obtenues des mesures effectuées concernant toutes les métriques en fonction de la connectivité montrent que plus forte est la connectivité, meilleures sont les performances.

En résumé, et d'après tous les résultats de simulation obtenus, il devient clair que notre protocole fournit des résultats acceptables et prévisibles par rapport aux concepts et outils mis en œuvre pour son implémentation.

Comme travaux futurs, on peut citer les points suivants :

- Tester le comportement du protocole par rapport à la scalabilité du réseau ;
- Rendre le protocole tolérant au panne de nœuds, au partitionnement et fusionnement du réseau ;
- Construire une arborescence dans un réseau dynamique ;
- Adapter le protocole de tel sorte qu'il puisse résoudre le problème de l'exclusion mutuelle simple et de la K-exclusion mutuelle dans les réseaux mobiles ad hoc et examiner l'apport de ces adaptations vis à vis de celles existantes dans la littérature.

Bibliographie

[AEE97]: Divyakant Agrawal, Ömer Egecioglu, Amr El Abbadi, 'Analysis of Quorum-Based Protocols for Distributed (k + 1)-Exclusion'. IEEE Transactions on Parallel and Distributed Systems, May 1997.

[AGA91]: D. Agrawal et A. E. Abbadi, "An Efficient and Fault-Tolerant Solution for Distributed Mutual Exclusion," ACM Trans. Computer Systems, Vol. 9, No. 1, pp. 1-20, February 1991.

[BAC94]: Roberto Baldoni et B. Ciciani, "Distributed algorithms for multiple entries to a critical section with priority". Information Processing Letters, 50:165–172, 1994.

[BAL94]: Roberto Baldoni. « An $O(n^{M=(M+1)})$ distributed algorithm for the k-out of-m resources allocation problem ». In The 14th International Conference on Distributed Computing Systems, pages 81–88, 1994.

[BMR95]: Roberto Baldoni, Yoshifumi Manabe, Michel Raynal, Shigemi Aoyagi, 'k-Arbiter: A safe and general Scheme for h-out of-k mutual exclusion problems' N° 2523, rapport de recherche, April 1995.

[BUV95]: S. Bulgannawar and N. H. Vaidya. 'A distributed K-*mutual exclusion algorithm'*. In Proceedings of the 15th International Conference on Distributed Computing Systems, pages 153-160. IEEE, 1995.

[BVP02]: R. Baldoni, A. Virgillito, R. Petrassi, "A Distributed Mutual Exclusion Algorithm for Mobile Ad-Hoc Networks", In Proc of the 7th IEEE Symposium on Computer and Communications (ISCC 20002), pp 539-545, Toarmun, Italy 1-4 July 2002.

[CAR83]: O. S. F. Carvalho, Gérard Roucairol "On Mutual exclusion in Computer Networks", Communications of the ACM, Vol 26, num 2, pp 146--147 1983

[CHC97]: Ye-In Chang Bor-Hsu Chen , 'An extended binary tree quorum strategy for K-mutual exclusion indistributed systems' Proceedings of the 1997 Pacific Rim International Symposium on Fault-Tolerant Systems, 15-16 Dec 1997.

[CSL90]: Y.I. Chang, M. Singhal, M. T. Liu, "An Improved Mutual Exclusion Algorithm for Distributed Systems," proc 1990 Int. Conf Parallel processing Vol III, pp 295-302, août 1990.

[CSL90a]: Y.I. Chang M. Singhal and M. Lui "A fault tolerant algorithm for distributed mutual exclusion," Proc. of the 9th Symposium on Reliable distributed Systems October 1990.

[CSL91]: Y. Chang, M. Singhal, M. Lui "A dynamic token-based distributed mutual exclusion," 10th Annual int. Phoenix Conf. On Computer and Communications Mars 1991.

[DHK94]: D.M. Dhandhere, S.S. Kulkarni, 'A token based k-resilient mutual exclusion algorithm for distributed systems' Indian institute of technology, 1997.

[DIJ74]: E.W. Dijkstra, "Self-stabilizing systems in spite of distributed control ", CACM, vol.17, 11(NOV. 1974), 2p.

[FUS97]: S. S. FU "A Comparison of Mutual Exclusion Algorithms for Distributed Memory Systems," News letter at the technical committee of Distributed processing, the IEEE 1997

[FYA91]: Satoshi Fujita, Masafumi Yamashita, and Tadashi Ae. « A non trivial solution of the distributed k-mutual exclusion problem ». ISA '91 Algorithms LNCS 557, 1991.

[GAB81]: Eli M.Gafni, Dimitrip.Bertsekas, '*Distributed algorithms for generating loop-free routs in networks with frequently changing topology'*. IEEE Transactions on communications, Vol.Com-29,N01, January 1981.

[GIF79]: D.K.Gifford, Weighted voting for replicated data. In *Proceedings of 7th Symposium on Operating Systems*, pages 150–162. ACM, 1979.

[GMB85]: Hector Garcia-Molina, Daniel Barbara. "How to assign votes in a distributed system. " Journal of the ACM, 32(4):841–860, October 1985.
[HJK93]: S.T. Huang, J.R. Jiang, and Y.C. Kuo, ***"k** coteries for fault-tolerant **k** entries to a critical section*," in International Conf. Distributed Com**pu**ting Systems, pp. 74-81, 1993.
[HOT01]: A. Housni, M. Tréhel, "Distributed mutual exclusion by prioritized groups based token and permission ", AICCSA (ACS/IEEE International Conference on Computer Systems and Applications), sponsored by ACM SIGART, Beirut, Liban, p. 253-259, 26-29 June 2001.
[HPR88]: J.M. Helary, N. Plouzeau, M. Raynal, "A distributed algorithm for mutual exclusion in an arbitrary network" The Computer Journal August Volume 31 Issue 4, 1988.
[JHK97]: J.R. Jiang, S.T. Huang, Y.C. Kuo, 'Cohorts structures for fault-tolerant k entries to a critical section'. IEEE Transactions on Computers, 1997.
[JIA01]: J-R. Jiang, "A distributed h-out of-k mutual exclusion algorithme using k-coteries, " Technical report,Hsuan Chuang University, 2001.
[JIA01a]: J-R.Jiang, "A Group Mutual Exclusion Algorithme for Ad Hoc Mobile Network," Technical report, Hsuan Chuang University, 2001.
[JIA02]: J-R.Jiang, "A Distributed h-out of-k Mutual Exclusion Algorithme for Ad Hoc Mobile Networks," Proceeding of the International Parallel and Distributed Processing Symposium (IPDPS '02), 2002 IEEE.
[JIA04]: Jehn-Ruey Jiang, 'On The Nondomination of Cohorts Coteries. IEEE Trans. On Computers, 53 (2004) 922-923.
[JIA04a]: Jehn-Ruey Jiang, 'A Fault-Tolerant h-out of-k Mutual Exclusion Algorithm Using Cohorts Coteries for Distributed Systems' **[JIH94]: J.R. Jiang , S.T. Huang,** 'Obtaining Nondominated K-Coteries for Fault-Tolerant Distributed K-Mutual Exclusion', Proceedings of the 1994 International Conference on Parallel and Distributed Systems, p.582-587, December 19-21, 1994.
[KAY96]: Hirotsugu Kakugawa, Masafumi Yamashita, "Local Coteries and a Distributed Resource Allocation Algorithm", Transactions of Information Processing Society of Japan, Vol. 37 No. 8, Aug. 1996.
[KFY93]: H.KAKUGAWA, S.FUJITA, M.YAMASHITA, T.AE, "Availability of k-Coterie," IEEE Transactions on Computers, vol 42, no. 5, mai 1993, pp. 553-558.
[KFY94]: H. Kakugawa, S. Fujita, M. Yamashita, and T. Ae, "*A Distributed k-Mutual Exclusion Algorithm using k-Coterie*," Information Processing Letters, Vol. 49, pp. 213-238 (1994).
[KUM91]: Akhil Kumar. Hierarchical quorum consensus: A new algorithm for managing replicated data. IEEE Transactions on Computers, 40(9):996–1004, September 1991.
[KUO99]: Yu-Chen Kuo, 'Nondominated (K, M)-Coteries for Distributed K-out-of-M Ressources Allocation Problem', SooChow J.Economics and Business Systems, no 27, pp 107-127, Dec 1999.
[KUO01]: Yu-Chen Kuo, 'k-Arbiter Join Operation', Journal of Information Science and Engineering 17,615-631, 2001.
[KUO01a]: Yu-Chen Kuo, 'Composite k-Arbiters', IEEE Transactions on Parallel and Distributed Systems, v.12 n.11, p.1134-1145, November 2001.
[LAM78]: L. Lamport "Time, clocks and the ordering of events in a distributed system", Communications of the ACM, Vol. 21, No. 7, pp. 558-565, July 1978.
[LAN77]: G. Le Lann "Distributed Systems - Towards a Formal Approach". IFIP Congress: pp 155-160. 1977
[LAH98]: Tony Larson, **Nicholas Hedman** 'Routing protocol in wireless ad-hoc A simulation study' Master thesis LULEA TERNISKA university 1998.

[LOK00]: Sandeep Lodha et **Ajay Kashemkayani**, « A Fair Distributed Mutual Exclusion Algorithm ». IEEE Trans. ParallelDistribute Systems Vol. 11, no. 6, Juin. 2000, PP. 537-549.
[MAA93]: Y. Manabe, S. Aoyagi, 'A distributed k-Mutual exclusion Algorithm Using k-coterie", IEICE Technical Report COMP93-43 (Sept.1993).
[MAE85]: M. Maekawa "A √N Algorithm for Mutual Exclusion in Decentralized Systems", ACM Transactions on Computer Systems, Vol. 3, No. 2, pages: 145-159, May 1985.
[MAK94]: K. Makki, "An Efficient Token-based Distributed Mutual Exclusion Algorithm," Journal of Computer and Software Engineering, décembre 1994.
[MAR85]: A.J. Martin "Distributed mutual exclusion on a ring of processors", Science Comuting, Programming, mai 1985.
[MAT99]: Y.Manabe, N.Tajima, "(h-k)-arbiter for h-out of-k Mutual Exclusion Problem," in Proc. Of 1999 IEEE International Conference on Distributed Computing Systemes, pp. 216-223, 1999.
[MBB92]: K. Makki, P. Banta, K. Been, N. Pissinou, and E. Park, "A token based distributed **k** mutual exclusion algorithm," in *IEEE Proceedings of the Symposium on Parallel and Distributed Processing,*pp. **408-411,** December **1992.**
[MBR98]: Y.Manabe, R.Baldoni, M.Raynal, S.Aoyagi,"k-arbiter:a safe and general scheme for h-out of-k mutual exclusion", Theoretical Computer Science, 193(1-2) :97-112, 1998.
[NAI93]: Mohamed Naimi, 'Distributed Algorithm for K-Entries to Critical Section Based on the Directed Graphs'. Operating Systems Review 27(4): 67-75 (1993).
[NEM91]: M. L. Neilsen and M. Mizuno, "A Dag-Based Algorithm for Distributed Mutual Exclusion", Proc. 11ième Int, Conf Destributed Computer systems, pp 354-360 mai 1991.
[NEM94]: Mitchell L. Neilsen and Masaaki Mizuno, "Nondominated k-coteries for multiple mutual exclusion". Information Processing Letters, 50:247–252, 1994.
[NTA96]: M. Naîmi, M. Tréhel and A. Arnold, "A O(log(n)) distributed mutual exclusion algorithm based on path reversal", Parallel and distributed computing, 34 , pp. 1-13. 1996
[NTM87]: M. Naimi, M. Tréhel "An improvement of the log(n) distributed algorithm for mutual exclusion",7th International Conference on Distributed Computing, pages 371-375, 1987.
[OMN02]: **Fatma A.Omara et Mohamed Shams El-Deen Nabil** "An improved Algorithm for the Mutual Exclusion in The Distributed Systems", Proc, Of The First International Conference On Informatics And Systems, Faculty Of Computer and Information, Cairo University, Juin 2002, PP. CS2-1-1-CS2-1-14.
[OMN02a]: **Fatma A.Omara et Mohamed Shams El-Deen Nabil** "A New Hybrid Algorithm for the Mutual Exclusion Problem in The Distributed Systems", IJICIS, Vol. 2, no. 2, Juillet 2002.
[RAM89]: K. Raymond "Tree-based algorithm for distributed mutual exclusion". ACM Transactions on Computer Systems, 7(1): 61-77, February 1989.
[RAM89a]: K. Raymond "A distributed algorithm for multiple entries to a critical section," Information processing Letters, no. 30, feb. 1989, pp. 189-193.
[RAY91]: M. Raynal "A simple taxonomy for distributed mutual exclusion algorithms", Operating Systems Review, Vol. 25, No. 2, April 1991, pp. 47-49
[RAY91a]: M. Raynal "A distributed solution for the k-out of-m resources allocation problem, " Lecture Notes in Computer Science, Spring Verlag,497, pp 599-609,1991.
[RIA81]: Glenn Ricart, Ashok K. Agrawala "An Optimal Algorithm for Mutual Exclusionin Computer Networks". CACM 24(1): 9-17 janvier 1981.
[SIN89]: M. Singhal "A Heuristically Aided Algorithm for Mutual Exclusion in Distributed Systems". IEEE Transactions on Computers 38(5): pages: 651-662 1989.

[SIN92]: M. Singhal "A Dynamic Information-Structure Mutual Exclusion Algorithm for Distributed Systems". IEEE transactions on Parallel and Distributed system 3(1):121-125 1992.

[SRR92]: P.SRIMANI et R.REDDY, "Another distributed algorithm for multiple entries to a critical section," Information Processing Letters, vol. 41, no. 1, jan. 1992, pp. 51-57.

[SUK85]: I. Suzuki, T. Kasami "A Distributed Mutual Exclusion Algorithm", ACM Transactions on Computer Systems (TOCS), Volume 3 pages 344-349 1985

[TEM00]: V.P. Team, ''The network simulator-Ns2'', *VINT Project Team*, Available at http://www.isi.edu/nsnam/ns/.

[THO79]: R. H. Thomas. 'A majority consensus approach to concurrency control for multiple copy databases'. *ACM Transactions on Database Systems*, 4(2), 1979.

[WAC02]: Yutaka Wada , Zixue Cheng, 'An efficient distributed method for allocating resources based on an unobstructed squeezing technique', ACM SIGOPS Operating Systems Review, v.36 n.3, p.33-45, July 2002.

[**WAK97]: J. Walter, S. Kini,** "Mutual Exclusion on Multihop, Mobile Wireless networks", Technical Report 97-014, Texas A&M University, 1997.

[**WWV98]: J.Walter, J.Welch, et N.Vaidya,** "A Mutual Exclusion Algorithm for Ad Hoc Mobile Networks," 1998 Dial M for Mobility workshop, Dallas TX, 1998, 15 pgs.

[WCM01]: J.Walter, G.Cao, et M.Mohanty, "A k-Mutual Exclusion Algorithme for Ad Hoc Wireless Networks," Proceedings of the first annual workshop on principles of Mobile Computing (POMC 2001), August, 2001.

[WCM01a]: J. Walter, G. Cao and M. Mohanty, "A Token Forwarding K-Mutual Exclusion Algorithm for Wireless Ad Hoc Networks", Technical Report, Texas A&M University, 2001.

Résumé

Dans ce travail, nous proposons une nouvelle solution au problème de la h-k exclusion mutuelle dans les réseaux mobiles ad hoc. Les différents mécanismes que nous proposons s'articulent sur une structure logique de niveaux divisée en deux arborescences logiques ; *Racine* et *Distribution*. Cette structure permet de canaliser le mouvement des jetons, de minimiser le nombre de messages échangés pour transmettre le ou les jetons et de s'adapter aux changements de la topologie du réseau et. En outre, le principe de niveaux permet une organisation logique des nœuds en sous-ensembles et cette organisation logique suit systématiquement la topologie physique du réseau.

Le protocole proposé est basé sur deux jetons ; *Racine* et *Clé*, chacun appartient à l'une ou l'autre arborescence. Le nœud qui possède le jeton racine collecte les requêtes qui sont mises dans une file d'attente suivant l'ordre de leurs arrivées. Le nœud qui possède le jeton clé détient les ressources disponibles dans le réseau et se charge de les distribuer.

La mobilité des nœuds a fait partie intégrante de la solution proposée grâce au mécanisme d'inversement de liens en cas d'une défaillance de liens. Ce mécanisme d'inversement de liens est utilisé aussi en cas de déplacement de jeton, car la racine doit être dans le niveau le plus bas, de plus les chemins vers la racine doivent rester décroissant (du plus haut niveau jusqu'à la racine).

Les comportements observés durant les différents tests de simulation nous ont permis de déduire une dépendance immédiate du temps moyen d'attente et du nombre moyen de messages pour une entrée en section critique ainsi que le délai de synchronisation vis-à-vis du nombre de ressources disponibles dans le système. En général et dans tous les cas, l'augmentation du nombre de ressources est en faveur du protocole en terme de performances. Le protocole se montre bien adapté à la mobilité car cette dernière n'influe pas sur ses performances. Cependant, dans le cas de forte mobilité, nous constatons une légère dégradation des performances comparées à celles du cas de faible mobilité. Ces dernières étant moins bonnes que celles du cas statique. Néanmoins, les résultats obtenus sont généralement acceptables.

Mot clés : Réseau mobile ad hoc, h-k exclusion mutuelle, section critique, ressources, jeton.