N° d'ORDRE : ***06/2009-E/IN***

REPUBLIQUE ALGERIENNE DEMOCRATIQUE ET POPULAIRE
MINISTERE DE L'ENSEIGNEMENT SUPERIEUR ET
DE LA RECHERCHE SCIENTIFIQUE
UNIVERSITE DES SCIENCES ET DE LA TECHNOLOGIE
« HOUARI BOUMEDIENNE »
FACULTE D'ELECTRONIQUE & INFORMATIQUE

THÉSE

Présentée pour l'obtention du diplôme DOCTORAT D'ETAT
EN INFORMATIQUE

Spécialité : Informatique

Par : Mahfoud BENCHAÏBA

Sujet

Allocation de ressources dans les réseaux mobiles ad hoc

Soutenue le : 02/11/2009, devant le jury composé de :

Mr- N. BADACHE, Professeur, USTHB.	Président,
Mr- M. AHMED NACER, Professeur, USTHB.	Directeur de Thèse,
Mr- A. BOUABDALLAH, Professeur, Univ. Compiègne France.	Examinateur,
Mme- A. AISSANI-MOKHTARI, Professeur, USTHB.	Examinateur,
Mr- M. BOUFAIDA, Professeur, Univ. Constantine	Examinateur.

Dédicaces

A mes parents,
à toute ma famille,
à toute ma belle famille,
à tous mes proches de sang et amis,
je dédie cette Thèse.

Remerciements

Pour commencer, je remercie vivement le Professeur Mohamed Ahmed Nacer, pour avoir dirigé mes travaux de Thèse et m'avoir encouragé durant toutes les étapes de réalisation du projet.

Je remercie également le Professeur Nadjib Badache, pour ses orientations, encouragements et pour m'avoir fait l'honneur de présider le jury de ma soutenance.

Je remercie le Professeur Madjid Bouabdallah pour m'avoir accueilli et aidé à différentes reprises au sein de son équipe durant mes séjours au sein du laboratoire Heudiasyc de l'Université de Technologie de Compiègne et pour l'intérêt qu'il porte à mon travail en acceptant d'être examinateur de cette Thèse.

Je remercie le Professeur Mahmoud Boufaïda pour l'intérêt qu'il porte à mon travail en acceptant d'être examinateur de cette Thèse.

Je remercie le Professeur Aïcha Mokhtari pour l'intérêt qu'elle porte à mon travail en acceptant d'être examinatrice de cette Thèse.

Je remercie également mes parents de m'avoir aidé à faire mes études à tous les niveaux et m'avoir montré à être patient.

Je remercie aussi ma petite famille et en particulier ma femme qui m'a supportée, aidée et encouragée tout au long de la période de la réalisation de cette Thèse.

Je remercie Yacine Challal et Imed Romdani pour le plaisir de partager avec eux des moments scientifiques et amicaux exceptionnels durant mes différents séjours au laboratoire Heudiasyc de l'Université de Technologie de Compiègne.

Je remercie également tous mes collègues enseignants du Département Informatique de l'USTHB pour les bons moments qu'on a passé ensemble au Département.

Pour finir, je remercie tout ceux qui m'ont aidé de près ou de loin pour concrétiser ce travail.

Table des matières

Liste des figures

Figure **Page**

Liste des tables

Table **Page**

Glossaire et terminologie

Accès à la SC : Par abus de langage, cette expression est utilisée pour exprimer l'opération de manipulation d'un ou plusieurs exemplaires de ressources critiques. Cette expression est motivée par le fait que la condition de sûreté implique que chaque ressource ne peut être utilisée que par un seul processus à la fois.

Charge de demandes : ou charge, désigne le nombre (ou la fréquence) de demandes de SC par processus et par unité de temps. La période de redemande pour un processus est calculée par le temps qui sépare le moment de sortie de la SC de ce processus et le moment de sa prochaine demande.

Connectivité : Elle est calculée en terme de pourcentage exprimant le rapport entre le nombre de liens existants du graphe et le nombre de liens possibles que peut avoir un graphe.

DAG : Contraction de Direct Acyclic Graph, c'est une structure d'arbre avec liens orientés vers la racine.

EM : Exclusion mutuelle.

h-k-EM : *h* parmi *k* exclusion mutuelle.

k-EM : *k*- exclusion mutuelle.

Nœud : Les termes *processus*, *site* et *nœud* ont tendance à se fondre malgré leurs différences. Dans les systèmes distribués, nous utilisons souvent le terme de *processus* et occasionnellement le terme de *site*. Dans les environnements mobiles, nous utilisons couramment le terme de *nœud* et occasionnellement le terme de *processus*.

Protocole : Désigne algorithme ou solution.

Quorum : C'est un ensemble de processus construit d'une certaine manière.

Scalabilité du réseau : Augmentation du nombre de nœuds du réseau.

SC : Désigne Section Critique.

Systèmes distribués : On confond systèmes distribués et *réseaux statiques*.

1– EM : Désigne l'exclusion mutuelle comme cas particulier de la *k*-exclusion mutuelle.

1- 1 EM : Désigne l'exclusion mutuelle comme cas particulier de *h-k* exclusion mutuelle.

Résumé

L'étude des services fondamentaux d'allocation de ressources est d'une extrême importance pour construire de nouveaux systèmes pour les réseaux mobiles ad hoc. Ces réseaux présentent des challenges majeurs à cause des nouvelles caractéristiques introduites qui ne sont pas connues des systèmes distribués classiques.

Le point d'entrée au domaine d'allocation de ressources et notamment au domaine de synchronisation est l'*exclusion mutuelle*. Ce problème, fondamental dans les systèmes informatiques, est rencontré lorsqu'on est en présence d'un ensemble de processus qui demandent, de façon simultanée, l'accès à une ressource *unique* à travers une parcelle de code appelée *section critique* (SC). Au plus, un processus peut exécuter sa SC à un moment donné. Quand la ressource existe en *k* exemplaires, il s'agit du problème de la *k- exclusion mutuelle* dans lequel chaque processus ne peut utiliser qu'une seule ressource à la fois. Donc, au maximum *k* processus exécutent en même moment leurs SC. Ce problème est alors une généralisation de l'exclusion mutuelle. Cependant, un processus peut vouloir demander *h* (*h* <=*k*) exemplaires de ressources ; il s'agit du problème de la *h parmi k exclusion mutuelle* (ou *h*-*k*- EM). Le nombre total de ressources alloué au même moment aux différents processus ne doit pas dépasser *k*. La *h*-*k* EM est une généralisation des deux problèmes précédents. En effet, si *h=1* on retrouve le problème de la *k*- EM et si de plus *k=1*, on retrouve l'exclusion mutuelle; ce qui veut dire que ces trois problèmes sont étroitement liés.

Dans cette thèse, nous avons abordé ces trois problèmes qui sont, en l'occurrence, l'exclusion mutuelle, la *k*- exclusion mutuelle et la *h* parmi *k* exclusion mutuelle qui sont respectivement décrits par les trois parties de ce document. Chacune de ces trois parties commence par un état de l'art des solutions existantes dans la littérature des systèmes distribués et réseaux mobiles ad hoc avant de présenter la (ou les) solution(s) proposée(s).

Le protocole d'exclusion mutuelle que nous proposons [19,20, 21] s'écarte de tous ceux connus dans ce domaine, il se base sur les liens physiques et n'utilise aucune structure logique. Les requêtes sont acheminées suivant un rayon dynamique de diffusion. Le jeton est géré selon une méthode qui combine la circulation et la demande. De plus, la politique de satisfaction des nœuds favorise en même temps les nœuds les plus proches et les requêtes les plus anciennes. La mobilité des nœuds fait partie intégrante de la solution proposée grâce aux différents mécanismes entrepris. Aussi, les pannes de nœuds, le partitionnement, la fusion et par conséquent, tous les problèmes qui en résultent (pertes et duplications de jetons) sont pris en charge dans la solution. Le protocole est distribué, équitable, satisfait les requêtes de manière symétrique, favorise la scalabilité du réseau et considère que le nombre de nœuds est indéfini. A partir des différents tests et métriques effectuées, le protocole donne des résultats satisfaisants dans sa globalité. Le nombre de messages qu'il génère est assez faible surtout

pour les moyennes et fortes charges. Le protocole se montre bien adapté à la mobilité, car cette dernière n'influe pas beaucoup sur ses performances globales.

Le protocole de *k*- exclusion mutuelle que nous proposons ne généralise ni n'adapte aucune autre solution qui le précède. Il est paramétré en fonction de l'état du système afin d'interagir avec tous les mouvements qui peuvent survenir. Il est orienté demande de jeton et suppose l'existence de *k* jetons dans le système. Face aux contraintes imposées par l'environnement mobile ad hoc, plusieurs techniques ont été proposées. Nous citons, par exemple, les relances, l'utilisation de la file d'attente temporaire, l'utilisation de jetons de type *Collector* pour favoriser la satisfaction de requêtes et l'activation des jetons oisifs, la cession de jeton pour la satisfaction de requêtes lors du parcours et l'évitement d'isolation temporaires ou perte de jetons, la collecte d'informations pour construire une connaissance locale du système sans aucun coût, etc. En général, notre protocole de *k*- EM offre ses meilleures performances dans les conditions de fortes charges comparées à celles de faibles charges. En comparaison avec son unique prédécesseur relativement aux métriques communes, il offre des résultats encourageants.

Le premier protocole de *h*-*k*- exclusion mutuelle que nous proposons [16, 14, 15] reprend certains concepts de la solution dans [20]. Il se base sur les liens physiques et ne considère aucune structure logique et ne fait pas recours aux services de la couche de routage. Les requêtes sont acheminées suivant un rayon d'envoi dynamique. La politique de satisfaction des nœuds favorise en même temps les nœuds les plus proches, les requêtes les plus anciennes et les demandeurs de moins de ressources (ce qui évite la sous utilisation de ressources). La mobilité des nœuds fait partie intégrante de la solution proposée grâce aux différents mécanismes entrepris. Aussi, le protocole prend en charge les pannes de nœuds, le partitionnement, le fusionnement et par conséquent, tous les problèmes qui en résultent (perte de jeton, perte de ressources, duplications de jeton et de ressources). Les résultats de simulation montrent que le protocole donne des résultats satisfaisants dans sa globalité. Ces résultats sont en accord avec ceux obtenus dans le protocole d'exclusion mutuelle cadre [20]. Les complexités des différentes métriques sont à peu près semblables, mais avec des valeurs ajoutées induites par les messages liés à la gestion de ressources.

Le second protocole de *h*-*k*- exclusion mutuelle que nous proposons se base également sur les liens physiques, ne fait pas recours aux services de la couche de routage, mais considère deux structures logiques de DAG à niveaux correspondant aux deux jetons qui y circulent respectivement. La première, i.e. la structure de DAG distribution permet la distribution de ressources aux nœuds demandeurs et la récupération de ressources libérées et la seconde, i.e. la structure de DAG racine permet de récolter les requêtes sur les ressources. Ces deux structures sont mappées directement sur les liens physiques du réseau sous-jacent. Différents mécanismes ont été définis pour récolter et satisfaire les requêtes en se basant sur les chemins définis par ces structures. Ces structures sont maintenues constamment lors des déplacements de jetons et des changements de liens physiques entre les nœuds en faisant recours à la méthode d'inversion de liens afin d'assurer la continuité du service. Le maintient de plusieurs chemins vers les nœuds détenteurs des deux jetons permet de minimiser le recours à l'inversion de liens. L'harmonie de fonctionnement entre les mécanismes associés aux déplacements de jetons, la circulation des requêtes, des ressources et la distribution des ressources fait que le protocole favorise l'efficacité du service. En comparaison avec la solution similaire de J-R. Jiang [55], cette structuration et les mécanismes associées permettent d'une part, d'équilibrer mieux la charge du service de *h*-*k* EM et d'autres part, de réduire le nombre de messages échangés notamment ceux liés à la réorganisation des deux structures de DAG, et tout cela pour réduire et équilibrer la consommation d'énergie entre les différents nœuds. Les résultats de simulation montrent que le protocole donne ses meilleurs

résultats pour les moyennes et fortes charges de demandes, que l'augmentation du nombre de ressources est en faveur du service et que la mobilité a peu d'influence sur ses performances. En comparaison avec la solution donnée au Chapitre 7, il s'avère que le protocole donne des résultats moins performants. Evidemment, ceci revient aux compléxités addditionnelles liées à l'entretient des structures logiques.

Mots clés: Allocation de ressources, exclusion mutuelle, *k*- exclusion mutuelle, *h* parmi *k* exclusion mutuelle, jeton, section critique, réseau mobile ad hoc.

Keywords: Resource allocation, mutual exclusion, *k*- mutual exclusion, *h*-out of-*k* mutual exclusion, token, critical section, mobile ad hoc networks.

Chapitre I

Préliminaires

1. Motivation de la thèse

Le développement récent de la technologie et des besoins dans le domaine de l'informatique a donné naissance à différents types de réseaux de communication. L'histoire des réseaux a commencé depuis les années 70 par l'introduction des premiers réseaux filaires [79]. Depuis, l'évolution dans ce domaine n'a cessé d'accroître; l'introduction du réseau Internet a fait augmenté encore l'intérêt à ces réseaux. De nos jours, deux nouveaux types de réseaux sont d'actualités; à savoir, les réseaux mobiles avec infrastructure et les réseaux mobiles sans infrastructure. Le premier type de réseau est formé d'un sous réseau de machines fixes appelé *backbone* et d'un ensemble, qui peut être très grand, de machines libres ou mobiles qui gravitent autour (voir Figure 1). Ce type de réseau trouve en grande partie son application dans le domaine de télécommunication et IP. Le deuxième type de réseau, intitulé *réseau mobile ad hoc* (ou MANET : Mobile Ad hoc NETwork) est formé seulement de machines mobiles [129]; il ne dépend d'aucune infrastructure ni d'un contrôle centralisé (voir Figure 2). Il est né pour répondre à un besoin très particulier, en l'occurrence, la tactique dans la défense militaire américaine. Ensuite, il a été dégradé pour être utilisé dans de nombreux domaines, tels que dans les catastrophes (incendies, tremblements de terre, inondations etc …), dans des sites archéologiques, dans des musées, dans des expéditions scientifiques; et en général, dans les endroits où il n'est pas pratique, voire impossible, de déployer un réseau filaire. Vu la diversité des domaines d'applications de ces réseaux [119], un intérêt croissant de recherche leur est porté ces dernières années. Les nouvelles contraintes introduites par cet environnement font qu'il est nécessaire de réviser les solutions déjà conçues pour les systèmes distribués classiques. Cependant, la majorité des travaux menés portent sur la couche Mac [52, 71, 153] et sur le routage [129, 119]; ce qui produit les fonctionnalités fondamentales de communications. Cependant, peu de travaux sont effectués dans d'autres domaines et en particulier sur les problèmes généraux l'allocation de ressources. Notre intérêt

porte sur les problèmes de contrôles ou services fondamentaux liés à l'allocation de ressources dans les réseaux mobiles ad hoc.

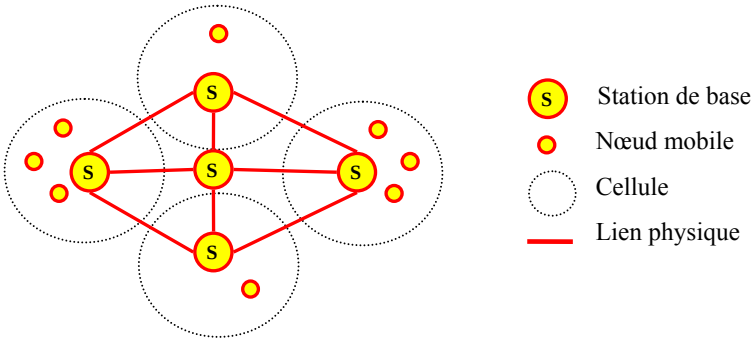


Figure 1: *Architecture d'un réseau avec infrastructure.*

(a) : Avant mouvement (b) : Après mouvement

Figure 2: *Architecture d'un réseau sans infrastructure.*

2. Un point de vue sur les réseaux mobiles ad hoc

2.1. Caractéristiques

Un réseau mobile ad hoc peut être vu comme un réseau qui se déploie spontanément, aléatoirement et indépendamment de toute infrastructure statique. Ce réseau est composé d'une collection de noeuds mobiles communicants par des interfaces de communication sans fil en formant une topologie arbitraire et dynamique. Chaque nœud est capable de communiquer avec des noeuds délimités par sa portée de communication autrement ses messages doivent traverser une séquence de nœuds intermédiaires avant d'arriver à leur destinations. Il s'ensuit que chaque noeud agit en même temps comme un hôte et comme un routeur pour router les messages.

Dans un réseau mobile ad hoc, les nœuds peuvent se déplacer librement, ce qui induit des changements de liens puisque chaque nœud peut apparaître ou disparaître de la portée de communication d'un autre. Ces réseaux sont sujets à un certain nombre de contraintes : Premièrement, puisque chaque nœud est alimenté par une batterie avec une autonomie limitée, il est nécessaire de prendre en considération la conservation d'énergie (i.e. réduire la complexité en messages…). Deuxièmement, à cause des déconnexions fréquentes par mesure de conservation d'énergie ou par nature des services connus de ces réseaux, il faut remédier à ces pratiques à travers la gestion du mode veille, et surtout par la conception de solutions *distribuées* et *symétriques*; ce qui permet d'éviter de baser le contrôle sur des nœuds particuliers. Troisièmement, par cause de mobilité d'une part, le changement fréquent de la topologie (dont la conséquence est le partitionnement du réseau) rend les solutions de routage en particulier et des autres problèmes en général connues des systèmes distribués non applicables directement aux réseaux mobiles ad hoc. D'autre part, la traversée de zones à

obstacles ou à bruits engendre une asymétrie des liens de communication, des délais de transmissions variables et des pertes de paquets de messages. Le souci de conservation d'énergie par réduction de la puissance du signal de transmission favorise aussi l'asymétrie dans les liens de communication. Quatrièmement, la nature du medium de communication rend ces réseaux physiquement vulnérables aux attaques, ce qui incite à prendre en charge les problèmes de sécurité. Et finalement, la bande passante limitée, oblige à rationner le partage de celle-ci [70] (au niveau de la couche MAC). Toutes ces caractéristiques des réseaux mobiles ad hoc engendrent un challenge important aux applications et services de ces réseaux.

2.2. Modes d'implémentation des services d'allocation de ressources

Les protocoles de services distribués liés à l'allocation de ressources peuvent être implémentés de deux manières différentes, selon l'architecture de conception en couches: soit naturellement au dessus de la couche de routage, soit au même niveau que la couche de routage. La majorité des premiers travaux sur les services et applications distribués dans les réseaux mobiles ad hoc appliquent directement les solutions des réseaux filaires en bénéficiant des services de la couche de routage (voir Figure 3). Des recherches montrent que les solutions structurées selon cette manière sont inefficaces, voire parfois incorrects [129, 54].

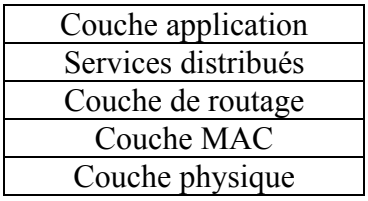


Figure 3 : *Services sans prise en charge de la mobilité.*

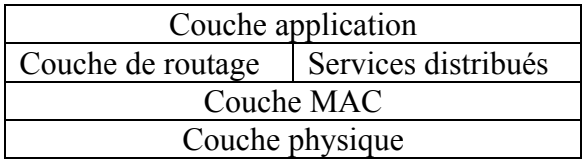


Figure 4 : *Services avec prise en charge de la mobilité.*

Considérons la structure logique d'arborescence (construite sur le réseau physique) de la Figure 5(a). Chaque nœud possède un pointeur relatif vers le nœud particulier H qui fourni un service donné. Admettons que le nœud *A* demande un service. Si on considère la couche de routage, son message de demande de service va suivre, par exemple, le chemin physique A-B-*C*-*H* en accord avec la structure logique. Dans la Figure 5(b), des ruptures de liens physiques entre les nœuds *A* et *B* et les nœuds *B* et *C* et une création de lien entre A et *D* ont eu lieu. Etant donné que la structure logique est toujours la même, pour acheminer la requête du nœud A, la couche de routage devra trouver le chemin qui mène de *A* à *B*. Par conséquent, la requête du nœud *A* va suivre le chemin *A*-*D*-*C*-*B*-*C*-*H*; ce qui augmente le coût de communication. Il est alors utile de mettre à jour la structure en dépendant de la mobilité de telle sorte que les liens logiques correspondent toujours aux liens physiques. La réalisation de ces mises à jour à l'aide de la couche de routage serait très coûteuse.

Il est clair que les services distribués qui tiennent compte de la mobilité (Figure 4) sont beaucoup plus efficaces. Ces services ne font alors pas recours aux services de la couche de routage mais ils utilisent naturellement des fonctionnalités de la couche MAC, notamment

pour la transmission des messages et parfois pour l'entretient du voisinage ou pour la diffusion de messages.

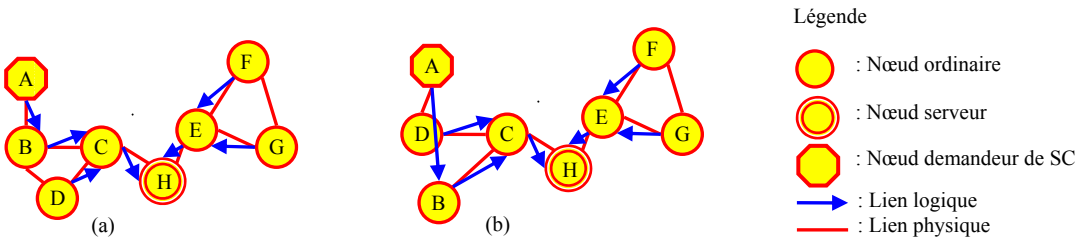


Figure 5: *Mobilité et chemins des messages.*

Les travaux que nous avons menés dans le cadre de cette thèse se basent sur la deuxième option de structuration. Plusieurs outils et mécanismes, permettant l'acheminement des messages de services d'allocation de ressources et la prise en charge des conséquences de la mobilité et des pannes de nœuds dans différentes solutions apportées, sont produits.

2.3. Routage dans les réseaux mobiles ad hoc

La tache de routage est de construire des routes entre noeuds d'un réseau. Par cause de mobilité des noeuds, la topologie du réseau est sujette à des modifications fréquentes; ce qui incite les protocoles de routage à réagir rapidement. Le routage doit alors être un protocole distribué qui établie plusieurs routes sans boucles et avec un minimum d'effet d'intrusion [119]. Les solutions de routage des systèmes distribués consomment beaucoup d'énergie et de bande passante; ce qui les rend non efficaces pour les exigences des réseaux mobiles ad hoc. Le routage dans les réseaux mobiles ad hoc est alors un domaine en plein essor. Les solutions actuelles se divisent en trois approches.

Les protocoles proactifs [107]: Leur but est d'établir à l'avance les routes vers toutes les destinations à travers l'entretien des tables de routage. Ces solutions essayent de maintenir la consistance des informations entretenue à travers l'échange périodique des tables entre les différents nœuds du réseau. Les performances de ces solutions dépendent alors de l'efficacité de mise à jour des informations de routage.

Les protocoles réactifs [62, 54, 105]: Ils ne créent pas de routes à l'avance. Quand un noeud désire envoyer un message vers une destination donnée, l'exploration de la route vers uniquement cette destination est alors enclenchée. Ces solutions engendrent un retard de communication mais en échange, elles ne nécessitent pas d'entretenir des informations mises à jour.

Les protocoles hybrides [48]: Ces solutions mixent l'approche proactive et l'approche réactive. Elles utilisent l'approche proactive dans une zone locale et l'approche réactive entre les différentes zones; ce qui leur permet de combiner les avantages des deux approches mais aussi leurs inconvénients.

Il ressort que le routage dans les réseaux mobiles ad hoc est un domaine en plein essor mais les problèmes de performances restent ouverts [108, 119, 132, 43]. Par conséquent, les performances des applications qui se basent sur les services de la couche de routage dépendent alors de celles de cette couche et sont loin d'être optimales.

2.4. Quelques services de la couche MAC

La couche MAC (Medium Access Control) se charge naturellement de la gestion d'accès au canal de communication pour échanger des bits d'informations entre deux nœuds voisins. Les protocoles régissant l'accès au medium de communication doivent prendre en charge un certain nombre de problèmes, par exemple le problème du nœud caché, du nœud exposé, de conflits d'accès au médium [71, 140, 139, 66] etc… Ces protocoles respectent la norme IEEE 802.11 [39, 52] (ISO/IEC 8802-11) qui est un standard international décrivant les caractéristiques d'un réseau local sans fil (Wireless Local Area Network ou WLAN). Le nom WiFi (contraction de Wireless Fidelity, parfois noté Wi-Fi) correspond initialement au nom donné à la certification délivrée par la WECA (Wireless Ethernet Compatibility Alliance), l'organisme chargé de maintenir l'interopérabilité entre les matériels répondant à la norme 802.11. Par abus de langage, le nom de la norme se confond aujourd'hui avec le nom de la certification. Ainsi, un réseau WiFi est en réalité un réseau répondant à la norme 802.11.

Les solutions que nous apportons font appel à la couche MAC naturellement pour l'échange de messages mais aussi pour le service auxiliaire d'entretient du voisinage. A travers l'utilisation des messages *''Hello''* périodique, la couche MAC permet de savoir la présence ou l'absence d'un nœud voisin [53, 31]. Pour ce qui nous concerne, ces services se traduisent par les deux d'événements *LinkUp(j)* et *LinkDown(j)* qui correspondent respectivement à l'événement d'établissement et de perte d'un lien avec un nœud désigné par l'identité *j*.

3. Allocation de ressources dans les réseaux mobiles ad hoc

Depuis l'aire de l'informatique, l'allocation de ressources est considérée comme une brique de base pour la construction des systèmes d'exploitation. Ce service représente une grande partie de la mission de ces systèmes. En effet, tous les objets matériels et logiciels sont des ressources, la gestion de composants s'avère vitale pour le fonctionnement de ces systèmes informatiques. Dans les systèmes centralisés caractérisés par la présence d'une mémoire partagée centralisée, la gestion du partage de ces ressources est mise en œuvre à travers des mécanismes et outils offerts par ces systèmes. Il s'agit des moniteurs, des sémaphores, des mécanismes liés au matériel [121, 134, 81, 122] etc. L'avènement des réseaux de communication a rendu difficile la tache d'allocation de ressources avec l'introduction de nouvelles caractéristiques qui sont des conséquences de la répartition, à savoir:

- l'absence de mémoire commune,
- les délais de transmission entre sites qui sont finis mais non bornés,
- l'absence de référentiel temporel global,
- l'asynchronisme et le non déterminisme de l'exécution des processus.
- la connaissance locale d'un processus sur le réseau: chaque site ne connaît, par défaut, que les canaux (liens) le reliant avec ses voisins ou peut être ses voisins immédiats.
- la panne de liens et de sites: Avec la multiplicité de ces composants, forcément de temps à autre au moins l'un d'eux tombe en panne. Les produits logiciels doivent alors être résistants (ou tolérants) aux pannes.

La conséquence des trois premières caractéristiques est l'absence d'état global réel [114, 115, 32]. D'autres caractéristiques structurelles et dynamiques liées aux supports de communications des réseaux fixes sont aussi à prendre en considération: Topologie,

déséquencement de messages, duplication de messages, altération de messages et perte de messages.

Les solutions aux problèmes d'allocation de ressources connues dans les systèmes centralisés ne sont alors plus valables dans le contexte des systèmes distribués. De ce fait, il est nécessaire d'en définir de nouvelles solutions qui tiennent forcément compte de ces nouvelles caractéristiques. Les réseaux mobiles ad hoc ajoutent de nouvelles caractéristiques (support de communication sans fil, mobilité des nœuds, absence d'infrastructure et de contrôle centralisés,…) non connus dans les systèmes distribués. Ces nouveaux défis engendrent une valeur ajoutée de complication quant à l'établissement de solutions aux problèmes d'allocation de ressources dans ce type de réseaux.

4. Problèmes traités

Dans cette thèse, nous nous intéressons aux services généraux d'allocation de ressources dans les réseaux mobiles ad hoc qui peuvent aussi être considérés comme des outils de synchronisation. Ces services généraux ne s'intéressent pas à des ressources particulières mais peuvent s'appliquer dans divers contextes, soit à des composants des systèmes d'exploitation soit à des applications. Il s'agit d'apporter des solutions à des problèmes de contrôle, classiques des systèmes distribués, dans le contexte des réseaux mobiles ad hoc.

Le point d'entrée au domaine d'allocation de ressources et notamment au domaine de synchronisation est l'*exclusion mutuelle*. Ce problème, fondamental dans les systèmes informatiques, est rencontré lorsqu'on est en présence d'un ensemble de processus qui demandent, de façon simultanée, l'accès à une ressource *unique* à travers une parcelle de code appelée *section critique* (SC). Au plus, un processus peut exécuter sa SC à un moment donné. Quand la ressource existe en k exemplaires, il s'agit du problème de la *k- exclusion mutuelle* dans lequel chaque processus ne peut utiliser qu'une seule ressource à la fois. Donc, au maximum k processus exécutent en même moment leurs SC. Ce problème est alors une généralisation de l'exclusion mutuelle. Cependant, un processus peut vouloir demander h ($h <= k$) exemplaires de ressources ; il s'agit du problème de la *h parmi k exclusion mutuelle* (ou *h-k-* EM). Le nombre total de ressources alloué au même moment aux différents processus ne doit pas dépasser k. La h parmi k EM est une généralisation des deux problèmes précédents. En effet si $h=1$, on retrouve le problème de la k- EM et si de plus $k=1$, on retrouve l'exclusion mutuelle; ce qui veut dire que ces trois problèmes sont étroitement liés.

Les problèmes d'allocation de ressources sont aussi variés qu'ils en existent d'autres. Si on continue dans le chemin de généralisation, un processus peut effectuer une seule demande sur plusieurs classes de ressources et chacune avec un nombre donné d'exemplaires [26, 145]. Les lecteurs/rédacteurs est un autre problème classique qui consiste en le partage d'une seule ressource avec la contrainte que les rédacteurs utilisent chacun la ressource en exclusion mutuelle. Nous citons aussi l'exclusion mutuelle de groupes [29, 63, 64, 151, 65, 128] qui s'inspire de l'exclusion mutuelle, il s'énonce comme suit: Il existe m ressources différentes accessibles par un ensemble de processus de telle sorte *au plus une ressource* est active à un instant donné (principe d'exclusion mutuelle). Celle-ci peut être accessible par un nombre indéterminé de processus, c'est le principe de concurrence. Si on limite le nombre de processus qui accèdent simultanément à la ressource à k, le problème devient celui de la G k-exclusion mutuelle (ou GK- EM) [59].

Dans cette thèse, nous sommes concernés par le développement de solutions aux trois problèmes décrits en premier en essayant de tenir compte du maximum de contraintes imposées par les réseaux mobiles ad hoc.

4.1. L'exclusion mutuelle

L'accès à une ressource non partageable doit être synchronisée afin d'assurer qu'un seul processus à la fois utilise cette ressource. Les processus font donc recours à des protocoles d'EM qui ont pour rôle d'attribuer un privilège à un ensemble de processus, un processus à la fois; le processus qui détient ce privilège entre à sa SC. Ce privilège peut être attribué selon deux politiques différentes, à savoir la politique *équitable* et la politique avec *priorité*. Les solutions qui utilisent la priorité [92, 130, 51] répondent aux applications qui par nature définissent des classes différenciées de processus. Par exemple, dans le téléenseignement [130], la prise de parole est une ressource partagée entre l'enseignant et l'étudiant. En cas de conflit, la priorité est donnée à l'enseignant. Notre intérêt porte sur les solutions équitables.

Un protocole d'EM doit satisfaire les conditions suivantes définies par E.W.Dijkstra [42]), lui permettant une mise en œuvre fiable :

- *Condition d'EM :* Au plus un processus exécute la SC à un moment donné.
- *Condition de progression* : Si aucun processus n'est dans la SC, n'importe quel autre processus demandant l'accès à la SC doit pouvoir y accéder.
- *Condition de blocage :* Eviter l'interblocage au moment de la demande d'accès à la SC.
- *Condition d'attente bornée* : Un processus demandant sa SC, doit pouvoir y accéder au bout d'un temps fini.

Toutes ces conditions se résument seulement aux deux suivantes:

- *Condition de sûreté :* à tout moment, il y a au plus un processus exécutant sa SC.
- *Condition de vivacité :* pour toute exécution infinie du protocole, chaque demande d'accès à une SC doit finir par être satisfaite.

4.1.1 Modèles d'implémentation de l'exclusion mutuelle

Deux modèles majeurs de partage de charges de décision entre les processus qui concourent pour l'accès à la ressource critique peuvent être utilisés pour l'implémentation des mécanismes d'EM dans les systèmes distribués :

- Le modèle *centralisé*: Un processus particulier assure la fonction de coordinateur [131] (désigné éventuellement par élection [123, 97, 144, 35]), toutes les requêtes lui sont destinées. Il doit disposer de toutes les informations sur le système et doit donc se charger d'octroyer au demandeur la permission d'accéder à la ressource partagée.
- Le modèle *distribué*: La décision d'accès en SC est distribuée à travers un ensemble ou tous les processus impliqués dans le partage de la ressource critique. La solution au problème d'EM devient plus compliquée à cause de la difficulté d'obtenir une connaissance complète du système. Dans la suite de ce document, nous sommes concernés par ce modèle.

4.1.2. Approches d'exclusion mutuelle

Les protocoles d'exclusion mutuelle sont des services qui ont pour rôle d'ordonnancer les processus demandeurs quant à l'accès à une ressource critique. Ils doivent introduire des mécanismes et outils permettant de répondre à une question fondamentale qui est : Quel est la condition nécessaire et suffisante pour qu'un processus exécute sa SC? La réponse à cette question conduit aux deux approches principales d'exclusion mutuelle.

Approche à permission

Un processus n'accède en SC que s'il reçoit la *permission* de tous ou une partie de l'ensemble des processus. Les conflits sont résolus par l'instauration d'un système de priorité ou par un mécanisme d'ordonnancement. Cependant, le problème posé est de trouver le nombre minimal et suffisant de permissions que doit collecter un site pour entrer en section critique.

Approche à jeton

Un processus n'accède à sa SC que s'il obtient le *jeton*. Le jeton est un message spécial qui circule à travers tous les processus du système passant le droit d'accès à la ressource critique au processus qui le détient, le processus *privilégié*. La circulation du jeton est régie par une politique précise. La présence d'un jeton unique dans le système assure de manière intrinsèque l'exclusion mutuelle. Cependant, l'approche à jeton est susceptible à la perte de ce dernier.

4.2. La *k*- exclusion mutuelle

Les protocoles de la *k*- EM se scindent également en deux approches, l'approche à permission et l'approche à jeton et doivent vérifier les mêmes propriétés que celles de l'EM sauf que la condition de sûreté se reformule de la manière suivante: 'Chaque processus ne peut accéder qu'à une seule ressource à la fois et au maximum *k* processus exécutent leurs SC simultanément''.

Une solution au problème de *k*- exclusion mutuelle peut être obtenue en identifiant chaque ressource de manière unique et par suite un processus demandeur spécifie une identité donnée lors de sa demande. Il s'agit donc d'appliquer *k* protocoles d'EM pour servir ces processus. Cette manière d'adaptation s'avère simple mais engendre une sous utilisation apparente de ressources: Cette difficulté apparaît dès lors qu'un processus requière une ressource donnée qui est en cours d'utilisation alors qu'une autre instance est disponible. Une autre solution peut être envisagée, il s'agit de modifier un protocole d'EM pour permettre à chaque processus d'octroyer sa permission à au plus *k* processus à la fois. Cependant, cette approche peut violer la propriété de sûreté, car plus que *k* processus peuvent avoir la permission d'accès à la SC. De ce fait, il faut utiliser un protocole unique propre à la *k*- exclusion mutuelle dont le mécanisme d'allocation se base sur la disponibilité d'une copie quelconque parmi les *k* ressources.

4.3. La *h* parmi *k* exclusion mutuelle

Les protocoles de la *h* parmi *k* EM se scindent également en deux approches, l'approche à permission et l'approche à jeton et doivent vérifier les mêmes propriétés que celles de l'EM sauf que la condition de sûreté se reformule de manière suivante :

$\sum_{N}^{i=1} hi \leq k$, où h_i est le nombre de ressources accessibles par le nœud *i* à un instant donné. De plus, chaque exemplaire de ressource ne peut être allouée qu'à un seul processus à la fois.

Vu la multiplicité de ressources que peut demander un processus, la règle suivante doit être observée: *''un processus ne peut demander de ressources que s'il ne détient aucune*''. En apparence, cette règle introduit dans ce cas un retard quand à l'acquisition de ressources. Mais en réalité, elle ne représente pas réellement une contrainte puisque chaque processus peut demander à tout moment le nombre voulu de ressources (sous réserve de ne pas dépasser le nombre *k*). L'avantage en est la favorisation de la vivacité et la facilitation du développement de solutions au problème de la *h* parmi *k* EM. Une autre règle due au modèle d'allocation de ressources qui nous intéresse, en l'occurrence le modèle *And* : '*''un processus ne peut*

continuer son exécution que s'il acquière toutes les ressources demandées'' est à prendre en considération. D'autre part, pour éviter le problème d'interblocage (i.e. une des conditions de la vivacité), le service de *h*- *k*- EM ne doit pas allouer par parties les ressources à un même processus; il doit observer le principe de *tout ou rien*. En effet, considérons deux processus *p1* et *p2* qui demandent respectivement *h1* et *h2* ressources de telle sorte que *h1* + *h2* est très supérieur à *k*; si par cause de non disponibilité, on alloue une quantité *s1<h1* à *p1* ; et par la suite, après libération de ressources de la part d'un processus *p3*, on alloue une quantité *s2<h2* à *p2* de telle sorte que (*h1-s1*) et (*h2-s2*) soient supérieurs au nombre de ressources allouées aux processus autres que *p1* et *p2,* s'il en existent. Aucun des deux processus *p1* et *p2* ne peut continuer son exécution quelque soient les libérations futures. Par conséquent, aucun de ces deux processus ne peut être satisfait, c'est l'interblocage!

Une solution au problème de la *h* parmi *k* exclusion mutuelle peut être obtenue en identifiant chaque ressource de manière unique et par suite un processus identifie les ressources requises lors de sa demande. Il s'agit donc d'appliquer *k* protocoles de *k*- EM pour servir ces processus. Cette manière d'adaptation s'avère simple mais engendre une sous utilisation apparente de ressources: Cette difficulté apparaît dès lors qu'un processus requière une ressource donnée qui est en cours d'utilisation alors qu'une autre 'copie' est disponible.

De ce fait, il faut d'une part se libérer de la méthode d'identification et d'autre part, concevoir un protocole unique propre à la *h* parmi *k* EM dont le mécanisme d'allocation se base sur la disponibilité de *h* copies quelconques parmi les *k* ressources.

4.4. Interface de communication du service d'allocation de ressources

Le service d'allocation de ressources (EM, *k*- EM et *h*-*k* EM) fonctionne selon une architecture en couches définie au niveau de tous les nœuds du réseau. A chaque noeud sont implémentés deux processus, un processus d'application (qui fait partie d'un ensemble qui se partagent l'accès à une ou plusieurs ressources) et un processus du service d'allocation de ressources. L'application utilise des messages de *calcul* pour assurer son fonctionnement tandis que le service d'allocation de ressources utilise des messages de *contrôle*. Le processus du service d'allocation de ressources communique d'une part, avec le processus local d'application et d'autre part, avec les autres processus du service d'allocation de ressources pour gérer l'accès à la ressource (ou aux ressources demandées) et avec le réseau qui lui permet d'être informé sur l'état de connexion avec son voisinage dans le réseau. La Figure 6 donne les différents messages de l'interface du processus du service d'exclusion mutuelle au niveau d'un noeud *i*.

Lorsqu'un processus d'application désire accéder à sa SC, il formule une requête au processus local du service d'allocation de ressources et attend l'autorisation d'accès; le processus local du service d'allocation de ressources lui obtient l'autorisation d'accès en communicant avec les autres processus d'allocation de ressources à travers le protocole correspondant. Une fois reçue cette autorisation, le processus d'application exécute sa SC et à la fin il envoie un message de libération au processus local de service d'allocation de ressources afin de permettre à un autre processus en attente d'accéder à sa SC. De manière générale, un processus d'allocation de ressources au niveau du nœud *i* est déclenché par un ensemble d'événements *asynchrones* en entrée et déclenche, lui-même, un ensemble d'évènements en sortie en réponse à un évènement en entrée ou suite à un besoin interne. Les événements en sorties sont les messages de service d'allocation de ressources qu'il échange avec les autres processus distants du service d'allocation de ressources ou avec le processus d'application local. Certains de ces événements sont totalement ordonnés grâce à l'utilisation du mécanisme d'estampillage de L. Lamport [76]. Chaque processus d'allocation de

ressources est alors structuré de manière *non déterministe* pour attendre et répondre à chaque événement; au niveau d'un processus, les événements son traités de manière *exclusive* (ou séquentiellement) afin de préserver la cohérence du système.

D'autre part, il est à noter que le service d'allocation de ressources fonctionne de manière continue étant donné que les demandes de ressources peuvent survenir à tout moment.

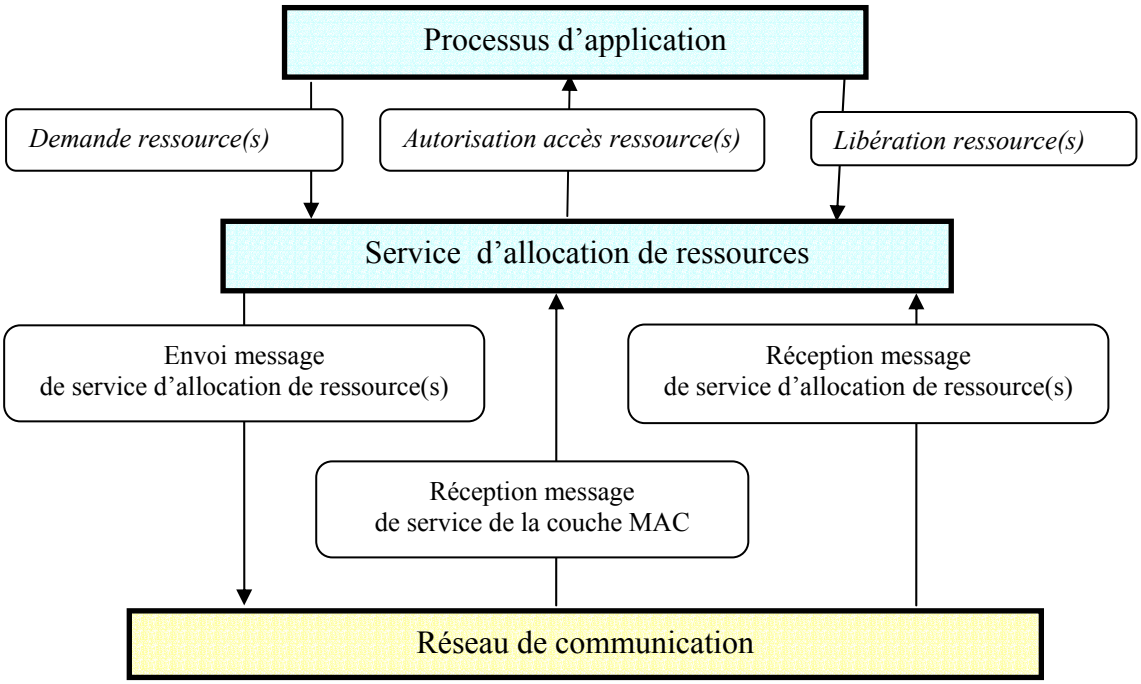


Figure 6: *Architecture du service d'allocation de ressources.*

4.5. L'estampillage dans le service d'allocation de ressources

Dans les systèmes informatiques distribués, il est souvent utile de déterminer un ordre total entre un ensemble d'événements se produisant au sein de ce système. Par exemple, dans le cas de services utilisant le modèle client - serveur, vu la multiplicité de demandes de services issues des différents clients, l'entretient de cet ordre au niveau du serveur permet d'assurer la vivacité du service; la datation de ces événements s'avère alors nécessaire. Etant donné l'absence de référentiel temporel global dans de tels systèmes, la synchronisation d'horloges physiques [76, 87, 106, 91] (, un mécanisme non évident à obtenir,) est l'un des moyens pour obtenir cet ordre. Cependant, certains systèmes se contentent d'un ordre logique obtenu par l'utilisation des mécanismes d'estampillage, notamment celui définit par L. Lamport [76]. Dans cette thèse, l'estampillage de L. Lamport est de rigueur dans l'allocation de ressources. Il permet d'une part, de résoudre les conflits entre les requêtes des différents nœuds quant à l'accès aux ressources en *contribuant* à créer un ordre de satisfaction et d'autre part, à favoriser la propriété de vivacité du service.

Le mécanisme d'estampillage de L. Lamport est définit comme suit :

- A chaque processus P_i est associée une horloge logique locale H_i, initialement à *0,* et dont les valeurs forment une suite monotone croissante. Cette horloge permet donc de dater les événements et notamment les messages émis et reçues au niveau de chaque noeud.
- A l'occurrence d'un événement interne à P_i, $H_i := H_i + d$ (typiquement $d = 1$) ;
- A l'émission d'un message, $H_i := H_i + d$;

- A la réception d'un message accompagné de H_j, $H_i := Max\ (H_i, H_j) + d$.

L'ordre obtenu par ce mécanisme n'est que partiel. Dans le but d'obtenir un ordre total, l'identité du processus où l'événement se produit est associé à son estampille, ce qui permet de briser les égalités entre les estampilles.

Il est à noter que dans certaines solutions d'allocation de ressources, l'ordre entre les requêtes est obtenu de manière implicite notamment lorsqu'une structure logique [110] est entretenue. Par exemple dans le cas d'un arbre, les requêtes reçues par un noeud de l'ensemble de sa descendance sont traitées dans l'ordre d'arrivée.

4.6. Défis d'implémentation

Dans le but de résoudre les trois problèmes d'allocation de ressources, dans les réseaux mobiles ad hoc, objets de cette thèse, il devient nécessaire d'adresser un certains nombre de défis; de ce fait, plusieurs questions se posent. Comment acquérir l'autorisation d'accès en SC? A qui envoyer la requête? Comment sélectionner le prochain nœud privilégié? Comment lui véhiculer le jeton (en supposant que l'approche à jeton est adoptée)? Si la route du jeton est perdue, comment faire ? Quoi faire si le jeton ne parvient pas? Si la perte du jeton (des ressources) est suspectée, comment le confirmer? Si la perte du jeton (des ressources) est confirmée, qui va le (les) régénérer? Comment s'assurer qu'un seul nœud le (les) crée? Comment adresser le partitionnement et le fusionnement ? Comment éliminer le jeton (les ressources) dupliqué (es) dans le cas de fusionnement? Comment assurer la propriété de sûreté et de vivacité? Comment rendre le protocole efficace?

Notre contribution est dédiée à répondre à toutes ces questions pour chaque cas d'étude.

4.7. Etude de performances

L'analyse de performances d'un protocole prévu pour les réseaux mobiles ad hoc est très utiles afin de cerner les cas de son applicabilité. Cependant, vu les nouvelles caractéristiques introduites par ces réseaux, il est souvent très difficile d'obtenir cette analyse de manière analytique pour les protocoles qui prennent en charges ces contraintes et notamment la mobilité et les pannes. De ce fait, les concepteurs de protocoles font recours à des tests dans des environnements simulés [136, 154]. La simulation ne donne pas une preuve de correction formelle. Néanmoins, elle permet d'une part de tester un protocole, et d'autre part d'anticiper les problèmes qui pourraient se poser dans le futur. Elle renseigne sur le comportement du protocole 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 protocole, une prise des mesures de performances. Ces mesures peuvent être déployées dans une analyse qui permet de tirer des conclusions sur le fonctionnement et l'efficacité du protocole ainsi que les schémas favorables et défavorables quant aux performances calculées.

Les métriques principales considérées dans le cadre de l'allocation de ressources et selon le problème traité sont: la *complexité en messages*, le *délai de synchronisation*, la *taille mémoire*, la *taille des messages* de contrôle et le *temps de réponse* du service. Dans le cas de simulations, ces métriques représentent une moyenne calculée après plusieurs tests. Ces évaluations sont réalisées en fonctions d'un certain nombre de paramètres, notamment la *charge* de demandes de services, la *mobilité* et la *connectivité*.

- La complexité en messages est le nombre total de messages nécessaires à un processus pour accéder à une section critique. Son calcul dépend de la nature du service.
- Le délai de synchronisation est le temps écoulé entre deux entrées successives en SC quelconques. En considérant que la durée de transmission d'un message est majorée, cette métrique est exprimée par le nombre de messages (souvent séquentiels) transmis entre deux SC.
- Le temps de réponse est l'intervalle de temps séparant l'envoi d'une requête par un processus et la sortie de la SC de ce même processus. Dans le contexte des réseaux, il est en général exprimé en nombre de messages et dans notre cas, il est remplacé par la complexité en messages.

Les paramètres sont :

- La mobilité: tous les protocoles que nous avons proposé dans cette thèse considèrent le mouvement libre de nœuds. De ce fait, le modèle de mobilité considéré lors des différentes simulations est le modèle aléatoire (*Waypoint* [78]) qui se base sur la variation de la distance relative entre les nœuds. Si les nœuds sont en mouvement pour un certain temps, alors la mobilité est la moyenne des changements de distances entre tous les nœuds durant cette période de temps. Cette définition donne une bonne estimation de la manière dont les nœuds se déplacent les uns par rapport aux autres. La formule de la mobilité est donnée ci-après (3), où $A_x(t)$ est la moyenne des distances séparant un nœud x de tous les autres à l'instant t (1); M_x est la mobilité moyenne d'un nœud x durant la simulation (2) (Δt représente le pas du calcul); n est le nombre de nœuds.

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

$$M_x = (1/(t-\Delta t))\sum_{t=0}^{T-\Delta t} \left|A_x(t+\Delta t) - A_x(t)\right|\ ; \tag{2}$$

$$Mob=(1/n)\sum_{i=1}^{n} Mi\ ; \tag{3}$$

- La charge de demandes est le nombre (ou la fréquence) de demandes par processus et par unité de temps. La période de demande pour un processus est calculée par le temps qui sépare le moment de sortie de la SC de ce processus et le moment de sa prochaine demande.
- La connectivité est liée au nombre de voisins d'un nœud donné. Elle est calculée en termes de pourcentage des liens existants par rapport au nombre de liens possibles présents dans le graphe.

4.8 Domaines d'utilisation

Les services d'allocation de ressources distribués relèvent des solutions de contrôle de base dans les systèmes d'exploitation des environnements distribués en général et mobiles en particulier. Ils peuvent aussi servir d'outils de base auxquelles font appel des applications utilisateurs aussi variées. Par conséquent, ils sont utilisés dans beaucoup de problèmes. L'exclusion mutuelle peut être à la base, par exemple, des problèmes de réplication et cohérences de données [137, 4, 72, 5], de configuration [103, 89, 138, 141], d'allocation de canaux [61, 102, 30, 135, 70], etc… La *k*- exclusion mutuelle et la *h* parmi *k* exclusion mutuelle peut servir dans tout problème où il s'agit de gérer des ressources exclusives à plusieurs exemplaires. Finalement, tous ces trois services peuvent servir de base pour la généralisation à l'allocation de plusieurs types de ressources chacune à plusieurs exemplaires.

5. Quelques Recommandations de conception

La conception d'un système informatique doit répondre à un certain nombre d'exigences liées aux propriétés de l'environnement sous-jacent et aux performances du système. Dans notre contexte, il s'agit des réseaux mobiles ad hoc qui sont connus par un certain nombre de contraintes non connues dans les environnement fixes. Dans cette thèse, les recommandations suivantes sont de rigueur.

A- Puisque chaque nœud d'une part, agit comme un hôte et un routeur et possède une autonomie d'énergie limitée et d'autre part est sujet à de fréquentes déconnexions, il est nécessaire d'éviter de centraliser le service au niveau d'un nœud spécifique et surtout de manière prolongée dans le temps. La *distribution des responsabilités de contrôle entre les différents nœuds du réseau* s'avère requise.
B- Les solutions aux problèmes d'allocation de ressources qui nous intéressent sont des solutions de contrôle qui sont considérées comme faisant partie du système d'exploitation. Puisqu'ils se situent juste au dessus de la couche de routage, ils peuvent utiliser les services de celle-ci pour le cheminement des messages; leurs performances dépendent en partie de celles de la couche de routage. Il peut être intéressant de *prendre en charge l'acheminement des messages par les protocoles qui résolvent ces problèmes d'allocations de ressources* afin d'assurer des performances meilleures.

6. Organisation de la thèse

Cette thèse est composée principalement de trois parties, chacune traitant un problème spécifique. Chaque partie commence par un état de l'art dans le domaine correspondant et ensuite une contribution de solution(s) au problème posé est explicitée. Avant d'entamer ces différentes parties, le Chapitre I décrit un certain nombre d'éléments préliminaires nécessaires à la suite de ce document. Il commence par mettre en évidence les motivations de cette thèse, ensuite défini les problèmes traités, leurs propriétés et certains choix de stratégies de conceptions et de simulations.

Dans la première partie, le Chapitre II décrit les solutions classiques principales et distribuées d'EM connues dans l'environnement distribué statique selon les deux approches principales. Ces solutions sont présentées d'une manière qui permet de regrouper ensemble celles présentant des outils et mécanismes semblables ou proches. Une comparaison sommaire de ces solutions est reprise dans un tableau récapitulatif (Table 1). Les solutions développées pour les réseaux mobiles ad hoc font aussi l'objet d'une description qui permet de ressortir leurs cotés encourageants et leurs insuffisances. Le Chapitre III décrit de manière méthodique un nouveau protocole d'exclusion mutuelle pour les réseaux mobiles ad hoc. Les idées de bases sont d'abord explicitées avant d'aborder son fonctionnement. Le fonctionnement est illustré à travers un exemple de déroulement qui montre la mise en œuvre des concepts et outils importants prévus dans ce protocole. Une discussion montrant certains aspects cachés de ce protocole suivie d'une étude sur la correction sont ensuite donnée. L'évaluation des performances de ce protocole est donnée à travers la présentation des résultats ainsi que les commentaires relatives aux différentes mesures effectuées.

Dans la deuxième partie, le Chapitre IV décrit les solutions les plus connues de la *k*- EM dans les systèmes distribués (selon les deux approches principales connues) et les concepts sur lesquelles elles s'appuient. Une simple comparaison est alors réalisée sur la base d'un tableau (Table 4) qui récapitule les éléments importants concernant ces protocoles. Cette description est suivie de celle plus ou moins en détail et critique de la seule solution de *k*- EM

connue dans les réseaux mobiles ad hoc. Le Chapitre V décrit un nouveau protocole de *k*-exclusion mutuelle dans les réseaux mobiles ad hoc qui adopte une approche qui se base principalement sur des heuristiques. La présentation commence par les outils et mécanismes clés du protocole; ensuite, la description du fonctionnement est donnée à travers les événements principaux formant ce protocole. Un exemple de déroulement qui montre la mise en œuvre des concepts et outils importants prévus dans ce protocole est ensuite décrit. Une extension qui tolère les pannes de nœuds et le partitionnement et leurs conséquences (i.e. perte et duplication de jeton(s)) est ensuite décrite. Une étude de correction est ensuite menée. La deuxième partie de ce chapitre est dédiée à la présentation des différents résultats de simulation effectués qui montrent ses performances et qui le situe par rapport à son unique prédécesseur.

Dans la troisième partie, le Chapitre VI décrit les solutions les plus connues de la *h* parmi *k* EM dans les systèmes distribués et les concepts y afférents. Une comparaison sommaire de ces solutions est reprise dans un tableau récapitulatif (Table 5). Cette description est suivie de celle plus ou moins en détail et critique des deux solutions de *k*- EM connues dans les réseaux mobiles ad hoc. Le Chapitre VII décrit un premier nouveau protocole de *h* parmi *k* exclusion mutuelle prévu également pour les réseaux mobiles ad hoc. Ce protocole reprend certains concepts et mécanismes du protocole décrit dans la première partie. Le plan de description est similaire à celui du protocole de la première partie. Le Chapitre VIII décrit un deuxième nouveau protocole qui se base sur des structures logiques à l'inverse du premier. La présentation commence par la définition de son approche. Par la suite, une description de ses différents outils et mécanismes illustrée par plusieurs exemples est donnée. Une discussion qui fait ressortir certains aspects cachés du protocole. La deuxième partie du chapitre expose les différents résultats de simulations avant d'effectuer une comparaison avec la première solution sur la base de ses simulations. Ce chapitre est terminé par une conclusion qui résume cette solution et projette quelques travaux futurs complémentaires.

Ce document se termine par une conclusion qui récapitule les éléments importants des différentes contributions et décrit certaines orientations futures pouvant en découler.

Chapitre II

L'exclusion mutuelle distribuée: Un état de l'art

1. Introduction

Le problème de l'exclusion mutuel a été largement étudié dans les systèmes distribués [113, 124, 125]. Ainsi, plusieurs solutions ont été proposées [143, 113, 125] selon deux approches essentielles : L'approche à permission [28, 37, 46, 76, 80, 120, 136] et l'approche à jeton [50, 77, 88, 93, 110, 118, 124, 133, 96, 104]. Les protocoles basés sur la permission imposent à un processus demandeur de SC de recevoir la permission d'accès à sa SC de tous les processus du réseaux ou d'un sous ensemble selon les outils utilisés. Dans l'approche à jeton, un message spécial (i.e. jeton) transporte le privilège d'accès à la section critique. Le seul détenteur de ce message est autorisé à accéder à sa SC, ce qui garantie l'exclusion mutuelle. Deux méthodes sont utilisées dans cette approche. Dans la première méthode, le jeton circule perpétuellement dans le réseau en suivant une structure prédéfinie (tel qu'un anneau [77]), ou des chemins définis dynamiquement selon des critères prédéfinis. Par conséquent, un processus désirant accéder à sa SC se contente d'attendre son '*tour'*, i.e. la réception du jeton. Dans la seconde méthode, un processus désirant entrer en SC est amené à demander le jeton; le problème de base est comment l'atteindre? Dans certains protocoles, la requête est adressée à tous les processus du réseau [50, 118, 133]. Dans d'autres, une structure logique est définie pour désigner le détenteur du jeton [110]. Dans ce dernier cas, la requête est envoyée sur un chemin (défini par la structure) qui mène au jeton. Quelques-unes des solutions de l'approche à jeton ont été adaptées pour fonctionner dans les réseaux cellulaires ainsi que dans les réseaux mobiles ad hoc.

L'exclusion mutuelle dans les réseaux mobiles ad hoc [12, 19, 20, 21, 38, 83, 148, 149, 150] a gagné d'intérêt ces dernières années, un certain nombre de solutions sont proposées. La majorité de ces solutions utilisent l'approche à jeton. Par cause de nouvelles caractéristiques des réseaux mobiles ad hoc, les solutions initialement prévues pour les réseaux fixes [143] ne peuvent pas s'appliquer aux environnements mobiles, mais ces dernières en sont en général une adaptation.

Etant donné que les solutions d'EM actuelles connues dans les réseaux mobiles ad hoc sont des adaptations de celles des systèmes distribués, ce chapitre sera dédié à la présentation d'un bon nombre de solutions de base connues dans la littérature des systèmes distribués selon une méthodologie de classification qui regroupe les solutions utilisant des outils et mécanismes semblables. Les solutions adoptées pour les réseaux mobiles ad hoc sont également présentées.

Ce chapitre est structuré de la manière suivante: La Section 2 décrit les solutions classiques principales et distribuées d'EM connues dans l'environnement distribué. Une comparaison sommaire de ces solutions est réalisée sur la base d'un tableau récapitulatif (Table 1). La Section 3 décrit les solutions développées pour les réseaux mobiles ad hoc en passant par quelque unes des réseaux mobiles avec infrastructures. Chaque solution des réseaux mobiles ad hoc décrite est succédée par une discussion qui ressort les avantages et les insuffisances de celle-ci. La Section 4 propose un certain nombre de recommandations à prendre en considération dans le développement de solutions d'EM dans les réseaux mobiles ad hoc. Finalement, la Section 5 conclue le chapitre.

2. L'exclusion mutuelle dans les systèmes distribués

2.1. Approche à permission

Les permissions ont pour rôle de garantir la propriété de sûreté. Un processus désirant l'accès à une SC, doit demander la permission à d'autres processus. Le droit d'entrer en SC est assujetti à l'obtention d'un nombre suffisant de permissions. Les solutions utilisant cette approche se distinguent par le nombre de permissions attendues et par la structure des ensembles de processus auxquels chacun demande ses permissions. Les conflits sont résolus par l'instauration d'un système de priorité ou par un mécanisme d'ordonnancement. Cependant, le problème posé est de trouver le nombre minimal de permissions que doit collecter un site pour entrer en SC. Les protocoles de cette approche doivent faire face à trois questions essentielles: Quel est le nombre minimal suffisant de droits (permissions) à collecter? Quels sont les processus qui peuvent donner ce droit? Comment ordonner les requêtes?

Les solutions à permissions peuvent être classées de différentes manières. Si on s'intéresse à la nature d'une permission [114], on peut distinguer deux familles de solutions. Pour cela, on se met au niveau d'un processus *i* qui reçoit des demandes de permissions de la part d'autres processus du système. Dans la première famille, un processus peut donner sa permission à plusieurs autres processus simultanément : Il gère individuellement chacun des conflits qui le mettent en compétition avec les autres processus. Si par exemple le processus *i* n'est pas intéressé par la SC, il répond favorablement à toutes les requêtes reçues. Si par contre il est prioritaire (par exemple, sa requête est ancienne) par rapport à certaines requêtes et non prioritaire par rapport à d'autres, il ne répond favorablement qu'à ces dernières. La sémantique associée à une permission envoyée par *i* à un autre processus *j* est donc : '' *en ce qui me concerne, vous pouvez entrer en SC*'' ; c'est alors la *permission individuelle*. Dans la seconde famille, un processus *i* sollicité par des requêtes, n'accorde sa permission qu'à l'une d'entre elles à un instant donné ; les autres restent en attente. Il est donc nécessaire à un autre processus *j* intéressé par la SC, qui a obtenu toutes les permissions demandées, qu'il les rende lorsqu'il sort de sa SC. Lorsque la permission obtenue de *i* est retournée à *i*, ce processus peut satisfaire une des requêtes en attente s'il en existe. La sémantique associée à une permission

envoyée par *i* à un autre processus *j* est donc: *en ce qui concerne tous les processus qui me demandent la permission, vous pouvez entrer en SC''*, c'est la *permission d'arbitre*.

2.2. Approche à jeton

Dans l'approche à jeton, un message spécial (i.e. jeton) transporte le privilège d'accès à la section critique ; son unicité dans le réseau garanti l'exclusion mutuelle. Deux méthodes sont utilisées dans cette approche. Dans la première méthode, le jeton *circule* perpétuellement dans le réseau en suivant une structure prédéfinie (tel qu'un anneau [77]) ou des chemins définis dynamiquement selon des critères prédéfinis. Par conséquent, un processus désirant accéder à sa SC se contente d'attendre son '*tour'*, i.e. la réception du jeton. Dans la seconde méthode, un processus demandant la SC est amené à *demander* le jeton. Le problème de base est comment l'atteindre ? Dans certains protocoles, la requête est adressée à *tous* les processus du réseau [50, 118, 133] ou à une partie de processus (selon des *heuristiques* [124]); dans d'autres, une *structure logique* est définie pour désigner le détenteur du jeton [110, 93]. Quelques-unes des solutions connues de l'approche à jeton dans les systèmes distribués ont été adaptées pour fonctionner dans les réseaux cellulaires ainsi que dans les réseaux mobiles ad hoc.

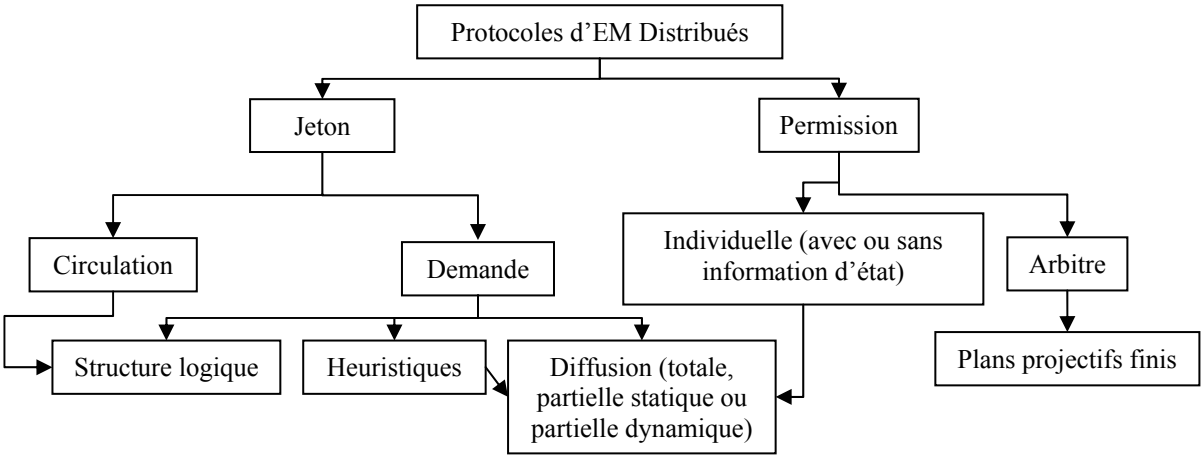


Figure 7 : *Une classification des protocoles d'EM distribués.*

2.3. Les solutions classiques

Bien entendu, le problème d'EM est très largement étudié dans les systèmes distribués. Dans cette section, il n'est pas question d'exposer toutes les solutions existantes, ce qui est aberrant. Le but recherché est de comprendre la tendance générale de ces solutions. Dans ce cadre, il est alors question d'illustrer cette tendance à travers la présentation des solutions *classiques* et de *base* connues dans cet environnement. Cette présentation va certainement donner une idée sur le fonctionnement général des solutions dans cet environnement. Pour ce faire, nous exposons les éléments importants des solutions ciblées selon une méthodologie de classification qui permet de regrouper les solutions utilisant des *outils* et *mécanismes* similaires ou proches. La Figure 7 donne une proposition de classification de ces solutions et la Table 1 donne un récapitulatif avec classement des solutions classiques. Les hypothèses communes à toutes les solutions sont : Le réseau sous jacent est connexe et composé de N processus fiables (le cas de pannes de processus sera indiqué), les liens sont fiables et bidirectionnels et chaque processus connaît ses voisins. Par la suite, nous nous plaçons au niveau d'un processus désigné par i.

2.3.1. Solution de Le Lann

Cette solution [77] fût l'une des premières solutions proposées avec l'approche à jeton. Les processus du système sont organisés selon un *anneau logique statique* unidirectionnel et le jeton circule de manière continue le long de cette structure. Quand un processus reçoit ce jeton, il accède à sa SC s'il le souhaite; à la sortie de la SC, le jeton est passé au processus successeur sur l'anneau. Si le jeton n'est pas souhaité, il est immédiatement passé au prochain sur l'anneau. Il est évident que la propriété de sûreté est assurée étant donné que le jeton est unique et il est soit chez un processus soit en transit. La propriété de vivacité est aussi assurée étant donné que d'une part, l'anneau contient tous les processus et d'autre part, le jeton circule le long de cet anneau et dans le même sens. Ce protocole coûte *N* messages au maximum pour exécuter une SC.

2.3.2. Solution à jeton de G. Ricart et A.K. Agrawala

Dans [118], G. Ricart and A. K. Agrawala ont proposé un protocole orienté *demande* de jeton qui requiert *0* ou *N* messages pour entrer en SC. La topologie du réseau est supposée *complète*. Chaque processus *i* entretien un compteur (S_i) permettant d'estampiller de manière locale (différentielle) ses propres requêtes. Les numéros de séquences des dernières requêtes reçues de tous les autres processus sont entretenus dans un tableau de taille *N*. Le jeton transporte un tableau qui mémorise le nombre d'accès à la SC de chaque processus.

Un processus *i* demandeur de SC envoie sa requête, estampillée avec son numéro de séquence local, à tous les autres processus. Un processus détenteur du jeton le lui envoie s'il n'est pas en SC. A la sortie d'un processus *i* de sa SC, il affecte le numéro de séquence de la requête qui vient d'être satisfaite à l'entrée dans la structure du jeton qui correspond à ce processus. Ensuite, il cherche dans le tableau qui mémorise le jeton, le premier processus *k* (*k* est choisi dans l'ordre *i+1*, …*N*, *1*..*i-1*) tel que le numéro de séquence de la dernière requête de *k* est supérieure au numéro de séquence mémorisé par le jeton lors de sa dernière visite à *k*, puis envoie le jeton à *k*. Cette solution n'est pas tout à fait Fifo mais assure la vivacité.

Basée sur celle de Ricart et Agrawala [118], la solution de I. Suzuki et T. Kazami [133] fait recours au transport par le jeton d'une file de requêtes et un tableau qui mémorise le jeton. La file transportée est mise à jour par la file locale de chaque processus visité par le jeton dans l'ordre ascendant de numéros des processus afin d'assurer la propriété de vivacité. Quand il s'agit de passer le jeton au prochain, le processus en tête de la file du jeton est sélectionné. Ce protocole engendre pratiquement les mêmes performances que son prédécesseur. Une modification de ce protocole a été apportée pour borner les numéros de séquences des requêtes à la valeur *L*. Ce protocole requière alors à un processus $L*N + (N-1)$ messages pour *L* entrées en SC.

S. Mishra et P. Srimani dans [88] ont rendu le protocole de I. Suzuki et T. Kazami [133] tolérant au pannes de sites, de liens et par conséquent à la perte de jeton. Ils ont introduit des mécanismes basés sur un *temporisateur* durant lequel un processus demandeur attend le jeton avant de lancer sa recherche. Ce qui permet de tolérer la panne d'un site. D'autres mécanismes pour la régénération du jeton en cas de perte et l'élimination de jetons dupliqués ont été utilisés. En absence de panne, ce protocole se comporte comme son prédécesseur et engendre $L*N+(N-1)$ messages pour une entrée en SC, où *L* est un entier >2.

J. M. Helary, N. Plouzeau et M. Raynal [49, 50] proposent un protocole à *diffusion totale* dans un réseau supposé de topologie *connexe* (pas forcément complète) et dans lequel chaque processus ne connaît que ses voisins. Pour atteindre le détenteur du jeton, un processus demandeur de SC diffuse dans le réseau une requête estampillée [76] à travers ses voisins selon la méthode de vague avec transport d'information de contrôle [114]. Cette information

(, quoi que partielle,) permet d'éviter de visiter un même processus plus d'une fois par la même requête. Au cours de son voyage, la route d'une requête (qui est un champ de celle-ci) se construit à chaque traversée de processus. Le détenteur du jeton reçoit nécessairement cette requête. Quand le jeton devient disponible, il est envoyé sur le chemin inverse du chemin suivi par la requête en tête de file du processus privilégié ; les estampilles des requêtes et les identités des processus concernés sont utilisées pour créer un ordre totale sur l'ensemble des requêtes présentes dans la file locale. Le nombre de messages nécessaires pour accéder à une SC est compris entre *N* et *(N+1)(N-1)* messages.

2.3.3. Protocole de M. Singhal

Le protocole de M. Singhal [124] a amélioré les performances de celui de I. Suzuki et T. Kazami [133] en engendrant au maximum *N* messages même pour les grandes charges de demandes. Il utilise une méthode d'heuristique permettant de déterminer l'ensemble de processus dont l'un d'eux détient ou va détenir éventuellement le jeton. Ainsi, une requête est adressée seulement à cet ensemble (i.e. *diffusion partielle à un ensemble dynamique*) au lieu de tous les processus; le cas échéant, cet ensemble contient tous les autres processus. Ce protocole fait recours à l'utilisation d'informations sur l'état du système pour essayer de réduire sa complexité. Pour ce faire, les informations concernant les processus demandeurs sont implicitement transmises à travers le jeton.

Chaque processus entretient deux vecteurs, un vecteur de connaissance d'état par rapport à la SC de tous les processus et un vecteur de numéros de séquences locaux connus de tous les processus; le jeton transporte aussi ces informations. Toutes ces informations sont mutuellement mises à jour à la réception du jeton ou à la réception d'une requête pour prendre les valeurs les plus récentes. La requête d'un processus demandeur est envoyée aux processus demandeurs de SC selon les informations entretenues localement. Le jeton est véhiculé au processus demandeur, à la sortie d'une SC ou à la réception d'une requête, selon une manière circulaire équivalente à celle de [133].

2.3.4. Protocole K. Raymond

Dans [110], K. Raymond propose un protocole basé sur une structure logique *statique* (i.e. les voisins d'un processus dans la structure logique ne changent pas,) d'arbre, bâtie sur les liens physiques du réseau, dont la racine est le processus détenteur du jeton. Ce protocole requière *O(Log(N))* messages pour accéder à une SC. L'arbre est maintenu par une structure de pointeurs distribués sur les processus et dirigées au processus détenteur du jeton. Quand un processus désire accéder à sa SC, il enfile sa requête et l'envoie au processus prochain sur le chemin du détenteur du jeton. Cette requête est retransmise de proche en proche jusqu'au processus privilégié, i.e. la racine de l'arbre. Notons que chaque processus transmis une seule requête commune à lui et à l'ensemble de ses voisins qui le pointent. Le processus détenteur du jeton, lorsqu'il aura à transmettre le jeton, choisi parmi ses voisins (qui ont transmis une requête) le premier dans sa file et lui envoie le jeton en mettant à jour son lien dans la structure d'arbre (afin de pointer le nouveau détenteur du jeton). Ainsi, de proche en proche, le chemin vers le jeton sera reconstruit à nouveau jusqu'au processus destinataire qui sera autorisé à exécuter sa SC. Le jeton suit alors le chemin inverse du chemin de la requête.

Le protocole ne requière pas un ordre particulier sur l'ordre des messages mais sert les requêtes selon l'ordre d'arrivée et chaque processus ne considère que son voisinage. L'auteur de ce protocole propose une discussion sur des mécanismes pour couvrir la panne de site (à condition que ses voisins ne tombent pas en panne), ce site ne peut pas être le détenteur du jeton ni ses voisins.

Dans [36], Y. Chang, M. Singhal et M. Liu ont développé un protocole qui améliore celui K. Raymond [110] en tolérant la panne de liens et de processus et en maintenant plusieurs chemins vers le détenteur du jeton, d'où l'utilisation de la structure de DAG. Un processus demandeur de SC possède des chemins alternatifs pour envoyer sa requête dans le cas de panne de liens ou de nœud sur l'un des chemins qui mènent vers le jeton. Le protocole tente d'éviter la formation de cycles lorsque le jeton retourne sur les liens inverses vers le processus demandeur de SC. En effet, chaque processus récepteur de jeton, en supprimant tous les liens sortants, demande à chacun des processus concernés par ce changement de le considérer comme un voisin sortant.

Dans [41], D. M. Dhamdhene and S.S. Kulkarni ont développé un protocole dans le but est de résoudre le problème du cycle qui reste posé dans le protocole de Chang, Singhal et Liu [36]. Ce protocole est *k*- résistant aux pannes, i.e. il tolère la panne de *k* sites/liens.

2.3.5. Solution de M. Maimi et M. Trehel

Dans [93], M. Naimi et M. Trehel ont proposé un protocole d'EM distribué orienté *jeton* et basé sur une structure logique *dynamique* (i.e. les voisins d'un processus dans la structure logique changent dans le temps,) d'arbre orienté, dont la racine est le dernier processus demandeur de SC. Ce protocole requière en moyenne *O(Log(N))* messages pour accéder à une SC. L'arbre est maintenu par une structure de pointeurs distribués sur les processus et dirigées au dernier processus demandeur de SC. Le protocole considère que chaque processus est capable de communiquer directement avec n'importe quel autre processus. Chaque processus *i* entretient deux pointeurs $pere_i$ et $suivant_i$; le premier indique le processus auquel la demande d'accès en SC doit être véhiculée et le second indique le processus auquel le jeton doit être envoyé à la sortie de la SC. Le point clé de cette solution est le mécanisme qui permet la mise à jour de la structure pour servir la séquence de requêtes des processus. En fait, la suite des variables *suivant* forme une structure linéaire orientée du processus qui détient le jeton en passant par chacun des processus demandeurs de SC dans l'ordre d'arrivée de leurs requêtes et en finissant par le dernier processus demandeur jusqu'à ce moment. Un nouveau processus demandeur est considéré comme le suivant du dernier processus dans cette suite linéaire. Le jeton va donc passer par chacun des processus demandeurs dans l'ordre indiqué. Quand un processus *i* sort de sa SC, il envoie le jeton à son prochain (s'il existe,) indiqué par $suivant_i$, de ce fait $suivant_i$ devient nil. Dans le cas où $suivant_i = nil$, le jeton est gardé localement au repos. En échange, lorsqu'un processus *i* demande la SC, il envoie sa requête uniquement à son processus $pere_i$ qui le fait suivre jusqu'au processus racine de l'arbre. Les chaînages sont mis à jour de telle sorte que le processus $pere_i$ et le processus racine courante pointent le processus *i*; ce qui veut dire que *i* devient la nouvelle racine de l'arbre. Il est clair que ce protocole est vulnérable aux pannes de liens et de nœuds par cause du caractère dynamique de sa structure.

2.3.6. Solution de L. Lamport

Le protocole de L. Lamport [76] est orienté *permission individuelle avec information d'état* et avec *diffusion totale*. Un processus demandeur de SC est amené à enfiler sa requête estampillée à l'aide des horloges de L. Lamport [76] et à l'envoyer avec son estampille aux *N-1* processus du système supposé de topologie *complète*. (En réalité, tous les messages de service d'EM sont estampillés et leurs estampilles servent aux prises de décisions.) Par la suite, il attend la permission de tous ces processus. Un processus recevant une requête, l'insère dans sa file locale et renvoie systématiquement sa permission estampillée à l'émetteur. Un processus demandeur accède à sa SC, s'il reçoit une réponse de tous les *N-1* processus et sa requête est plus ancienne parmi celles existant dans sa file d'attente. A la fin

de l'exécution de la SC, il supprime sa requête de sa file et envoie un message de libération accompagné de l'estampille de la requête qui vient d'être satisfaite à tous ces *N-1* processus. Ceci permet d'une part, aux autres processus de supprimer sa requête de leurs files respectives et d'autre part, à un processus qui trouve sa requête ancienne d'accéder à sa SC. Ce protocole requière *3*(N-1)* messages par SC. Il est clair que la décision d'accès à la SC est prise *localement* sur la base *d'information d'état local*. La demande de permission sert à informer le récepteur de la requête de l'émetteur et l'estampille de la réponse sert à assurer que la prochaine demande de ce processus émetteur de la réponse soit plus récente que celle du processus entrant en SC.

Dans [117], G. Ricart et A. K. Agrawala proposent un protocole, orienté *permission individuelle* avec *diffusion totale*, qui améliore les performances de celui de L. Lamport [76] à *2*(N-1)* messages par SC. Un processus demandeur de SC envoie une requête estampillée aux *N-1* processus du système et attend leurs réponses. En recevant toutes les réponses attendues, le processus demandeur accède à sa SC. Un processus recevant une requête (processus *source* de la permission), retourne immédiatement une réponse s'il n'est pas lui-même demandeur ou sa requête est plus récente. Dans le cas contraire, il diffère sa réponse jusqu'à ce que sa requête soit satisfaite auquel cas, il envoie son accord à toutes les requêtes différées. Ce retardement permet de rendre le message de libération inutile.

Dans le protocole de O. Carvalho et G. Roucairol [28], qui améliore les performances de celui de G. Ricart et A. K. Agrawala [117] à un nombre de messages inférieur ou égal à *2*(N -1)*, un processus qui désire entrer en SC n'envoie sa requête estampillée qu'aux processus qui ont accédé à la SC depuis sa dernière demande. Autrement dit, un accord acquit par un processus est gardé et utilisé pour les prochaines entrées en SC jusqu'à ce qu'il soit réclamé.

Dans [120], B. A. Sanders développe un protocole générique orienté *permission individuelle à diffusion partielle ou totale*. Son protocole se base sur une structure d'information, entretenue au niveau de chaque processus i, constituée des ensembles I_i, ensemble d'information et R_i, ensemble de requête. Un autre ensemble S_i des processus sur lesquels des informations sont gardées est entretenu : $j \in S_i$ si $i \in I_i$. Parallèlement, $CSSTAT_i$ indiquant au mieux selon i, le processus appartenant à S_i en SC ou un état libre est également entretenue. Tous les messages échangés entre les processus sont estampillés [76]. Un processus désirant accéder à sa SC envoie sa requête estampillée [76] à tous les éléments de son ensemble R_i et attend la réception de la permission de tous ces processus avant d'entrer en SC. La sémantique associée à cette requête est : ''selon vos informations, est-il possible d'accéder à ma SC?''. A la sortie de la SC, un message de libération est envoyé à tous les éléments de I_i. Un processus recevant une requête l'insère dans sa file locale (triée selon les estampilles des requêtes); ensuite, si $CSSTAT_i$ indique que la SC est libre, un message d'accord est envoyé au processus j en tête de file (qui sera supprimé de cette file) et si j est un élément de S_i, il est affecté à $CSSTAT_i$. Le message d'accord signifie ''selon mes informations, vous pouvez entrer en SC''. A la réception d'un message de libération, $CSSTAT_i$ devient libre. Ensuite, tant que la file n'est pas vide et $CSSTAT_i$ n'indique pas un élément de S_i, un message d'accord est envoyé au processus j en tête de file (qui sera supprimé de celle-ci) et si j est un élément de S_i, il est affecté à $CSSTAT_i$.

Dans le cas où les ensemble I_i, R_i et S_i, sont statiques et selon la manière de construire ces ensembles, plusieurs protocoles à permission peuvent être générés: Par exemple, un protocole centralisé, le protocole de G Ricart et A. K. Agrawala [117], le protocole de M. Maekawa [80]. Les ensembles indiqués peuvent aussi être dynamiques (cette idée est issue du protocole de O. Carvalho et G. Roucairol [28]), leurs changement durant le fonctionnement du protocole doit obéir à des règles de correction. Dans le cas d'ensembles statiques, si une

panne de site est détectée des mises à jour des structures sont envisagées selon un mécanisme de tolérance aux pannes.

2.3.7. Solution R. S. Thomas

Dans [137], R.S. Thomas propose un protocole d'EM dans lequel un processus demandeur de SC est admet à accéder à sa SC s'il obtient la permission de la *majorité* des processus votants du système. La sûreté est bien assurée puisqu'au plus un processus est capable d'avoir cette majorité de votants. En supposant que plus que la moitié des processus continuent à s'exécuter, le service d'EM reste valable même si moins que la moitié des processus tombent en panne; c'est un protocole très résistant aux pannes !

D. K. Gifford dans [46], propose comme généralisation à cette méthode de majorité un protocole à *votes pondérés*. Un processus doit collecter la majorité du nombre total de votes pour accéder à sa SC. Etant donné que chaque processus peut avoir plus d'une voix selon la fiabilité de la machine sous jacente, la disponibilité du service d'EM augmente. Par conséquent, la méthode de majorité de Thomas devient un cas spécial par le fait que chaque processus dispose d'une seule voix. Remarquons que le nombre de processus avec lesquels un processus doit communiquer pour l'invocation d'une SC peut être contrôlé par la manière d'affectation des voix.

Dans [37], S. Y. Cheung, M. Ahamad et M. H. Ammar proposent une méthode de *vote multidimensionnel* comme extension à la méthode de vote pondéré. Les voix affectées à un processus sont représentées par un vecteur multidimensionnel.

2.3.8. Solution de Maekawa

Le protocole de M. Maekawa [80] est orienté *permission d'arbitre* et engendre entre $3*\sqrt{N}$ et $5*\sqrt{N}$ messages par SC. Il structure les processus du réseau supposé de topologie complète en *coterie*, ensemble de *quorums* [45, 60] qui se basent sur des concepts mathématiques de plans projectifs finis [7]. A chaque processus est associé un quorum, ensemble de processus seulement desquels la permission d'entrée en SC est demandée.

Concept de coterie et quorum

Le concept de coterie a été proposé par H. Garcia-molina et D. Barbara dans [45], il permet de diminuer le nombre de messages pour chaque entrée en SC.

Soit P l'ensemble de tous les processus et l'ensemble $C= \{Q_1, Q_2,...,Q_m\} \neq \phi$ *où* Q_i est un sous-ensemble de P appelé *quorum*, C est une *coterie* si et seulement si les conditions suivantes sont vérifiées:

a- Pas d'ensemble vide : $\forall i \quad i \in 1..m, \quad Q_i \neq \phi;$

b- Propriété d'intersection : $\forall i, j \in 1..m, \quad Q_i \cap Q_j \neq \phi;$

c- Minimalité : $\forall i, j \ (i \neq j) \in 1..m, \quad Q_i \not\subset Q_j$ *et* $Q_i \neq Q_j$.

La sûreté est assurée par le fait que deux quorums distincts ne sont jamais disjoints. Les processus d'intersection entre deux quorums servent d'arbitres pour satisfaire les différentes invocations. La structure de quorums peut être construite de différentes manières, voir [3, 99, 101].

Principe du protocole

Le protocole de M. Maekawa [80], regroupe les processus dans des sous ensembles S_i telles que les propriétés suivantes sont satisfaites :

a- $\forall i,j \;\; 1 \leq i,j \leq N, \quad S_i \cap S_j \neq \phi;$

b- $\forall P_i \in P$, $\exists S_i / P_i \in S_i$, $\quad 1 \leq i \leq N,$

c- $| S_1| = | S_2| = ... = | S_N| = K;$

d- $\forall i, \; 1 \leq i \leq N, \; \exists D \; S/ \; i \in S$ où D est un entier.

- La propriété (a) assure l'existence d'au moins un processus commun entre deus sous ensembles quelconques. Ce processus va servir d'*arbitre* afin d'assurer la sûreté.
- La propriété (b) définie au processus *i* l'ensemble des processus desquels la permission d'entrée en SC doit être obtenue.
- La propriété (c) implique que chaque processus doit envoyer et recevoir le même nombre de messages pour accéder à sa SC.
- La propriété (d) implique que tous les processus servent d'arbitres pour le même nombre de processus.

Lorsqu'un processus *i* désire entrer en SC, il tente de *verrouiller* tous les membres de son quorum S_i en envoyant à chacun (y compris lui-même) une requête estampillée [76]. S'il réussi à verrouiller tous les processus, alors il exécute sa SC sinon il attend que les membres déjà verrouillés, par d'autres processus, soient libérés pour les capturer et les verrouiller. Un processus ne peut attribuer sa permission qu'à un processus à la fois (c'est le principe d'arbitre). A la réception d'une requête d'un processus *i*, chaque processus *k* vérifie s'il n'a pas encore donné sa permission (n'est pas verrouillé). Dans ce cas, le processus *k* envoie sa permission au processus *i* (donc devient verrouillé) à travers un message *Locked*; autrement, il place la requête en queue de file. Par la suite, si le processus *k* possède dans sa file une requête d'un processus *j* ayant un numéro de séquence plus petit que celui du processus *i*, il envoie à *i* un message *Failed*. Autrement, un message *Inquire* est envoyé au processus qui a verrouillé *k* pour savoir s'il a réussit à verrouiller tous les membres de son groupe. L'envoi de ce message n'est pas nécessaire si le processus *k* a déjà envoyé un message *Inquire* (pour le compte d'un autre processus) pour lequel il n'a pas encore reçu de réponse. Quand un processus reçoit un message *Inquire*, il retourne un message *Relinquish* s'il a déjà reçu au moins un message *Failed*. Quand un processus ayant déjà émis une requête aux membres de son groupe reçoit au moins un message *Failed*, il retourne un message *Relinquish* à tous les processus qui lui ont envoyés *Inquire* pour leur signifier qu'il n'a pas réussi à verrouiller tous les processus de son groupe, ce qui permet de les libérer. Cette réaction permet d'éviter des situations d'interblocage. Si le processus a réussi à verrouiller tous les processus de son groupe, il leur envoie un message *Release* après avoir exécuté sa SC, ce qui permettra de les déverrouiller.

Le protocole de M. Maekawa [80] produit des mécanismes permettant de remédier à la panne de processus au prix d'une complexité additionnelle en messages.

Dans [22], A. Bouabdallah et J.C. König ont développé un protocole qui améliore celui de M. Maekawa [80] pour le rendre tolérant à *t-1* pannes de processus, où $t >= 1$. Ces auteurs proposent une méthode de construction des quorums de telle sorte que *cardinal* $(Q_i \cap Q_j) >= 2t$. Ainsi, chaque processus *i* désirant accéder à sa SC, doit attendre seulement *cardinal* $(Q_i) - t + 1$ permissions au lieu de *cardinal*(Q_i). En présence de pannes arbitraires de $(t-1)$ ou moins de processus, la cardinalité de l'intersection des quorums est d'au moins $t+1$ processus. D'autre part, étant donné que chaque processus donne sa permission à un seul autre processus, l'exclusion mutuelle est assurée en présence du nombre indiqué ci-dessus de pannes. Il est clair que le protocole dans [22] ne nécessite aucune procédure de recouvrement après panne. Ce protocole requière $O(\sqrt{(tN)})$ messages par entrée en SC et $\sqrt{(2t)}$ messages pour tolérer *t-1* ou moins pannes de processus.

Approche	Outils &mécanismes		Autres	Protocole	Objectif	Messages	Observations
Jeton	Circulation		-Anneau logique statique	Le Lann [77], 1977	-	*0 ou N*	-Topologie complète
	Demande	Diffusion	totale statique	Ricart-Agrawala [118], 1983	-	*0 ou N*	-Topologie complète -estampillage local - Suzuki-Kazami [133] est modifié pour borner les numéros de séquences de requêtes.
				Suzuki-Kazami [133], 1985	Améliore Ricart-Agrawala [118]		
				Mishra-Srimani [88], 1990	Introduit tolérance aux pannes dans Suzuki-Kazami [133]	*L*N+(N-1)* en absence de pannes	-Topologie complète -Estampillage local -Tolérance à la panne d'un site, régénération du jeton et élimination de copie dupliquée du jeton.
				Helary-Plouzeau-Raynal [50], 1986	-	Connexe *: >=N* et *<=(N+1)(N-1)* Arbre :>=N et <= (n-1)+d Anneau *: >=N* et *<=2N* Complète : N	-Topologie connexe -Estampillage local
		Structure logique	statique	Raymond [110], 1989		*O(log(N))*	- Topologie connexe - Arbre statique
				Chang-Singhal-Liu [36], 1990	Améliore Raymond [110],	*O(2log(N))*	-Topologie connexe - Recouvre panne de site et de lien. -Utilise DAG
				Dhamdhene- Kulkarni [41], 1994	Améliore Chang-Singhal-Liu [36],	*>=O(d.k) et <=O(d.k^2+e.k)]*	-Résistant à k liens/sites -Utilise DAG
			dynamique	Naimi-Trehel[93], 1987		O(log(N))	- Topologie complète
		Heuristiques	Diffusion partielle et dynamique	Singhal [124], 1989	Améliore Suzuki-Kazami [133]	*<=N*	-Topologie complète -estampillage de Lamport -Discute effet de pannes et propose des mécanismes de recouvrement.

Permission	Individuelle Information d'état	Diffusion totale	Lamport [76], 1978	-	*3*(N-1)*	-Topologie complète -Estampillage de Lamport
	Individuelle	Diffusion totale	Ricart-Agrawala [117], 1981	Améliore Lamport [76]	*2*(N-1)*	-Topologie complète -Estampillage de Lamport
		Diffusion partielle	Calvalho-Roucairol [28], 1983	Améliore Ricart-Agrawala [117],	*<= 2*(N-1)*	-Topologie complète -Estampillage de Lamport
		Diffusion totale	Thomas [137], 1979	Améliore plusieurs protocoles à permission	Entre (N+1) et 2*(N-1)	-Topologie complète - Très résistant aux pannes de sites
			Gifford [46], 1979	Généralise et améliore Thomas [137]	Entre (N+1) et 2*(N-1)	
			Cheung-Ahamad-Ammar [37], 1990	Généralise Gifford [46],		
	Individuelle ou arbitre selon les structures d'informations construites.	Diffusion partielle ou totale selon les structures d'informations construites.	Sanders [120], 1987	Généralise plusieurs protocoles à permission	*|Ii- {i}|+2*(|Ri-{i}|)*	-Topologie complète -Estampillage de Lamport -Entretient structures d'information statique ou dynamique, -Tolérance à la panne de site
	Arbitre	Quorums statiques	Maekawa [80], 1985	-	*>=3*√N , <=5*√N*	-Topologie complète -Estampillage de Lamport - Tolérance aux de sites
			Bouabdallah-König [22], 1992	Améliore Maekawa [80]	*O(√ (tN) et √ (2t) pour t-1 pannes de sites*	-Topologie complète -Estampillage de Lamport - Tolérance aux de sites

où
d : diamètre de l'arbre.
L : entier *>2* borne maximale pour un nombre
e : nombre de liens du réseau.
t : entier *>=1*.

Table 1 : *Classement de solutions classiques d'exclusion mutuelle distribuée.*

3. Exclusion mutuelle dans les réseaux mobiles ad hoc

Dans la suite de ce chapitre, nous décrivons les protocoles d'EM développés pour les réseaux mobiles ad hoc et notamment certains connus des réseaux avec infrastructures.

3.1. Protocoles de Badrinath et *al*

Dans [8], B. R. Badrinath, et *al* ont proposé deux protocoles d'exclusion mutuelle pour les réseaux cellulaires. Le premier est une adaptation aux réseaux cellulaires de celui proposé par L. Lamport [76]. Le second est une adaptation aux réseaux cellulaires de celui proposé par Le Lann [77]. Rappelons qu'un réseau cellulaire est formé par deux parties : Une partie filaire et une partie mobile dans laquelle chaque nœud mobile (ou simplement nœud) communique via une interface de communication sans fil. Quand un nœud désire communiquer avec un autre nœud, il envoie un message à la *station de base* à laquelle il est rattaché, celle-ci se charge de transmettre le message à travers le réseau filaire à la station de base du nœud destinataire, et finalement le message est délivré à sa destination. Les nœuds mobiles sont sujets à plusieurs contraintes, notamment en terme de durée de vie de leurs batteries, ce qui les mène à opérer en mode veille ou à se déconnecter complètement du réseau. De ce fait, il est recommandé de réaliser la plupart du calcul sur la partie statique du réseau pour le compte de ces nœuds mobiles. Le rôle de ces nœuds se limite alors à l'initialisation du calcul et à la réception des résultats; les protocoles de [8] se basent sur cette idée.

Le protocole de L. Lamport est adapté à l'environnement mobile en transférant les communications et les calculs du protocole vers la partie statique du réseau. Le service d'EM est assuré avec une amélioration sensible de la complexité en messages puisque les stations de bases opèrent comme un *proxy* pour les nœuds qui leurs sont rattachés. Chaque nœud est remplacé par sa station de base qui maintient les structures nécessaires au fonctionnement du service. Les stations de base s'échangent des requêtes estampillées, les messages de réponses et de libération et assurent l'EM à la place des nœuds mobiles. Chaque nœud mobile se contente d'envoyer sa requête à sa station de base, d'attendre l'accord d'entrée en SC de sa station de base et d'envoyer le message de libration de sa SC à sa seule station de base. Si un nœud se déconnecte avant de recevoir une autorisation d'accès en SC, sa station de base libère la ressource au profit d'un autre nœud qui lui est rattaché. Si le nœud se déconnecte alors qu'il est en SC, il doit se reconnecter pour envoyer son message de libération de ressource.

L'adaptation du protocole de Le Lann à l'environnement mobile est réalisée en organisant les stations de bases (au lieu des nœuds mobiles) en un anneau logique unidirectionnel. Chaque station de base entretient une file FIFO contenant les requêtes des nœuds mobiles qui lui sont rattachés. Le jeton circule le long de l'anneau. Quand une station de base le reçoit, elle l'envoie à tour de rôle à chacun des nœuds dont une requête est présente localement et ceci jusqu'à leur épuisement. Ensuite, il est envoyé à la station de base successeur sur l'anneau. Pour assurer qu'un nœud accède à sa SC une seule fois durant le tour (et donc assurer la vivacité), un compteur de tour est inclus dans le message du jeton. Cette adaptation engendre plusieurs avantages: Elle diminue sensiblement la communication avec les noeuds mobiles et réduit le coût de la traversée du jeton. Aussi quand un nœud mobile se déconnecte, il n'est pas nécessaire de réorganiser l'anneau. D'autre part, un nœud mobile qui opère en mode veille n'est interrompu que pour satisfaire sa requête précédente.

3.2 Protocole de J. Walter et S. Kini

Dans [149], J. Walter et S. Kini ont proposé un protocole d'EM basé jeton dérivé de plusieurs autres protocoles: Le protocole de routage basé arbre de E. Gafni et D. Bertsekas [44], le protocole présenté dans [36] et d'autres idées de [41, 110]. Ce protocole défini une structure mappée sur la topologie réelle du réseau, représentée par un DAG (Direct Acyclic Graph) ou graphe acyclique dirigé de pointeurs orientés jeton, maintenant plusieurs chemins menant au nœud détenteur du jeton. Le protocole est convenable aux caractéristiques des réseaux mobiles ad hoc puisqu'il ne requière aux nœuds de maintenir que des informations sur le voisinage immédiat. Dans l'absence de mouvement de nœuds, le protocole se comporte similairement à celui de K. Raymond [110] dans l'acheminement des requêtes au nœud détenteur du jeton et dans l'envoi du jeton au nœud demandeur de SC. De plus, à chaque nœud est associée une information d'*élévation* qui joue un rôle fondamental dans l'entretient de la structure de DAG notamment dans le cas de pannes de liens dans le but d'avoir continuellement comme racine de l'arbre le nœud détenteur du jeton. Ce protocole suppose que les pannes de liens sont dues uniquement aux mouvements des nœuds et considère les liens de communication bidirectionnels et fiables. Les nœuds meuvent avec une vitesse limitée de telle sorte qu'ils ne changent pas de liens durant l'activation du protocole et durant la transmission d'un message. Le protocole commence par construire la structure de DAG, qui maintient des pointeurs logiques au niveau des nœuds qui sont dirigés vers le détenteur du jeton. Ces pointeurs sont définis selon l'élévation de chaque nœud vis-à-vis de son voisinage. L'élévation d'un nœud i est de la forme (α, β, i) et augmente avec la distance du nœud détenteur du jeton (voir [44]). Les voisins d'un nœud sont regroupés en deux ensembles selon le sens de connexion avec celui-ci dans la structure du DAG, l'ensemble de ceux ayant des connexions entrantes et l'ensemble de ceux ayant des connexions sortantes, dans le but de maintenir plusieurs routes vers le détenteur du jeton. Les requêtes sont acheminées au détenteur du jeton le long de l'arbre, et chaque nœud envoie au maximum une requête commune à lui et aux voisins entrants (lors de la réception d'une première requête). Pour se souvenir de l'ensemble des requêtes reçues, celles-ci sont enregistrées dans une file locale Fifo. Celle-ci sert à l'acheminement du jeton vers le nœud demandeur. A la réception du jeton, le nœud qui détecte son identité en tête de file devient le nouveau détenteur en modifiant son élévation qui devient la plus petite que celle de tous les nœuds voisins ; ensuite ce nœud accède à sa SC. Puisque le nœud détenteur du jeton doit être toujours la racine du DAG, une réorganisation partielle du DAG est alors nécessaire.

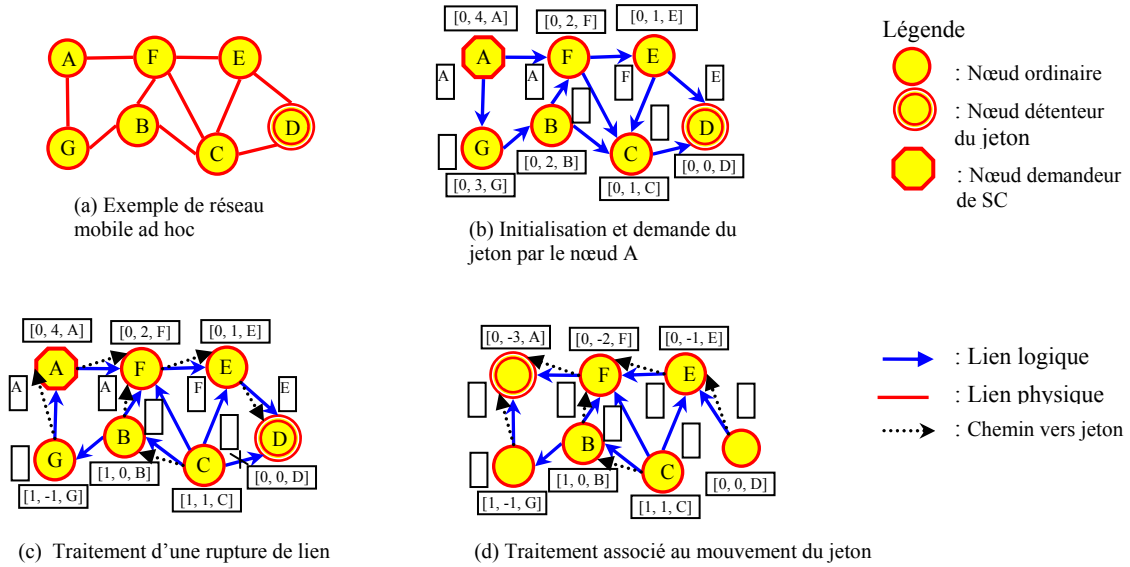
Quand le nœud détenant le jeton sort de sa SC, il envoie le jeton au nœud dont l'identité se trouve en tête de file s'il en existe ; autrement il est gardé localement. Dans le cas normal, le jeton est délivré au nœud demandeur sur le chemin inverse de celui de la requête. Si un ou plusieurs liens sur le chemin inverse sont rompus, une recherche du nœud prochain est lancée par le détenteur du jeton en diffusant un message de recherche sur tous les liens entrants. Chaque nœud recevant ce message pour la première fois, le propage de la même manière. Quand le nœud recherché est trouvé, le jeton est de nouveau acheminé.

Quand un nœud détecte une perte de son dernier lien sortant, la conséquence en est la perte de chemin vers le détenteur du jeton. Ce nœud invoque alors une réorganisation partielle du DAG en utilisant la méthode décrite dans [44] qui évite la formation de cycles. Généralement, une rupture de liens peut se produire à tout instant, le protocole produit des mécanismes lui assurant le fonctionnement normal.

Quand un nouveau lien est détecté, les deux nœuds adjacents de ce nouveau lien s'échangent les messages nécessaires afin de réaliser les modifications nécessaires sur les

liens sortants et entrants. Cette opération peut engendrer la perte de dernier lien sortant de chacun des deux nœuds, une réorganisation partielle du DAG est alors effectuée.

La Figure 8(a) montre un simple réseau mobile ad hoc. La Figure 8(b) montre l'état du réseau avec liens logiques après l'initialisation du protocole. Le nœud *D* est le détenteur du jeton (il n'a pas de liens sortant) et tous les autres nœuds pointent le nœud *D*. Le long du chemin vers le nœud détenteur du jeton, les élévations diminuent. Le nœud *A* lance une requête sur le chemin du jeton qui est enfilée au niveau du nœud *F*, qui la transmis au nœud *E*, qui l'enfile à son niveau et la transmis au nœud *D*, qui l'enfile localement.



(a) Exemple de réseau mobile ad hoc

(b) Initialisation et demande du jeton par le nœud A

(c) Traitement d'une rupture de lien

(d) Traitement associé au mouvement du jeton

Figure 8: *Exemple d'exécution du protocole.*

Dans l'exemple donné dans la Figure 8(c), le lien physique *C* - *D* est rompu suite à l'augmentation de la distance qui sépare le nœud *C* du nœud *D*. Le lien perdu est le dernier lien sortant du nœud *C* ne détenant pas le jeton. Le nœud *C* est amené à réorganiser ses liens, change son élévation. La conséquence en est la perte du dernier lien sortant du nœud *B*, de même que le nœud *G*. L'arbre est alors partiellement réorganisé. La Figure 8(d) montre le résultat du mouvement du jeton du nœud *D* au nœud *A* causant le changement de l'élévation des nœuds *E*, *F* et *A*. Ces changements assurent que tous les liens logiques sont orientés du nœud ayant la plus grande élévation au nœud avec la plus petite.

Le protocole proposé garantie les propriétés de sûreté et de vivacité (voir [149] pour la preuve). En terme d'analyse de complexité, le protocole proposé est comparé à celui duquel il est dérivé présenté dans [110] qui est exécuté au dessus de la couche de routage TORA [105]. Quand les nœuds ne sont pas mobiles, les deux protocoles engendrent des performances proches (*O(p)* messages, où *p* est le chemin le plus long du DAG) étant donné que les chemins logiques entretenus par le protocole d'EM correspondent à ceux entretenus par le protocole de routage. Mais concernant la complexité en temps, le premier protocole est meilleur étant donné que le second utilise la structure de couche protocolaire. Quand la mobilité est considérée, le premier protocole engendre moins de messages et moins de temps puisque le second est indépendant du protocole de routage utilisé.

Discussion

Le protocole développé dans [149] présente un certain nombre de mécanismes intéressants. Premièrement, les liens de la structure de DAG entretenue correspondent à tout moment aux liens physiques du réseau; ce qui participe à l'optimisation des messages de contrôle. Deuxièmement, chaque nœud n'entretient des informations que sur son voisinage; ce qui améliore aussi la complexité en messages. Troisièmement, ce protocole ne fait pas recours aux services de la couche de routage; par conséquent, ses performances ne sont pas dépendantes de celle de cette couche.

En échange, Le protocole définit dans [149] suppose d'une part, une mobilité réduite de nœuds et d'autre part, ne tolère pas la perte du jeton, la panne (ou la déconnexion de nœuds) et le partitionnement. Le nombre de nœuds du réseau est supposé alors fixe et connu de tous les nœuds du réseau. Néanmoins, la simulation effectuée montre clairement que l'utilisation de la couche de routage n'est pas efficace relativement au protocole qui prend lui-même en charge l'acheminement des messages. Il serait intéressant de savoir à quel degré la réorganisation de la structure de DAG affecte la structure originale. Autrement dit, est-ce qu'elle est partielle ou totale ? Le degré de réorganisation possède un lien direct avec la scalabilité du réseau.

3.3. Le protocole de Walter, Welch et Vaidya

Dans [150], J. Walter, J. Welch et N. Vaidya proposent une solution alternative à celle décrite dans [149], mais utilise toujours comme protocole de base celui de K. Raymond [110]. Les mêmes hypothèses que celles du protocole précédent sont supposées. Le protocole développé dans [150] utilise aussi une structure de DAG de pointeurs orientés jeton mappée sur les liens physiques. Chaque nœud du réseau entretient seulement des informations liées à son voisinage immédiat. Un nœud choisit *dynamiquement* son voisin de plus petite élévation comme nœud prochain préféré sur le chemin du détenteur du jeton. Quand un lien se perd, le nœud concerné re-route sa requête sur un autre chemin. Toutes les requêtes qui arrivent au niveau du nœud détenteur du jeton sont traitées de manière symétrique. Les requêtes sont acheminées au nœud détenteur du jeton de la même manière que dans [149]. Le jeton est délivré sur le chemin inverse de la requête vers son destinataire. Le nœud détenteur du jeton est toujours de poids le plus faible de tous dans le DAG. La différence essentielle avec le protocole de [149] est que dans ce nouveau protocole quand un nœud désire envoyer le jeton, il trouve nécessairement le chemin de retour vers son destinataire. Cela suppose qu'une panne de lien est traitée par le nœud demandeur (puisque chaque nœud prend en considération l'évolution de son voisinage). Cette technique permet d'éliminer la complexité additionnelle en messages de recherche de nœud destinataire. Naturellement, une réorganisation partielle du DAG peut être nécessaire lors de la circulation du jeton et lors de changements de liens. Notons aussi que ce nouveau protocole utilise moins de types de messages que celui de [149].

Quand un nœud détecte une rupture d'un lien sortant qui n'est pas le dernier, il re-route seulement sa requête si le prochain nœud sur le chemin du jeton est perdu. Dans le cas où il s'agit du dernier lien qui est perdu (ce qui veut dire que le chemin vers le détenteur du jeton est rompu,) il invoque une réorganisation partielle du DAG et par la suite sa requête est re-routée. Quand un nouveau lien est détecté, les deux nœuds concernés s'échangent de l'information pour leurs permettre éventuellement de mettre à jour leurs deux ensembles de liens et de re-router leurs éventuelles requêtes; la réorganisation partielle du DAG peut être alors nécessaire. Les propriétés de sûreté et de vivacité sont prouvées. L'étude de performance est réalisée de la même manière que dans [149].

La simulation a permis la comparaison des performances du protocole avec celui de K. Raymond [110] exécuté au dessus de la couche de routage (qui produit un chemin minimal entre deux paires quelconques de nœuds). Deux paramètres sont considérés: Le temps d'attente moyen par entrée en SC et le nombre moyen de messages par entrée en SC. Concernant le premier paramètre, les résultats montrent que le protocole de [150] donne de meilleurs résultats dans le cas de mobilité de nœuds et notamment lorsque la charge (ou nombre) de demandes augmente. Les résultats du protocole de K. Raymond deviennent plus médiocres dans le cas de faible connectivité puisqu'il considère les liens logiques, ce qui augmente les chemins réels vers le détenteur du jeton. En échange les deux protocoles exhibent des résultas très proches dans le cas de mobilité nulle. Concernant le deuxième paramètre, le protocole de Raymond envoie moins de messages par entrée en SC dans toutes les simulations réalisées et concernant deux valeurs particulières de connectivités.

Discussion

D'après les mécanismes alternatifs utilisés dans ce protocole, il semble que ce nouveau protocole améliore les performances de l'ancienne version étant donnée qu'il évite de faire recours à la diffusion de messages de recherche du nœuds destinataire. Cependant, une simulation pourrait mieux clarifier cet apport éventuel. Cependant, les mêmes hypothèses que l'ancienne version sont reprises dans ce protocole. Les remarques sur l'ancienne version du protocole sont aussi valables pour ce nouveau protocole étant donné que le fonctionnement en globalité reste inchangé. Néanmoins, les auteurs du protocole ne donne aucune précision sur un problème qui se produit comme conséquence dans le cas ou une requête est reroutée. Imaginons la situation suivante : Le nœud *4* a envoyé sa requête sur le chemin du jeton *3*, *2*, *1*, *0*; le nœud *0* étant le nœud privilégié. Tous ces nœuds enfilent la requête du nœud *4* localement. Lors d'une rupture de lien entre *4* et *3*, supposant que le nœud *4* est amené à rerouter sa requête sur un autre chemin. Naturellement le nœud *3* supprime la requête du nœud *4* de sa file mais pas les nœuds *2*, *1* et *0*. Si le nœud *0* envoie le jeton pour le compte du nœud *4*, dans le meilleur cas, il va suivre un chemin inutile jusqu'au nœud *3*! De même, si une rupture de lien se produite entre, par exemple, *3* et *2*, le nœud *4* n'est pas au courant de ce changement; donc, il ne re-route pas sa requête.

3.4. Le protocole de Baldoni et Virgillito

Le protocole proposé par R. Baldoni et A. Virgillito [12] est basé sur un anneau logique unidirectionnel et dynamique et combine entre la circulation et la demande du jeton. Le protocole essaye de réduire la consommation d'énergie en diminuant le nombre de sauts traversés pour exécuter une SC et d'éviter d'échanger des messages de contrôle quand la SC n'est pas invoquée par les noeuds. La mobilité est gérée par la couche de routage du réseau. Le protocole exploite des informations de la table de la couche de routage dans le but de sélectionner le prochain nœud privilégié et utilise les services de celle-ci pour transmettre les messages de contrôle. Les auteurs supposent que les nœuds ne tombent pas en panne et le réseau est fiable. De plus, le réseau n'est pas sujet à des partitionnements permanents. Si un partitionnement se produit, chaque paire de nœuds arrive finalement à communiquer entre elles. Finalement, chaque nœud peut obtenir des informations de la table de la couche de routage.

Le protocole dans son exécution, transite continuellement entre deux état : *idle* et *coordinator-change* et s'exécute en tours. Durant chaque tour, qui est matérialisé par la circulation du jeton à travers tous les nœuds du réseau, un nœud est désigné pour être un coordinateur, soit c_k ; toutes les requêtes lui sont adressées. Un tour est achevé quand tous les nœuds (au nombre de n) du réseau sont visités. La circulation du jeton durant un tour permet

d'accomplir deux missions principales : Satisfaire tous les demandeurs (ayant lancé leurs demandes avant la visite du jeton) et informer tous les noeuds de l'identité du nouveau coordinateur. Quand un nœud désire accéder à sa SC, il envoie une requête au coordinateur et se met en attente du jeton. Le coordinateur insert la requête reçue dans une file ordonnée selon une politique *p* (par exemple, l'ordre d'arrivée). Initialement, le protocole est dans l'état *idle* et le coordinateur c_0 est le nœud p_1 et tous les autres nœuds sont informés de l'identité du coordinateur. Le coordinateur c_k fait transiter le système de l'état *idle* à l'état *coordinator-change* lorsque sa file de requêtes devient non vide. Le coordinateur fait passer le jeton au premier noeud de sa file, soit p_j, qui devient le coordinateur c_{k+1}. Durant le mouvement du jeton le long de l'anneau (de *n-1* nœuds autres que le coordinateur c_k), chaque noeud recevant le jeton entre en SC s'il est demandeur. Quand le jeton doit être renvoyé, le nœud émetteur détermine de manière dynamique et au volet les nœuds non encore visités (à l'aide d'un tableau de booléens transporté par le jeton) et désigne le prochain nœud à visiter s'il en existe et lui passe le jeton. Autrement, le jeton est passé au coordinateur c_{k+1} (qui met son état à *idle*). Le nœud successeur est déterminé selon une politique déterministe *p'* qui se base essentiellement sur la distance (en nombre de pas) entre l'émetteur du jeton est son récepteur. Les nœuds déconnectés sont considérés de distance infinie, mais sont traités par des tentatives de transmission et d'attentes.

Le protocole assure la sûreté puisqu'un nœud ne peut accéder à sa SC que s'il reçoit le jeton et le jeton est représenté par un message unique. La propriété de vivacité est assurée car la structure transportée par le jeton assure que chaque nœud soit visité durant chaque tour.
Trois mesures de performances sous deux niveaux de charges de demandes (faible et haute) sont réalisées : la complexité en messages, le délai de synchronisation (calculé en nombre de message dont le temps moyen de transfert est *T*) et la taille des informations de contrôle transportée avec chaque message de contrôle. La Table 2 montre les résultats théoriques de comparaison entre le protocole proposé et trois autres protocoles basés demande de jeton connus, les protocoles de I. Suzuki et T. Kazami [133], de M. Singhal [124] et de K. Raymond [110] exécutés au dessus d'un protocole de routage.

Protocole	Délai Sync. (forte charge)	Message (forte charge)	Message (faible charge)	Taille d'informations de contrôle
Baldoni-Virgillito	*T*	*2*	*N*	*O(n)*
Suzuki-Kazami	*T*	*N*	*N*	*O(n log n)*
Singhal	*T*	*n/2*	*N*	*O(n log n)*
Raymond	*T(log n)/2*	*4*	*Log n*	*O(1)*

Table 2: *Résultats de comparaison de performances.*

Une comparaison par simulation à l'aide du simulateur GloMoSim [154] est également effectuée, les résultats obtenus concordent avec celles de la Table 2 pour les charges indiquées. Les résultats montrent que les valeurs de toutes les mesures par entrée en SC décroisent avec l'accroissement de la charge selon les résultats théoriques, où la mobilité n'a pas d'impact significatif sur ces mesures.

Discussion

Le protocole décrit dans [12] apporte un certain nombre d'idées intéressantes. La structure d'anneau est établie de manière dynamique et au volé en optimisant à chaque fois le chemin de parcours du jeton vers son prochain destinataire. La mobilité est prise en charge mais de manière indirecte à travers le service de la couche de routage. Cependant, durant un tour du

jeton les nœuds satisfaits sont seulement ceux ayant lancé leurs demandes avant le passage du jeton. Ce qui laisse à conclure que la politique de satisfaction n'est pas tout à fait Fifo. Le calcul du nœud successeur demande beaucoup d'accès à la couche de routage puisque la distance vers tous les nœuds non visités par le jeton doit être connue. Par conséquent, d'une part le protocole de routage doit être proactif et d'autre part, le protocole ne s'adapte pas à la scalabilité du réseau. Les requêtes envoyées par les nœuds servent uniquement à la désignation du prochain coordinateur puisque le jeton visite tous les nœuds. Le jeton visite même les nœuds non demandeurs de SC. Le protocole n'est pas tolérant aux pannes (de nœuds et perte du jeton) ; de plus, il ne traite pas le partitionnement. Il considère aussi que le nombre de nœuds est fixe et connu de tous les nœuds. L'évaluation des performances du protocole ne tient pas compte des messages de routage.

3.5. Protocole de Malpani et *al*

Dans [83], N. Malpani et *al* proposent un protocole paramétré avec plusieurs variantes. Il utilise une structure d'anneau logique, unidirectionnel, dynamique et de taille variable sur lequel un jeton circule. La taille de l'anneau (dans lequel tous les nœuds doivent être visités) peut varier à chaque tour. Toutes les variantes se basent sur un même principe général mais diffèrent sur la manière de sélectionner le nœud privilégié successeur. Le jeton circule continuellement à travers tous les nœuds du réseau mobiles ad hoc au nombre *n* fixe.

L'idée de base de ce protocole réside dans la méthode de choix du prochain nœud pour lequel le jeton sera véhiculé. Les variantes du protocole se divisent en deux classes selon l'origine des informations de décisions, ceux qui se basent sur des informations issues uniquement du voisinage (appelés protocoles *locaux*) et ceux qui se basent sur des informations issues de tous les nœuds du réseau (appelés protocoles *globaux*). Pour éviter la perte potentielle du jeton, le protocole fait recours à la connexion TCP pour délivrer ce jeton. Concernant la mobilité, dans certaines variantes, les nœuds utilisent les messages ''hello'' pour s'assurer que leurs voisins sont toujours présents. Dans chaque variante du protocole, le jeton transporte un certain *compteur* d'information concernant chaque nœud du réseau. Le nœud récepteur, avant d'envoyer le jeton, utilise les informations entretenues pour choisir le prochain nœud auquel le jeton doit être envoyé ; ensuite il met à jour les informations transportées par le jeton. La majorité des variantes choisissent le prochain nœud privilégié, parmi ceux permis, ayant la valeur minimale. Les différentes variantes du protocole sont :

1. La variante *Local-Frequency* (*LF*): Avec cette stratégie, le jeton est envoyé au nœud voisin du nœud détenteur du jeton le *moins fréquemment* visité. A cet effet, le protocole garde des informations sur le nombre de fois que chaque nœud a été visité par le jeton et cette information est transportée par le jeton. Le protocole *LF* assure que chaque nœud soit visité au moins une fois dans le cas de topologie statique. Cependant, dans certaines topologies la longueur du tour peut s'accroître sans limite.

2. La variante *Local-Recency* (*LR*): Dan ce cas, le nœud voisin le *moins récemment* visité est choisi. Afin d'implémenter cette variante, pour chaque nœud, un compteur du temps de la dernière visite du jeton à ce nœud lui est associé. Similairement à la variante *LF*, la famine est absente dans le cas de topologies statiques; de plus, la longueur du tour ne dépasse pas *2*n*.

3. Les variantes globales (*GR* et *GF*): Dans ces variantes, le jeton est envoyé au nœud qui a été visité le *moins récemment* (dans *GR*) et le *moins fréquemment* (dans *GF*) de tous les nœuds du réseau. Dans la variante *GR*, les nœuds sont visités dans le même ordre mais, pas toujours le cas pour la variante *GF*.

4. Les variantes globales avec *Next* (*GRN* et *GFN*): Dans ces variantes, tous les nœuds intermédiaires sur la route choisie vers le destinataire sont visités par le jeton. Pour ce faire, le jeton est envoyé au nœud voisin sur la route du nœud ayant la plus petite valeur du compteur. La couche réseau est alors invoquée pour déterminer le nœud voisin sur la route d'une destination choisie. Les résultats de simulation sont concluants uniquement pour *GRN*.

L'évaluation de performances est réalisée à l'aide du simulateur NS2 [136].Chaque variante est exécutée comme une application au dessus de TCP, du protocole de routage DSR [62] et de la couche MAC IEEE 802.11 [52]. Les mesures de performances considérées sont: La longueur du tour, la complexité en messages qui mesure le nombre d'octets véhiculés tout au long d'un tour (les paquets de contrôle de la couche MAC sont aussi considérés) et finalement, le temps nécessaire pour accomplir un tour. Les évaluations mesures les valeurs moyennes de ces métriques.

Dans les topologies statiques, les meilleurs résultats, concernant le nombre de nœuds visités par tour, sont réalisés par les variantes *GF* et *GR*. Ces résultats concordent bien entendu avec la définition puisque les compteurs concernent seulement les nœuds visités sans considérer les nœuds intermédiaires. En contrepartie, ils sont coûteux en terme de taille d'information transportée et en temps afin d'atteindre une longueur de tour parfaite. La variante *LR* exhibe de bonne performance laquelle un tour converge vers une taille optimale. Aussi, les résultats de simulation montrent que pour chaque variante de protocole, le temps et la taille d'informations transportées montrent les mêmes tendances. La principale différence est que les variantes *GF* et *GR* ne donnent nullement de valeurs optimales. Parmi les variantes globales, la variante *GF* donne de meilleurs résultats. Les performances de la variante *GRN* sont comparables à celles des variantes locales et à celles de la variante *GF*.

Dans le cas de topologies dynamiques, la variante *LR* réalise de bons résultats dans toutes les situations considérées. Cependant, le comportement de la variante *GFN* devient quelque peu imprévisible dans certains cas.

Discussion

Ces variantes de protocoles adoptent un principe général proche du protocole de R. Baldoni et *al* [12]. Dans l'ensemble, les mêmes remarques faites sur ce protocole sont à reprendre pour ces variantes ; sauf que la couche de routage n'est sollicitée que pour l'acheminement de messages. On souligne seulement que la longueur du tour du jeton est en général supérieure à *n*, ce qui cause un retard pour la satisfaction de certains nœuds du réseau. Il est à noter aussi que certaines variantes sont données suite ''à un jeu combinatoire''. Autrement dit, elle n'apporte pas de bonnes performances, en l'occurrence par exemple la variante *GFN*.

3.6. Le protocole de Chen et Welch

Dans [38], Y. Chen et J. L. Welch proposent un protocole auto stabilisant. Il est basé sur le protocole LRV présenté dans [83] (qui présente de bons résultats de simulations), sur le concept d'auto stabilisation définit par E. Dijkstra [42] et sur l'idée de ''counter flushing'' définie dans [142]. Le protocole utilise des anneaux virtuels dynamiques (pour refléter le changement de topologie,) construits par la circulation des jetons. Le protocole requière que la topologie reste statique pendant sa convergence. Après que le protocole converge, sous certaines restrictions sur la mobilité, il garanti les propriété de sûreté et de vivacité. Dans cette thèse, nous ne nous sommes pas concerné par cette approche d'auto stabilisation.

3.7. Le protocole de W. Wu et *al*

Dans [148], W. Wu et *al* ont développé le premier protocole d'EM basé permission spécifique aux réseaux mobiles ad hoc. Ce protocole se base implicitement sur l'utilisation des services de la couche de routage. Donc, les problèmes de mobilité sont gérés par celle-ci. Le protocole exploite une technique nommée de ''recherche à l'avance'' pour essayer de minimiser le coût en messages et de faire participer uniquement les nœuds intéressés par la SC. Cette technique a été initialement utilisée dans [126] pour résoudre le problème de l'exclusion mutuelle dans les réseaux mobiles avec infrastructure. Ce dernier se base sur le protocole de G. Ricar et A. K. Agrawala [117] dans lequel lorsqu'un nœud demande la SC, il envoie sa requête à tous les autres nœuds afin de collecter leurs permissions.

Pour réaliser l'exclusion mutuelle, le protocole dans [148] se base aussi sur l'utilisation de deux structures clés, en l'occurrence les ensembles *Info_set* et *Status_set*. Ces structures entretenues au niveau de chaque noeud *i* représentent respectivement l'ensemble des nœuds auxquels la requête doit être adressée lorsque ce nœud souhaite accéder à sa SC et l'ensemble des nœuds qui, en demandant la SC, doivent envoyer leurs requêtes à *i*. Lors du fonctionnement du protocole, ces deux ensembles sont entretenus afin d'assurer la correction de celui-ci. Le protocole propose aussi des mécanismes pour tolérer le mode veille des nœuds et leurs déconnexions. Des messages spécifiques ont été ajoutés, en l'occurrence le message *Doze*, *Disconnect* et *Reconnect*. A l'aide des temporisateurs et donc de relances de requêtes, la tolérance à la panne de liens et de nœuds est supportée.

D'après les résultats de simulation, ce protocole offre ses meilleures performances sous les conditions de faible charge et de haute mobilité.

4. Recommandations de conception

A travers la présentation des différentes solutions des différents environnements et les nouvelles caractéristiques introduites par les réseaux mobiles ad hoc, il ressort un certain nombre de suggestions à prendre en considération dans l'élaboration de toute solution d'EM pour les réseaux mobiles ad hoc.

- Il semble que l'approche à jeton est mieux adaptée aux réseaux mobiles ad hoc pour plusieurs raisons. Premièrement, il est plus facile d'entretenir un jeton plutôt que des permissions. En effet, quelque soit l'organisation de l'ensemble des noeuds du réseau, il n'est pas évident de contacter les noeuds pour obtenir la permission ou pour entretenir son appartenance à une structure éventuelle définie vu les nouvelles caractéristiques des réseaux mobiles ad hoc (notamment, la mobilité et les déconnexions). En échange, il est relativement plus facile de gérer un jeton dans ce type de réseau. Deuxièmement, le nombre de messages engendré par l'utilisation de jeton est dans l'ensemble moindre comparativement à l'approche à permission. Néanmoins, l'approche à jeton est sensible à la perte de jeton. Des mécanismes de régénération du jeton et d'évitement de sa duplication s'avèrent nécessaires. En supposant que l'approche à jeton est adoptée, d'autres recommandations sont à souligner:
- Etant donnée le dynamisme du réseau, particulièrement en terme de connexions et déconnexions de noeuds, il n'est pas raisonnable de fixer le nombre de noeuds de ce réseau.
- Il est déconseiller de faire participer les noeuds non intéressés par la SC au service d'EM.

- Afin d'éviter d'encombrer le service, l'utilisation au mieux des messages de services pour véhiculer l'information de contrôle est indispensable. Autrement dit, il est utile d'éviter l'utilisation des messages spécifiques pour l'échange d'informations de contrôle.
- Créer une certaine redondance dans l'entretient de l'information de contrôle pour assurer la disponibilité du service. En contrepartie, il faut produire des mécanismes permettant de purger l'information devenue non utile afin d'assurer la cohérence du service et d'optimiser l'espace mémoire utilisé par le service.

5. Conclusion

A travers ce chapitre, nous avons montré la progression des solutions au problème d'EM partant des systèmes distribués en passant par les environnements mobiles et en finissant par réseaux mobiles ad hoc. Nous avons commencé par présenter les solutions distribuées classiques d'EM et de base connues dans les environnements de calcul distribué statique selon une méthodologie de classification qui permet de regrouper les solutions utilisant des outils et mécanismes similaires. Nous avons alors distinguer les solutions qui se basent sur l'approche à permission et celles qui se basent sur l'approche à jeton. Dans la première approche, deux types de solutions sont présentés. En premier, celles qui utilisent la permission individuelles [28, 37, 46, 76, 137]; en second, les solutions qui utilisent la permission d'arbitre [80, 120] qui sont en général les solutions qui utilisent un concept mathématique connu, les quorums. Dans ce cadre, la solution typique de Maekawa a été exposée. Dans la seconde approche, plusieurs groupes de solutions [36, 40, 50, 77, 88, 93, 110, 118, 124, 133] ont été présentés. Par la suite, un tableau récapitulatif et comparatif de toutes ces solutions a été dressé.

Après les solutions de environnements statiques, nous nous sommes focalisé sur celles développées pour les réseaux mobiles ad hoc. Ces solutions sont en bonne partie des adaptations de celles connues dans les systèmes distribués. Au préalable et afin de montrer l'évolution des solutions vers les réseaux mobiles (ou cellulaires) qui sont des précurseurs des réseaux mobiles ad hoc, deux solutions adaptées pour cet environnement ont été présentées. Ensuite, deux premières solutions [149, 150] dont l'une [150] est l'amélioration de l'autre qui semblent être convenables pour l'environnement mobile ad hoc, sont présentées. Ces deux solutions utilisent la même structure logique dont les liens logiques sont bâtis directement sur les liens physiques. Les deux autres solutions présentées par la suite [12, 83] présentent des caractéristiques générales semblables, en particulier la méthode de circulation du jeton et l'utilisation des services de la couche de routage. Toutes ces quatre solutions utilisent l'approche à jeton qui semble être une tendance pour ces réseaux. Néanmoins, une solution à permission [148] existe et fait l'objet de la description dans la suite du chapitre. Chacune des solutions présentées est succédée d'une discussion qui suggère quelques critiques et insuffisances de ces solutions. Dans l'ensemble, nous avons relevé les remarques suivantes: toutes ces solutions ne sont pas tolérantes aux fautes (i.e. perte de jeton et panne de nœuds), ne considèrent pas le partitionnement, ne sont pas scalables et certaines qui font recours à la couche de routage dépendent de ses performances. D'autre part, certaines solutions introduisent des restrictions sur le mouvement des nœuds formant le réseau.

Les solutions actuelles au problème d'EM dans les réseaux mobiles ad hoc représentent une étape vers la résolution de ce problème dans un environnement en plein essor. Dans le Chapitre III, nous somme concernés par la description d'une nouvelle solution d'EM, prévue pour les réseaux mobiles ad hoc, qui essaye de remédier aux insuffisances des solutions existantes dans la littérature et de tenir compte des recommandations qui ressortent.

Chapitre III

Un protocole d'exclusion mutuelle basé jeton pour les réseaux mobiles ad hoc

1. Introduction

Le problème d' EM a été largement traité dans les réseaux statiques distribués. Ainsi, plusieurs solutions ont été proposées [125], selon deux approches essentielles: L'approche à *permission* et l'approche à *jeton*. Dans l'approche à permission, pour entrer à sa SC, un nœud demandeur doit attendre de recevoir les réponses de tous les autres nœuds du réseau [76, 117], (ou d'un sous-ensemble dans certains algorithmes [28, 80]). Dans l'approche à jeton, un message spécial (i.e. jeton) transporte le privilège d'accès à la SC. Son unicité dans le réseau garanti l'EM. Deux méthodes sont utilisées dans cette approche. Dans la première méthode, le jeton circule perpétuellement dans le réseau, en suivant une structure prédéfinie (tel qu'un anneau [77]), ou sur des chemins définis dynamiquement selon des critères prédéfinis. Par conséquent, un nœud désirant accéder à sa SC se contente d'attendre son '*tour'*, i.e. la réception du jeton. Dans la seconde méthode, un nœud demandant la SC est amené à demander le jeton; le problème de base est comment l'atteindre? Dans certains algorithmes, la requête est adressée à tous les nœuds du réseau [36, 41, 118, 133]. Dans d'autres, une structure logique est définie pour désigner le détenteur du jeton [93, 110]. Quelques-unes des solutions de l'approche à jeton ont été adaptées pour fonctionner dans les réseaux cellulaires ainsi que dans les réseaux mobiles ad hoc.

Les protocoles initialement prévus pour les réseaux fixes ne peuvent pas s'appliquer aux environnements mobiles [18], mais ces derniers en sont en général une adaptation. Dans [8], B. R. Badrinath et *al* ont proposé deux algorithmes distribués d'EM pour les réseaux cellulaires. Le premier est une adaptation pour les réseaux cellulaires de l'algorithme proposé

par L. Lamport [76], le second est une adaptation du protocole proposé par Le Lann [77]. Dans [150], J. Walter, S. Kini et N. Vaidya ont proposé un protocole distribué d'EM pour les réseaux mobiles ad hoc orienté jeton dérivé de plusieurs autres algorithmes : Le protocole de routage de E. Gafni et D. Bertsekas [44] basé sur un arbre, le protocole présenté dans [36], et d'autres idées de [41] et [110]. Ce protocole défini une structure bâtie sur la topologie réelle du réseau qui est représentée par un DAG de pointeurs orientés jeton, maintenant plusieurs chemins menant vers le détenteur du jeton. Ce protocole s'adapte bien à l'environnement mobile ad hoc puisqu'il requiert aux nœuds de maintenir uniquement des informations sur les voisins immédiats. Quand un noeud est amené à véhiculer le jeton au prochain, il trouve nécessairement un chemin de retour puisque le partitionnement n'est pas permis. Cela suppose aussi que le chemin vers le détenteur du jeton est toujours entretenu à l'avance. Dans [12], R. Baldoni, A. Virgillito et R. Petrassi ont proposé un protocole utilisant un anneau logique dynamique et combinant la méthode de recherche et celle de circulation de jeton. Le protocole essaye de réduire la consommation d'énergie par la réduction du nombre de pas traversés pour l'accès à une SC et évite d'envoyer des messages de contrôle dans le cas d'absence de requêtes. La mobilité est supportée à l'aide de l'utilisation de la couche de routage, cela permet de sélectionner le nœud le plus proche pour l'envoi du jeton. Dans [83], N. Malpani, N. H. Vaidya et J. L. Welch ont proposé un algorithme orienté jeton, paramétrable et avec plusieurs variantes. Le jeton circule *continuellement* sur un anneau logique dynamique de taille variable. Toutes les variantes ont le même squelette mais diffèrent sur la politique de *sélection* du prochain nœud à visiter.

Tous ces protocoles ne prennent pas en compte certaines caractéristiques des réseaux mobiles ad hoc, par exemple les pannes et le partitionnement (voir Chapitre II Section 3 pour les critiques des différentes solutions présentées). Dans ce chapitre, nous proposons un nouveau protocole d'EM pour les réseaux mobiles ad hoc [20] basé sur l'approche à *jeton* (qui est d'ailleurs la tendance actuelle). Le protocole décrit dans ce chapitre ne repose pas sur la couche de routage, n'utilise pas de structure logique de communication et qui, pour la circulation du jeton, privilégie la satisfaction des demandes en fonction d'une part de leur ancienneté et d'autre part de leur distance par rapport au jeton. Par ailleurs, le protocole proposé prend en charge de nombreuses caractéristiques des réseaux mobiles ad hoc telles que : le mouvement des nœuds (i.e. les cassures et formations de liens), le partitionnement et la fusion, les pannes de nœuds et la perte de jeton. Le protocole favorise la scalabilité du réseau et considère un nombre indéterminé de nœuds. Par ailleurs, il permet de créer un jeton au niveau de chaque partition créée et au contraire, de ne garder qu'un seul dans le cas de fusion de partitions.

Ce chapitre est organisé comme suit : Après un bref aperçu sur quelques travaux antérieurs dans la Section 1, dans la Section 2 nous présentons les idées de base du protocole et les structures de données et de communication utilisées. Ensuite dans la Section 3, une description du fonctionnement du protocole est donnée. La Section 4 finalise la description du fonctionnement du protocole en montrant son comportement vis à vis du problème de partitionnement. Un simple exemple d'exécution suivi d'une discussion sur certains aspects cachés du protocole sont donnés dans la Section 5. La Section 6 propose une étude de correction du protocole. La Section 7 traite les différents résultats de simulation obtenus ainsi que leurs interprétations. Finalement, la Section 8 conclue le chapitre.

2. Eléments généraux du protocole

2.1. Modèle du réseau

Nous supposons un réseau composé d'un nombre variable de nœuds mobiles, chaque nœud possède un identificateur unique. Les mouvements des nœuds du réseau sont libres (i.e. mobilité aléatoire). Les nœuds utilisent un support de communication sans fil, les liens de communication sont bidirectionnels, FIFO et fiables. Un nœud peut à tout moment tomber en panne ou quitter le réseau. L'intégration d'un *nouveau* nœud est possible, cependant le *retour après panne* d'un nœud fait l'objet d'une discussion dans la Section 5.2. Les mouvements des nœuds peuvent engendrer des créations et / ou des pertes de liens, le partitionnement et la fusion. La couche de liaison permet à tout moment aux nœuds de connaître leurs voisins. Un nœud ne peut pas lancer de nouvelle requête que s'il n'a pas de requête en cours. Par ailleurs, la durée d'une SC est supposée bornée. Dans la suite du chapitre et afin de décrire notre protocole, nous nous plaçons au niveau du nœud désigné par l'indice *i*.

2.2. Idées de base du protocole

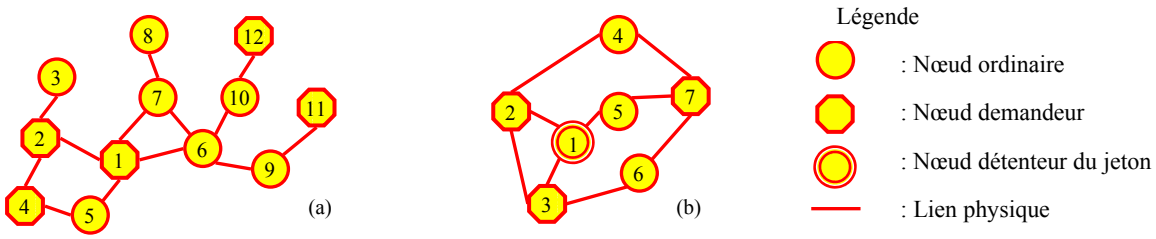
Le protocole utilise l'approche à jeton dont l'unicité dans une *partition* assure d'une manière intrinsèque l'EM dans celle-ci. Il combine la circulation et la demande de ce dernier. Le jeton est identifié de manière unique dans chaque partition et transporte entre autres une file de requêtes récoltées lors de son parcours. Le protocole se base sur la connaissance locale, celle du voisinage immédiat et ne prend en considération que les liens physiques pour acheminer de l'information. Ce qui lui permet de favoriser la *scalabilité* du réseau et de considérer un nombre *indéterminé* de nœuds. D'autre part, le protocole fait recours à un certain nombre de mécanismes qui sont décrits par la suite.

2.2.1. Rayon de diffusion

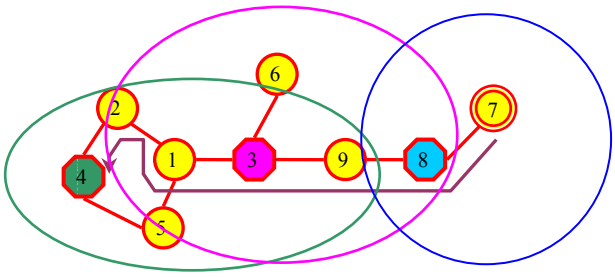
Puisque la détention du jeton conditionne l'accès à la SC, la diffusion d'une requête est l'un des moyens permettant de l'acquérir. Afin de minimiser l'impact de la diffusion, nous introduisons le concept du rayon de diffusion. Ce rayon est calculé de manière dynamique à partir des requêtes déjà reçues par *i* (dans une file locale notée Q_i). Plus précisément, seul le nombre de sauts des requêtes *valides* séparant leurs émetteurs de *i* sert à calculer ce rayon. La validité d'une requête est liée à celle de sa route qui l'est pendant un délai fixe (i.e. *ValidTimer*) du moment de sa réception. En supposant que les requêtes (avec routes valides, seules considérées) reçues localement sont triées dans un tableau temporaire T_i d'enregistrements selon l'ordre croissant de leurs longueurs de routes, le rayon est calculé comme suit *:* $R_i = /T_i\ [pos].route/$ *avec* $pos = Round\ (P^* /T_i /)$, où P est une constante représentant la proportion de nœuds à atteindre et la notation /*x*/ désigne la longueur de *x*. Les nœuds se trouvant dans la zone délimitée par ce rayon vont recevoir cette demande. Dans la Figure 9(a), admettons que le nœud *1* désire émettre une requête, il doit donc calculer son rayon de diffusion R_1. Pour se faire, supposons que sa file Q_1 contient les demandes *valides* des nœuds *2, 4, 11* et *12* avec leurs routes respectives *2 ; 5, 4 ; 6, 9, 11 ; 6, 10, 12*. Avec $R_1 = 3$, il atteindra tous ces nœuds ou avec $R_1 = 2$, il atteindra *50 %* de ces nœuds. Cette méthode de calcul du rayon favorise la création d'une cascade de chemins vers le détenteur du jeton.

Dans la Figure 9(c), la requête du nœud *4* avec la route *1*, *2*, *4* est enregistrée au niveau du nœud *3*, la requête du nœud *3* avec la route *9*, *3* est enregistrée au niveau de *8* et finalement la requête du nœud *8* est enregistrée au niveau de *7*, nœud détenteur du jeton. Donc, celui-ci est au courant de la requête du nœud *8*. Donc, les nœuds peuvent être satisfaits selon l'ordre *8*, *3* et *4*. De manière générale, l'ordre de satisfaction des nœuds n'est pas forcément l'ordre cité.

De ce fait, il faut transporter les requêtes rencontrées par le jeton au niveau des nœuds visités. Puisque le jeton récolte les requêtes rencontrées sur son chemin, tous ces noeuds seront éventuellement satisfaits, en dépendant de la mobilité et des pannes de nœuds.



(a) *Calcul du rayon de diffusion.* (b) *Situation de famine.*



(c): *Cascade de chemins vers le jeton.*

Figure 9: *Graphes pour diverses situations.*

2.2.2. Relances de requêtes

Un nœud demandeur de SC peut ne pas recevoir le jeton et ceci pour plusieurs raisons: Changement fréquent du voisinage de ce nœud, changements fréquents des liens dans le réseau dus à la mobilité, panne de nœuds, perte de jeton, etc… Dans le but d'éviter la famine à ce nœud, la notion de relance de requête est introduite: Un nœud demandeur de SC non satisfait au bout d'un délai défini par un temporisateur (i.e. *RequestTimer*) est amené à relancer sa requête. Cette relance est considérée comme une nouvelle tentative d'une même requête; autrement dit, elle est générée avec la même estampille afin de lui assurer son ancienneté dans le système. Le nombre de relances ne doit pas être illimité. En effet, au-delà d'un certain seuil, un mécanisme lié au soupçon de perte du jeton doit être entrepris (voir Section 4.1).

2.2.3. Sélection du nœud privilégié

Chaque nœud recevant une demande aura connaissance de l'identité de son émetteur, ainsi que de la route parcourue et l'inscrit dans sa file d'attente. Cette file peut être organisée suivant une politique qui privilégie les requêtes les plus proches. Cette politique induit des cas de famine. En effet, dans le cas d'un groupe de nœuds proches les uns des autres qui demandent l'entrée en SC fréquemment, le jeton ne cessera pas de circuler entre eux, ce qui prive les demandeurs les plus éloignés. Dans la Figure 9(b), admettons que les nœuds *2*, *3* et *7* sont demandeurs de SC, que le nœud *1* est en SC et que les nœuds *1*, *2* et *3* demandent continuellement le jeton. En considérant uniquement le nombre de sauts, le jeton sera monopolisé par les nœuds *1*, *2* et *3* et ainsi le nœud *7* sera privé de sa SC. Afin d'éviter ce problème, l'ancienneté des requêtes est aussi considérée [76]. D'où, la priorité d'une requête d'un nœud *j*, au niveau du nœud privilégié *i* est donnée par $Pr_j = (c1*NS_j + c2*\ nb\_hops,\ j)$,

où *c1 et c2* sont deux constantes dont les valeurs définissent l'importance de tel ou tel paramètre, *nb_hops* est le nombre de sauts séparant *i* de *j*, NS_j est l'estampille de la requête de *j* et *j* permet de créer un ordre total entre les requêtes. Comme la distance qui sépare deux nœuds est bornée, et les estampilles des nœuds servis augmenteront rapidement, la priorité tournera forcément en faveur des nœuds les plus éloignés.

2.2.4. Verrouillage du jeton

Quand un nœud est sélectionné comme prochain à être servi, le jeton lui est véhiculé sur le chemin enregistré avec sa requête. Des nœuds intermédiaires demandeurs de SC et se trouvant sur la route du jeton vers le destinataire peuvent être servis, puisque le jeton leur est parvenu, mais sans influencer la trajectoire initiale du jeton. Cette possibilité permet d'éviter au jeton de faire éventuellement plusieurs sauts avant d'atteindre son destinataire, et par conséquence améliorer le temps d'attente d'un nœud demandeur. Néanmoins, la mobilité des nœuds peut causer des cassures de routes ou leurs rallonges. Par conséquent, le fait que le jeton est ralenti dans sa route peut créer des cas de famine. Pour remédier à ce problème, la notion de *verrouillage* du jeton est introduite. Puisqu'un nœud demandeur de SC non satisfait au bout d'un temps fixé est amené à relancer sa demande un certain nombre de fois, un seuil dit de *verrouillage* (*Lock_limit*) est défini. Arrivant à ce seuil, le jeton est verrouillé pour sa destination, ce qui implique que les nœuds intermédiaires recevant ce jeton ne peuvent pas entrer en SC.

2.2.5. Recherche de route

Pendant la circulation du jeton vers une destination donnée, la route peut être perdue; d'où l'association d'un état au jeton, *ONRoute* s'il suit normalement une route prédéfinie, *NORoute* dans le cas de perte de route et un nouveau destinataire n'est pas encore trouvé. Un nœud *i* qui reçoit le jeton, s'il est demandeur et le jeton lui est destiné ou le jeton est non verrouillé ou avec un état *NORoute*, entre en SC, sinon son rôle se réduit à router le jeton suivant la route inscrite sur ce dernier. Dans le cas où cette route serait brisée, le nœud *i* tente de la recalculer d'une manière locale en cherchant la présence des nœuds suivants le saut brisé dans son voisinage. Si cette opération échoue, le jeton change de destinataire qui sera choisi selon la méthode de *sélection du nœud privilégié.* En cas d'échec, le jeton est envoyé (avec l'état *NORoute*) à l'un des voisins pour la recherche d'un destinataire. La conséquence en est éventuellement la non satisfaction de certains nœuds demandeurs. Ce problème peut être surmonté par la possibilité de redemander la SC après un délai fixé.

La Figure 10 montre un scénario de recherche de route du jeton. Dans la Figure 10(a), supposons que le prochain nœud destinataire du jeton sélectionné par le nœud privilégié *1* est *5*. Le jeton est transmis au nœud *2* pour suivre la route *1*, *2*, *3*, *4*, *5* vers son destinataire avec un état *ONRoute*. Arrivant au nœud *2* (Figure 10(b)), un changement de liens a eu lieu dont la conséquence est la perte du prochain nœud (i.e. le nœud *3*) sur le chemin restant du jeton. La route est récupérée sans inconvénient, puisque le nœud *4* qui suit le nœud *3* (i.e. le saut brisé) sur le chemin restant du jeton est voisin du nœud *2*; le jeton retrouve ainsi son chemin vers le nœud *5* en gardant l'état *ONRoute*. En échange dans la Figure 10(c) (où des changements de liens ont eu lieu partant de la Figure 10(a)), le jeton arrive sans difficulté jusqu'au nœud *3*. Par la suite, il y a une perte totale de la route du jeton vers son destinataire. Un autre destinataire est alors recherché ; si on admet que la requête du nœud *6* est reçue au niveau du nœud *3*, il sera alors sélectionné par celui-ci comme prochain destinataire et le jeton lui sera transmis avec un état *ONRoute*. Le même problème que celui de la Figure 10(c) se pose au niveau de la Figure 10(d). Dans ce cas, le jeton rebrousse chemin pour la recherche du nouveau destinataire mais cette fois-ci avec un état *NORoute*.

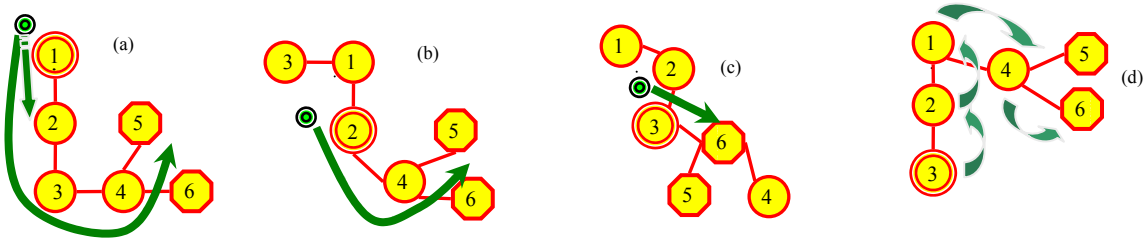


Figure 10: *Récupération de route du jeton.*

2.2.6. Recherche et régénération du jeton

Le soupçon de perte de jeton est déclenché quand un nœud ne le reçoit pas après un nombre limité de tentatives de demandes (i.e. *Req_Threshold*). Au-delà, une recherche dans toute la partition courante s'impose. Pour des raisons d'optimisation, le message de recherche diffusé est à triple objectif: Demande de jeton, recherche de jeton et proposition de création de jeton en cas de perte. Ainsi, tous les nœuds informés de cette recherche en cours se *bloquent* temporairement du point de vue génération de demandes de SC et véhicule du jeton, s'il y a lieu. Sachant que plusieurs nœuds peuvent être en même temps en situation de recherche du jeton, en cas de perte de ce jeton (ou sa non présence dans la partition courante), un conflit se produit lors de sa création. De ce fait, on utilise un mécanisme déterministe de sélection d'extremum connu dans les protocoles d'élection [97, 144]: Par exemple, ''seul le nœud ayant l'identité la plus petite'' peut le créer.

2.2.7. Désignation du jeton

A cause des problèmes de partitionnement, plusieurs jetons peuvent être créés, un pour chaque partition créée. Au moment de la fusion de partitions, il est impératif de ne garder qu'un seul jeton dans la partition ainsi créée. Pour faciliter la destruction d'un jeton, il suffit de l'identifier. L'identité retenue pour un jeton est celle du nœud qui l'a créé. Cependant un nœud, ayant créé un jeton dans une partition donnée, peut avoir la chance de créer un autre s'il se retrouve dans une autre partition. Afin d'éviter l'ambiguïté de désignation des jetons, à chaque nouveau jeton est associé un numéro de séquence local au processus qui l'a créé.

2.3. Structures

2.3.1. Structures de données de bases locales à un nœud *i*

Chaque nœud *i* est doté d'un certain nombre de variables et de structures définissant son contexte. Ces variables et structures seront données comme suit:

- $State_i$: Indique l'état de *i*, par rapport à la SC, qui peut être *idle*, *req*, *inCS,* ou. Initialement, $State_i=idle$.
- NS_i : Estampille du nœud *i*, elle permet de dater ses requêtes. Initialement, $NS_i=0$.
- $Request\_NS_i$: Estampille de la dernière requête, sert à garder la même date (i.e. même priorité) pour plusieurs tentatives de la même requête afin d'éviter la famine.
- $Holder_i$: Indique si le nœud *i* est détenteur du jeton ou non, initialement pour *i=0* $Holder_i = True$, autrement $Holder_i = False$.
- $Last\_Holder_i$: Désigne le dernier nœud détenteur du jeton.
- N_i : L'ensemble des voisins de *i.*
- $Nb\_attempts_i$: Nombre de tentatives d'une même demande de CS
- $Ignored\_Neighbors_i$: Ensemble de voisins ignorés par *i*, lors de la phase du traitement des cassures de routes, afin d'éviter un problème de boucle.

- Q_i : File d'attente de *i*, au format *(j, $Route_j$, NS_j, $Valid_j$, $Nb\_attemps_j$)*. Ce format représente la demande d'un nœud *j*, où $Route_j$ est une séquence de nœuds intermédiaires entre *i* et *j*, NS_j est le numéro de séquence de la demande de *j*, $Valid_j$ représente la validité de la requête de *j* (sert uniquement dans le calcul du rayon de diffusion de *i*) et $Nb\_attemps_j$ est le nombre de tentatives associées à cette requête.
- D'autres structures liées à la régénération du jeton son aussi entretenues.
- *Token _ID_i (*respectivement *Token _ID_NS_i)*: Indique l'identité (respectivement l'estampille) du jeton. Initialement *Token _ID_i =0* (respectivement *Token _ID_NS_i=0).*
- *Token_$CreatedNS_i$* : Indique un nombre séquentiel local à *i* utilisé lors de la création d'un jeton. Initialement *Token_$CreatedNS_i$=0.*
- *Received_NS_i* : Liste contenant les derniers NS des demandes des autres nœuds déjà reçus par *i.*
- *SearchingToken$_i$* : Mise à *True* si le nœud *i* cherche l'existence du jeton dans sa partition, sert à stopper l'émission de messages de demande d'entrée en SC et ceux de la circulation du jeton
- *Block_State$_i$* : *(Idle, ToAsk_For_CS, RequestTimer(i), ToSendToken)* : Donne le motif de blocage du nœud lors de la recherche du jeton. Elle sert à préciser l'action à entreprendre après le déblocage.

2.3.2. Types de communications entre noeuds

Le protocole utilise différents types de communications pour réaliser le service de l'exclusion mutuelle.

- Une demande d'entrée en SC est exprimée par le message *Request (j, NS_j, $Route_j$, R_j, $Nb\_attempts_j$, $aware_j$)*. Ce message est accompagné respectivement de l'identité du nœud source, de l'estampille du message, de la route suivie qui se construit au fur et à mesure que le message se propage, du rayon de diffusion prévu, du nombre de tentatives de cette requête et d'une connaissance sur les nœuds visités par ce message (permettant de minimiser l'impact de la diffusion).
- Le privilège d'entrée en SC est matérialisé par le message *Token (Token_id, Token_idNS, Token_Last_Holder, Token_State, Token_Locked, Token_Route, Token_Req_Set, Token_NS,)*. Ce message est accompagné de son identification, de l'identification de son ancien détenteur, des informations sur son état (i.e. suit un chemin prévu ou a perdu son chemin, verrouillé pour sa destination ou non), de la route à suivre ainsi que des informations sur les requêtes connues de ce jeton.
- La panne de nœuds, le partitionnement et les changements fréquents de liens entre nœuds peuvent engendrer une perte ou un isolement temporaire du jeton de la partition considérée. Le soupçon de perte du jeton dans la partition en cours apparaît dès lors qu'un nœud demande plusieurs fois le jeton mais sans le recevoir. Le message *IsThereToken (j, $Request\_NS_j$, $Route_j$, $Nb\_attempts_j$, $Token\_ID\_NS_j$, $aware_j$)* permet donc la recherche d'un jeton. Ce message transporte des informations sur la requête qui est à l'origine de cette recherche, une proposition d'identification du jeton à créer en cas de perte et une connaissance des nœuds visités par ce message.
- Dans le cas où le jeton en cours n'est pas perdu, son détenteur utilise le message *ThereIsToken ($aware_j$)* comme confirmation de son existence. Ce message transporte uniquement une connaissance des nœuds visités par ce message.
- Les fusions de partitions sont aussi une autre conséquence des mouvements libres des nœuds; des problèmes de duplication de jetons peuvent donc survenir étant donné que chaque partition est dotée de son propre jeton. Comme la fusion se produit à l'occasion de créations de liens entre nœuds, l'échange d'informations entre les nœuds concernés

devient vital. Pour cela, le message *LinkInfo (m, Token_ID$_j$, Token_ID_NS$_j$, SearchingToken$_j$, CandidateTokenID_NS$_j$, ns$_j$, route$_j$, NB_attempts$_j$, aware$_j$)* est utilisé. Au niveau d'un nœud émetteur, ce message transporte l'identification du jeton de sa partition et éventuellement les arguments d'un message *IsThereToken ()* reçu ou émis par ce nœud (voir Section 5.2).

- Enfin, dans le cas de détection de fusion, le message *DeleteToken (TokenID$_j$, TokenID_NS$_j$, aware$_j$)* est utilisé pour supprimer l'un des deux jetons. Afin d'informer les nœuds de la partition ciblée de l'identification du seul jeton retenu, l'identification de celui-ci accompagne ce message. Une information sur les nœuds visités par ce message est également transportée.

3. Description du fonctionnement du protocole

Chaque nœud du réseau exécute un algorithme symétrique. Son texte est formé d'un ensemble de primitives liées aux différents événements le constituant. Au niveau de chaque nœud, les primitives s'exécutent de manière atomique, cependant les événements qui se produisent durant l'exécution de la SC peuvent être traités.

3.1. Propagation d'une requête

Un nœud *i* désirant accéder à sa SC (*State$_i$:=Req*), s'il détient le jeton (*Holder$_i$ =true*), il y accède directement. Autrement, il est amené à diffuser sa demande *Request (),* estampillée selon les horloges de L. Lamport [76], dans son voisinage selon un rayon calculé comme décrit dans la Section 2.2.1. Tous les nœuds intermédiaires recevant cette requête se chargent de la véhiculer selon le rayon indiqué. Dans le cas particulier où le nœud source de la requête aurait une file vide ou toutes les requêtes y figurant sont invalides, c'est à ces nœuds intermédiaires (en commençant par ses voisins) de lui calculer un rayon en suivant le même procédé. Un délai de garde *RequestTimer* est associé à chaque requête émise. Au bout de ce délai, si le jeton ne parvient pas à ce nœud demandeur, la requête est relancée mais avec la *même* estampille afin de lui assurer son ancienneté. Dans le cas échéant, ce processus est exécuté un nombre de fois limité (i.e. *Req_Threshold*) avant de soupçonner la perte du jeton (voir Section 4.1).

Quand un noeud *i* est amené à envoyer une requête mais il est bloqué (i.e. *SearchingToken$_i$ =true*), l'envoie de la requête est reporté au moment du déblocage (*Block_state$_i$* est alors initialisé à *ToAsk_For_CS*).

3.2. Conditions d'accès à la SC

Un nœud demandeur de SC accède à sa SC dans les cas suivants:

- il détient le jeton à l'état de repos (*(State$_i$ =req)* and *(Holder$_i$ =true)*),
- il reçoit le jeton (*Holder$_i$ =true*) et de plus, soit qu'il est destinataire (i.e. *(Last(TRoute)=i)*), où *Last* est une fonction qui retourne le dernier élément de la route donnée en paramètre),, soit que le jeton est non verrouillé (i.e. *Token_Locked=false*), soit que le jeton est sans destination fixe (i.e. *Token_Locked= NORoute*).

3.3. Déplacement du jeton

Un nœud *i* détenant le jeton (i.e. *Holder$_i$=true*) et n'étant pas dans une situation de blocage (i.e.*SearchingToken$_i$ =false*) (sera vu dans la Section 4.1), se charge de le véhiculer, avec l'état *ONRoute*:

- Cas a sur la route d'un nouveau destinataire choisi selon la politique de *sélection du nœud privilégié,* si l'un des cas suivants se présente :

o le jeton est à l'état de repos et le nœud *i* vient de recevoir une nouvelle requête,
o le nœud *i* vient de sortir de sa SC et sa file d'attente contient au moins une requête,
o le nœud *i* était bloqué *et* était entrain d'attendre l'envoi du jeton à un prochain nœud demandeur (i.e. *(SearchingToken$_i$ =true) and (Block_state$_i$= ToSend_Token)*), mais vient de se débloquer suite à l'expiration du temporisateur *StabilisationTimer* (voir Section 4.1).
o le jeton vient d'être reçu avec un état *NORoute*, le nœud *i* n'est pas demandeur de SC et sa file locale n'est pas vide.

Pour ce cas (a), le jeton sera de plus verrouillé si le nombre de tentatives de la requête du nouveau destinataire a dépassé le seuil de verrouillage (i.e. *Lock_limit*).

- Cas b: sur la route d'un destinataire prédéfini mais après une éventuelle opération de récupération de cette route (dans le cas de sa perte suite au déplacement de nœuds prévus sur celle-ci) dans le cas suivant:

o le jeton vient d'être reçu avec une route prédéfinie et le noeud *i* n'est pas le destinataire.

Dans le cas (b), si la récupération de route est impossible, un nouveau destinataire s'il y a lieu est choisi sinon, le jeton sera affecté de l'état *NORoute* et sera passé à un voisin qui est sélectionné, afin de rechercher un nouveau destinataire en évitant les boucles (voir l'exemple de la Section 5.2.). Cette dernière opération est aussi effectuée dans le cas (a), si la file n'est pas vide et les chemins vers tous les nœuds de celle-ci sont non récupérables.

En dehors des cas cités ci-dessus, le jeton est gardé localement. Il est à noter que pour tous les cas indiqués ci-dessus, si le nœud *i* est bloqué à cause d'une recherche en cours du jeton, l'envoi du jeton est reporté au moment de son déblocage (*Block_state$_i$* est alors initialisé à *ToSend_Token*) (voir Section 4.1).

3.4. Mise à jour des files d'attente

La mise à jour des deux files entretenues (i.e. la file d'attente Q_i locale à chaque nœud *i* et celle transportée par le jeton) se fait lors de la réception du jeton, d'une nouvelle requête ou d'un message *IsThereToken ()* ou à la sortie d'une SC. Globalement, les mises à jour de ces files permettent de purger les requêtes déjà satisfaites ou anciennes et de se rendre compte de celles qui sont nouvelles ou présentant de meilleures routes.

A cet effet, le jeton est amené à transporter entre autres des informations sur les requêtes satisfaites. Il est clair que cette solution engendre l'augmentation de la taille du jeton. Une solution d'échange pourrait être de supprimer chaque requête selon un critère prédéfini (par exemple, quand elle atteint un certain délai de séjour dans la file). Cette dernière solution est viable puisque des mécanismes (tels que les relances, satisfaction des nœuds intermédiaires,..) permettent de remédier à ses effets secondaires éventuels.

4. Le partitionnement et la fusion

Chaque partition est connue par une identité (i.e. *Tid,* celle du jeton qui circule en elle,) afin de permettre la détection de fusions de partitions et donc de ne garder qu'un seul jeton. L'estampille du jeton (*i.e. TidNS*) est aussi utilisée afin de distinguer les partitions dont les jetons ont été créés par le même nœud. Ces informations sont connues de tous les nœuds de la partition.

4.1. Le partitionnement

Dans le cas où le nombre de tentatives d'un nœud *i* demandeur de jeton atteint sa limite (i.e. $Nb\_attempts_i = Req\_Threshold$), il commence à soupçonner la non présence de ce dernier (i.e. partitionnement ou perte du jeton), et change la manière de demander la SC. En effet, aucun rayon de propagation n'est calculé, sa demande va être diffusée sur tout le réseau. De plus cette dernière n'est plus véhiculée par un message *Request ()*, mais par un message *IsThereToken ()*. Le nœud *i* se *bloque* ensuite (i.e. met $Search\_Token_i$ à *true*, ce qui le même à ne plus générer de demandes) en attente de la réception d'un message de retour *ThereIsToken ()* pendant une durée de temps limitée (i.e. *SuspicionTimer),* au bout de laquelle le traitement du partitionnement est appliqué. Si le message *IsThereToken()* rencontre le jeton au niveau d'un nœud donné, celui-ci est amené à diffuser dans tout le réseau le message *ThereIsToken()* mais après un délai de stabilisation *StabilisationTimer* pour éviter le blocage de certains nœuds, après leur déblocage, par des messages *IsThereToken()* non *éteints*. Si avant la fin du délai *SuspicionTimer*, le message *ThereIsToken ()* est reçu, le nœud *i* se *débloque* (met $Search\_Token_i$ à *false* et exécute l'action éventuelle en attente enregistrée dans $Block\_state_i$) et continue la propagation de ce message. Par contre, si aucune réponse *ThereIsToken ()* ne lui est parvenue avant l'expiration de ce délai, alors puisque sa demande sous-entend aussi une proposition de création d'un nouveau jeton, le nœud *i*, s'il est seul à avoir demandé la création d'un jeton, le crée et le protocole redémarre pour cette partition. En cas de propositions simultanées de création de jetons, seule celle du nœud, par exemple de plus petit identificateur, aboutit (cette méthode est similaire aux mécanismes d'*élection* selon le critère d'extremum [123, 97, 144]), et le jeton créé par ce nœud extremum aura comme identité celle de ce nœud et comme estampille celle proposée par ce noeud. Les nœuds qui ne créent pas le jeton prennent alors acte du nouveau jeton.

Rappelons que durant le processus d'élection, chaque nœud *i* participant à l'élection propose son identité et un numéro de séquence local comme identification du nouveau jeton. A la fin de l'élection (à l'expiration de tous les temporisateurs *SuspicionTimer*), au niveau de chaque nœud de la partition, les variables qui contiennent l'identité du nœud candidat et l'estampille que ce dernier propose au jeton (pendant le processus d'élection) contiennent maintenant les valeurs finales et uniques qui sont l'identité de l'élu et l'estampille qu'il propose au jeton. A ce moment, *le seul nœud qui trouve l'identité suggérée au jeton égale à la sienne a sa charge la création du jeton.*

Quand un nœud *j* reçoit un message *IsThereToken ()*, il le traite comme une requête normale (i.e. il essaye de l'insérer dans sa file). Ensuite s'il n'est pas porteur du jeton, il considère ce message comme une exploration pour l'élection à la création d'un jeton. Ceci le mène à participer à cette élection en mettant à jour les variables appropriées et continue éventuellement la propagation de ce message et se bloque (, en déclenchant son propre temporisateur *SuspicionTimer,*) s'il ne l'était pas. Dans le cas où *j* serait porteur du jeton (i.e. pas de partitionnement), il ne traitera que le premier message reçu. Ce traitement se résume à *une attente* (i.e. *StabilisationTimer*) dans laquelle *j* s'assure que les différentes diffusions simultanées de ce type de message se soient arrêtées (dans cette période d'attente, *j* ne fait pas passer le jeton au prochain). Ensuite, il diffuse le message *ThereIsToken ()* dans tout le réseau.

4.2. La fusion

La fusion est détectée à l'occasion de créations de liens. A chaque création de lien, les deux nœuds le formant s'échangent les identifications des jetons qui leurs sont associés. En cas de duplication de jetons, un seul restera valide et l'autre sera supprimé.

Dès qu'un nœud *i* détecte un nouveau voisin, il lui transmet un message *LinkInfo ()* qui contient des informations sur son état de la manière suivante: Si *i* n'est pas bloqué, seules les informations (identité et estampille) sur le jeton utilisé par *i* sont transmises. Sinon, le message *LinkInfo()* inclus aussi les informations du message *IsThereToken()* traversant ou généré par *i*. Ce mécanisme permet de remédier aux cas de déconnexions momentanées: Le nœud correspondant peut être détenteur du jeton! On peut constater que pour un lien créé, deux messages sont échangés (par cause de symétrie), et donc dans un souci d'optimisation du nombre de messages *LinkInfo ()*, on impose la règle suivante: Si *i* est bloqué, alors il transmet systématiquement le message *LinkInfo ()*, mais si au contraire il est dans un état normal, un seul nœud (choisit de manière déterministe,) prend à sa charge la transmission de ce message.

Quand un nœud *j* reçoit un message *LinkInfo ()* de la part de *i*, s'il trouve que le jeton de *i* est le même que celui de *j* (i.e. le lien créé est interne à la partition) et si en plus *i* est bloqué, alors *j* exécute un traitement équivalent à celui exécuté lors de la réception d'un message *IsThereToken ()*. Par contre, si les jetons sont différents alors il y a eu une fusion et un jeton doit donc être supprimé; le choix porte par exemple sur le jeton le plus récent dans le cas de jetons créés par le même nœud ou sinon sur celui avec l'identité la plus grande). Si le jeton à supprimer est celui de la partition de *i*, *j* lui envoie un message *LinkInfo ()* normal. Dans le cas contraire, *j* diffuse le message *DeleteToken ()* dans sa partition après avoir vidé sa file d'attente.

Quand un nœud *k* reçoit un message *DeleteToken ()*, si le jeton de la partition locale est moins prioritaire, alors il met à jour ses informations locales par celles reçues, réinitialise son contexte (vide sa file, se débloque s'il était bloqué, désactive ses temporisateurs éventuels, etc.) et continue la diffusion de ce message. De plus si *k* est détenteur du jeton, il le supprime même s'il est en SC, et cela ne pose pas de problème car à la sortie de la SC, il se trouvera sans jeton.

5. Exemple d'exécution et discussion

5.1. Exemple d'exécution

La Figure 11 montre un scénario d'exécution du protocole d'une manière qui permet de faciliter la présentation. Initialement (Figure 11(a)), le nœud *4* est détenteur du jeton et les nœuds *2* et *7* veulent entrer en SC. On suppose que leurs estampilles respectives sont égales, mais leurs rayons de diffusion respectifs sont $R_2 = 1$ et $R_7 = 3$ (on note que la requête du nœud *7* atteint le nœud *2*, donc ce dernier possède une route vers le nœud *7*, mais pas dans l'autre sens). Puisque les deux requêtes (Figure 11(b)) arrivent avec la même estampille, c'est le nombre de sauts qui va déterminer l'ordre d'entrée en SC. Le nœud *2* étant le plus proche du nœud *4*, ce dernier à sa sortie de la SC le choisit comme prochain et lui passe le jeton (Figure 11(c)). Entre-temps, le nœud *1* sort de la portée du nœud *2* et *3* et entre dans la portée du nœud *7*, tandis que le nœud *3* entre dans la portée du nœud *2*, mais sans sortir de la portée du nœud *7*. Quand le nœud *2* sort de sa SC, il veut faire passer le jeton au nœud *7*, mais la route initialement construite (i.e. *1*, *3*, *7*) a été brisée, et donc le nœud *2* tente de la recalculer, la nouvelle route (i.e. *3*, *7*) sera obtenue (Figure 11(d)), et le jeton sera véhiculé via cette route vers le nœud *7* (Figure 11(e)). Dans la Figure 11(e), les nœuds *3*, *5* et *6* sont demandeurs de jeton, supposons que $R_3 = 1$, $R_5 = 3$ et $R_6 = 3$. Leurs requêtes arrivent au jeton avec les routes respectives *3*; *3*, *4*, *5* et *3*, *4*, *6*. Dans la Figure 11(f) (où des changements de liens ont eux lieu), on suppose que le nœud *6* est le prochain destinataire, le jeton lui est véhiculé via le

nœud *3*. Arrivant au nœud *3* et puisqu'il est demandeur (admettons que le nœud *6* n'a pas atteint le seuil de verrouillage), il profite de l'arrivée du jeton et entre à sa SC et à sa sortie il fait passer le jeton à son destinataire déjà prévu (i.e. le nœud *6*) après récupération de la route. Dans la Figure 11(g), le nœud *7* demande le jeton avec $R_7 = 1$ par exemple. A sa sortie de la SC, le nœud *6* qui a déjà reçu la requête du nœud *5* avec la route *4*, *5* n'arrive pas à lui passer le jeton. Le jeton est donc envoyé au nœud *3* (i.e. dernier détenteur du jeton) avec l'attribut *NORoute* qui à son tour l'envoie au nœud *7* (mais avec l'attribut *ONRoute*) puisqu'il a déjà reçu sa demande. Le nœud 7 utilise le jeton pour exécuter sa SC.

Dans la Figure 11(h), le nœud *7* est isolé temporairement. A sa sortie de la SC, il garde le jeton à son niveau. Admettons que le nœud *5* a atteint son seuil de redemande, donc il diffuse un message *IsThereToken ()* qui sera reçu par les nœuds *4* et *1*. Entre-temps (Figure 11(i)), un lien est créé entre les nœuds *7* et *1*, seul le nœud *1* (, puisque le noeud *1* est à l'état de recherche du jeton et *7* à l'état normal,) envoie le message *LinkInfo()* au nœud *7*. Ce dernier va donc constater la recherche du jeton. Après un délai de stabilisation *StabilisationTimer*, celui-ci envoie un message *ThereIsToken()* à tous les nœuds, afin de débloquer ceux qui étaient bloqués. Le nœud *4* diffuse une demande avec $R_4 = 2$. Supposons que le prochain destinataire est le nœud *5*, le jeton lui sera verrouillé puisqu'il a atteint le seuil *LockLimit* (admettons que *Locklimit= Req_Threshold*). Le nœud intermédiaire *4* n'est servi qu'au retour du jeton.

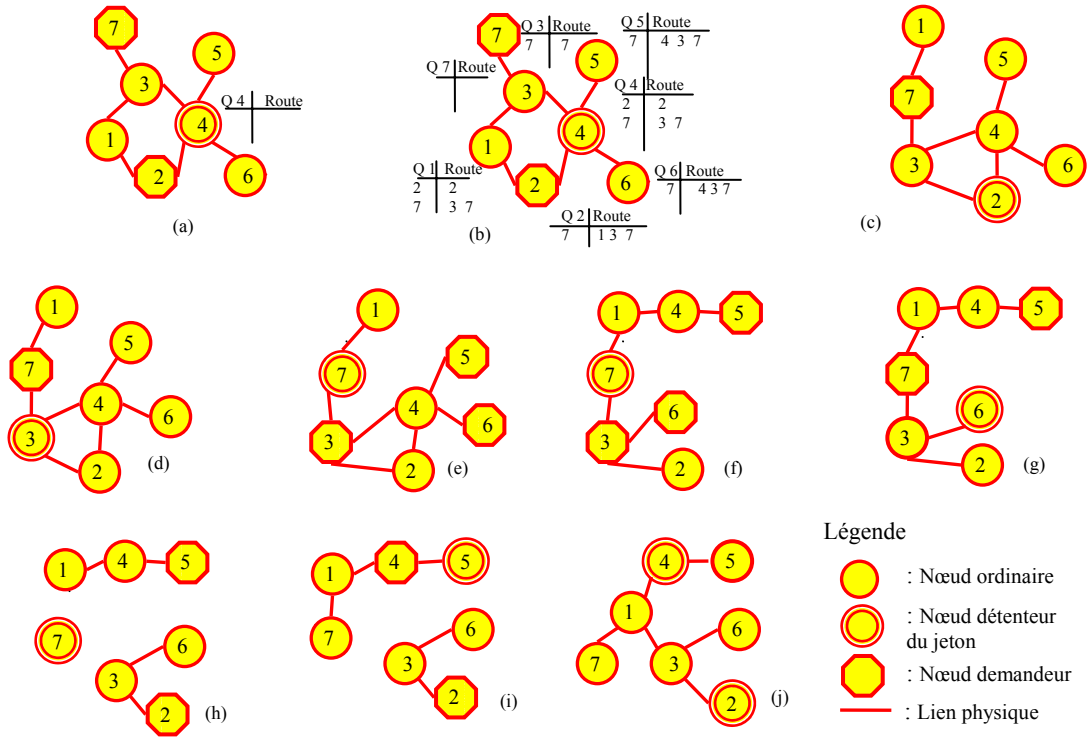


Figure 11: *Un simple exemple d'exécution du protocole d'exclusion mutuelle.*

Dans la Figure 11(i), un partitionnement a eu lieu. Le nœud *2*, après plusieurs tentatives de demandes, soupçonne la perte du jeton en diffusant un message *IsThereToken ()* mais sans réponse. Au bout du délai *SuspicionTimer,* puisqu'il est le seul à proposer la création du jeton, le nœud *2* le crée. Les nœuds *3* et *6,* chacun est amené à mettre à jour les informations sur le nouveau jeton suite aux expirations de leurs mêmes délais déclenchés lors de la réception du message de recherche du jeton. Dans la Figure 11(j), un rapprochement a eu lieu entre les nœuds *3* et *1*. Puisque les deux nœuds sont dans un état normal, seul le nœud *1*

envoie son message *LinkInfo ()*. Lors de sa réception par le nœud *3*, celui-ci constate la différence entre les deux jetons et diffuse dans sa partition un message de destruction du jeton. Le nœud *2* détruit alors le jeton et prend acte avec le nœud *6* de l'identification du nouveau jeton.

5.2. Discussion

Comment satisfaire les requêtes transportées par le jeton?

Rappelons que pour chaque requête rencontrée, le jeton transporte uniquement l'identité du nœud concerné et l'estampille de la requête. Le transport de routes augmente considérablement la taille du jeton et nécessite la mise à jour de cette route à chaque passage par un nœud. Ceci implique qu'au niveau de la file de chaque nœud, il peut y avoir présence de requêtes sans routes. Dans le pire des cas pour atteindre ces nœuds, le jeton circulera à leur recherche avec un état *NORoute*. Donc, un parcours arborescent du réseau sera effectué. Dans le cas général, le protocole prévoie d'autres mécanismes pour les servir (comme nœuds intermédiaires, nouvelles tentatives de demandes…)

Comment éviter au jeton de circuler inutilement en boucle?

Soit l'exemple de la Figure 12(a), le destinataire du jeton est le nœud *5* avec le chemin *2 3 4 5*. Arrivant au nœud *3*, dans la Figure 14(b), le lien avec le nœud *4* est rompu. Le nœud *3* ne peut plus router le jeton vers son destinataire. Supposons aussi qu'il ne trouve pas de nouveau destinataire (i.e. file vide), il décide donc de le redonner à son dernier détenteur qui est *2*, le nœud *2* reçoit le jeton avec un état *NORoute*, alors il choisit un nouveau destinataire qui est le nœud *6* à partir de sa file Q_2. Il renvoie donc le jeton au nœud *3* (qui est sur la route du nouveau destinataire) avec un état *ONRoute*. Le nœud *3* n'ayant toujours pas de route valide remet le jeton à son dernier détenteur qui est *2*, ainsi la boucle est formée et le jeton circulera uniquement entre *2* et *3*. De ce fait, dès que le nœud *3* retourne le jeton à *2*, ce dernier considère *3* comme un nœud à éviter, ce qui lui permet de choisir une destination autre que ce nœud.

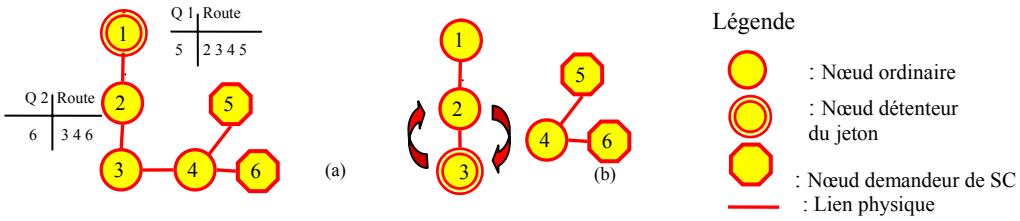


Figure 12: *Circulation en boucle du jeton.*

Pourquoi le détenteur du jeton attend-il un moment avant de diffuser le message ThereIsToken ()?

Admettons que le nœud possédant le jeton reçoit un message *IsThereToken ()* et réagit immédiatement en diffusant un message *ThereIsToken ()*. Un autre nœud recevant ce message se débloque immédiatement, mais si par la suite ce dernier reçoit un autre message *IsThereToken ()* du même flot, il se bloquera inutilement.

Pourquoi le message LinkInfo () transporte-il dans certains cas les informations d'un message IsThereToken ()?

Puisque les messages du type *IsThereToken ()* doivent être propagés dans tout le réseau, un nœud peut se déconnecter momentanément et donc peut ne pas recevoir ce message. De ce

fait, dès qu'il se reconnecte, nous assurons cette propagation en véhiculant dans le message *LinkInfo ()*, les informations correspondantes à *IsThereToken()*.

Comment réagit le protocole dans le cas de plusieurs fusions simultanées?

Pour simplifier la présentation, nous considérons trois partitions différentes, chacune correspond à une couleur (*jaune*, *rouge* ou *bleu*) et chaque couleur correspond au jeton d'une partition. Selon la politique qui gère la priorité des jetons, l'un d'eux sera plus prioritaire que les deux autres et donc si ce dernier correspond par exemple à la couleur *rouge*, les partitions *jaune* et *bleue* vont propager le message *DeleteToken ()* portant comme nouvelle couleur la couleur *rouge*, et donc après sa diffusion tous les nœuds prendront la couleur *rouge*. Un délai peut s'écouler entre le moment de fusion et celui de la suppression effective des jetons concernés dépendant de la propagation du message *DeleteToken ()*. Une situation transitoire de fonctionnement en mode dégradé peut être inévitablement observée avant de revenir au fonctionnement normal (voir la solution à ce problème dans la Section 7.3).

Le protocole tolère t'il la panne d'un nœud?

Lorsqu'un nœud tombe en panne, toutes les informations dont il dispose disparaissent; en l'occurrence la file d'attente, les temporisateurs, le jeton et les messages qui viennent d'être reçus nécessitant une propagation. Concernant la perte des requêtes reçues localement, dans le cas échéant le mécanisme de relance remédie à ce problème; similairement à la perte de jeton, des mécanismes de soupçon de perte et de régénération remédient à ce problème. Pour les temporisateurs, seul *StabilisationTimer* semble être gênant; celui-ci nécessite la diffusion d'un message qui indique la présence du jeton. En réalité, ce message n'a pas de sens puisque le jeton est perdu avec la panne de ce nœud. En ce qui concerne la propagation de messages, une gène pourrait apparaître lorsque la panne de ce nœud empêche une partie du réseau d'être informée par ce message; en particulier, le message de recherche de jeton. Dans ce cas, cette partie du réseau sera considérée comme une autre partition. Des mécanismes liés partitionnement sont déjà prévus pour ce cas.

Que se passe-t-il en retour de panne d'un nœud?

Un nœud *i* qui retourne après une panne doit naturellement réinitialiser son contexte, en particulier l'identité du jeton et son estampille à *-1*. Puisque son retour, vis-à-vis de ses voisins se manifeste par des détections de liens qui vont être suivis de l'échange de messages *Linkinfo ()*, ces messages finiront par lui apprendre l'identification du jeton de la partition qu'il vient d'intégrer; c'est un cas de fusion de partitions. Ces mécanismes supposent qu'à chaque fois qu'il y a fusion de deux partitions, le choix du jeton à supprimer doit porter sur le *plus jeune* dans le cas de jetons créés par le même nœud ou sur celui avec l'identité *la plus petite* dans le cas contraire. Admettons que la valeur de son numéro de séquence est celle du numéro de séquence du jeton de la partition qu'il vient de rejoindre. Ceci n'empêche pas l'existence de partitions multiples avec la même identification. En effet, supposons qu'un nœud *i* faisait parti avant sa panne d'une partition identifiée par *(i, x)* (cela veut dire que le jeton de cette partition a été créé par *i*); en revenant d'une panne, rien n'empêche qu'il se retrouve dans une autre partition identifiée par *(j, x-1)*. Il aura donc comme numéro de séquence (*x-1)*. Plus tard, s'il réussit à créer un nouveau jeton dans cette partition après perte, ce jeton aura comme identification *(i, x).* Ceci engendre la non détection de fusion et par conséquence la cohabitation de deux jetons dans une même partition. Pour remédier à ce problème, on affecte à chaque nœud un nouvel attribut ''*Return_After_fail''*. Chaque nœud qui retourne après une panne se voit affecter son attribut *Return_After_fail* de *true*. Le jeton courant est aussi affecté d'un attribut ''*Token_After_Fail''*, celui-ci correspond à l'attribut *Return_After_fail* du nœud qui l'a crée. Ainsi, la fusion de deux partitions avec jetons créés par le même nœud est détectée si *exclusivement* l'un des attributs des deux jetons possède

l'attribut *Token_After_fail=true*. Dans ce premier cas, c'est le jeton qui possède l'attribut *Token_After_fail=true* qui doit être supprimé. Dans le second cas, les deux jetons ont l'attribut *Token_After_fail=true*, ce cas suppose qu'un même nœud peut tomber en panne plus d'une fois. La fusion n'est pas détectable dans ce cas.

Dans le but d'avoir une identification unique pour le jeton, une autre solution simple pourrait être envisagée. Il s'agit d'ajouter un troisième champ dans l'identification du jeton, dont la valeur est générée à travers l'utilisation d'une fonction aléatoire. Ce champ est généré à chaque fois que le besoin se fait sentir (i.e. à l'initialisation et à chaque retour après panne); dans ce cas, le numéro de séquence peut redémarrer (après panne) sans inconvénient à partir de la valeur *0*. Mais, un risque apparaît si la fonction génère la même valeur pour le même nœud !

6. Etude de correction

Le protocole défini plusieurs outils dont le but essentiel est la satisfaction des deux propriétés, en l'occurrence la sûreté et la vivacité. En premier, on s'intéressera à la sûreté. Cette propriété, rappelons le, implique qu'à tout instant au maximum un nœud se trouve en SC. Rappelons qu'un jeton est géré au niveau de chaque partition. De plus, le seul nœud détenteur de ce jeton est permis d'accéder à sa SC. Si le protocole assure qu'à tout moment un seul jeton réside au niveau de chaque partition, la propriété de sûreté est alors vérifiée. Evidemment, dans l'absence de la création de jeton et du fusionnement de partitions, cette propriété est assurée étant donné qu'un seul jeton circule dans la partition courante. Nous allons étudier la sûreté dans deux situations : A la création d'un nouveau jeton après perte, à la suppression d'un ou plus de jeton(s) durant le fusionnement de partitions.

Pour la première situation (voir aussi la Section 4.1), supposons qu'un ou plusieurs nœuds sont en cours de recherche du jeton. Tous les noeuds de la partition courante seront informés au même titre que ceux en déconnexion temporaire (par l'utilisation des temporisateurs qui attendent leurs reconnexions et des messages d'informations *LinkInfo ()*). Si un nœud détient le jeton (ou le reçoit pendant qu'il est dans l'état de recherche du jeton), après un délai *StabilisationTimer*, ce noeud informe tous les autres de la non perte du jeton et ainsi la recherche en cours s'arrête. (Notons que la valeur du temporisateur *StabilisationTimer* est calculée en sorte qu'elle permette à tous les nœuds d'être informés avant l'expiration des temporisateurs *SuspicionTimer*). Autrement, à l'expiration de son temporisateur, un nœud doit créer le nouveau jeton avec une nouvelle identification. Ce *seul* nœud a été choisi par un processus d'élection qui a accompagné la recherche du jeton. Comme résultat de cette élection, tous les nœuds sont informés de l'identité du nouveau jeton. Par conséquent, un jeton existe dans la partition ce qui vérifie la sûreté.

Pour la seconde situation (voir aussi la Section 4.2), supposons deux partitions chacune avec son propre jeton identifié de manière unique. Pour faciliter la présentation, considérons que chaque partition est colore différemment (en *bleu* ou en *rouge*) et cette couleur correspond à ce jeton. Admettons maintenant qu'un lien est établi entre les nœuds *i* et *j* appartenant chacun à l'une de ces deux partitions. Quelque soit la politique qui gère la priorité des jetons, un jeton devient moins prioritaire par rapport à l'autre, par exemple le *bleu*. Par conséquent, le nœud *j* va diffuser dans sa partition le message *DeleteToken ()*. Ainsi, à la fin de cette déssimination, tous les nœuds vont être de couleur *bleu*. Un délai peu s'écouler entre le moment du fusionnement et la suppression effective du jeton en dépendant de la vitesse de propagation du message de suppression du jeton. Des mécanismes ont été définis pour

remédier aux effets éventuels de cette période transitoire (voir Section 7.3). La propriété de sûreté est alors vérifiée.

Considérons maintenant la propriété de vivacité qui implique que chaque nœud demandeur de SC (donc de jeton) arrive à y accéder. Plusieurs mécanismes ont été appliqués pour favoriser cette propriété. Le premier mécanisme est l'utilisation du rayon de diffusion et le transport de requêtes par le jeton (voir Section 2.2.1). Le rayon de diffusion permet de créer une cascade de chemins vers le nœud détenteur du jeton. Ce qui implique que ce nœud est indirectement au courant des requêtes lointaines. Mais, ce mécanisme est insuffisant pour satisfaire toutes les requêtes, d'où la collecte de requêtes rencontrées par le jeton lors de son parcourt. Par conséquent, tous ces nœuds seront éventuellement satisfaits en dépendant de la mobilité et des pannes. Le second mécanisme est la relance de requêtes (voir Section 2.2.2). En effet, un nœud qui ne reçoit pas le jeton durant une période fixe doit relancer sa requête. Afin de garder son ancienneté, cette requête est relancée avec la même estampille. Dans le cas échéant, ce processus est répété un certain nombre de fois (i.e. *Req_Threshold*) avant de suspecter la perte du jeton. Le troisième mécanisme est évidemment la recherche et la régénération du jeton (voir Section 4.1). Quand la perte du jeton est détectée, celui-ci sera régénéré et par suite les nœuds demandeurs régénèrent leurs requêtes après un délai fixe. Deux autres mécanismes sont aussi utilisés, ils permettent d'éviter la famine de nœuds. Le premier est la politique de satisfaction de requêtes (voir Section 2.2.3) qui utilise l'ancienneté des requêtes. Au court du temps, les anciennes requêtes deviennent nécessairement plus prioritaires et par suite, servies en premier. Le second mécanisme est le verrouillage du jeton (voir Section 2.2.4). Quand un nœud est sélectionné comme prochain privilégié, après un seuil de relances de la requête de ce nœud, le jeton lui est réservée exclusivement. Les nœuds intermédiaires ne peuvent l'utiliser. Pratiquement, si on se fie aux résultats de simulations (qui seront détaillés dans la Section 7.2), nous constatons que la majorité des requêtes d'admission en SC sont satisfaites dès la première demande et au maximum après *0.2* redemande.

7. Simulation

L'évaluation des performances du protocole est effectuée à l'aide de l'outil NS2 (*Network Simulator 2*) [136] version 2.26 sous le système d'exploitation LINUX MANDRAKE 8.2. La codification du protocole est effectuée à l'aide du Langage objet Otcl.

7.1. Environnement de simulation et paramètres

L'environnement dont les différentes simulations ont été effectuées, peut être décrit à travers un certain nombre de paramètres. Les paramètres constants sont: la surface de mouvement est de *1000 × 500 m*; le nombre de nœuds dans le réseau est de *25*; la durée de la SC est de *0.1 s*; le rayon de communication entre nœuds mobiles est de *250 m*; le temps de simulation est de *100 s*. Le nombre de ressources du réseau est *10*. Les paramètres variables sont:

- La mobilité: Le modèle de mobilité utilisé est le modèle aléatoire se basant sur le mouvement relatif des nœuds (voir Chapitre I). A partir de la formule de la mobilité, nous avons: mobilité faible: *0* < Mob <= *3*; mobilité moyenne: *3* < Mob <= *8*; mobilité élevée: *8* < Mob. Pour la simulation, nous avons utilisé quatre valeurs de mobilité: *0*, *1.5*, *5*, *10*. Avec les différents scénarios de mouvements réalisés lors de la simulation, il ressort que la connectivité moyenne varie entre *5* et *7* voisins par nœud.

- La charge: elle est définie par une loi de poisson de paramètre λ dont les valeurs prises pour la simulation sont: *0.07, 0.10, 0.125, 0.167, 0.25, 0.5, 1* correspondants respectivement aux intervalles (*1/λ*) *15s*, *10s*, *8s*, *6s*, *4s*, *2s*, *1s* durant lesquels un nœud, après satisfaction, génère une nouvelle demande d'entrée en SC.

7.2. Résultats et interprétations

Le nombre de messages par SC (nb_MSG /SC)

Cette métrique représente le nombre moyen de messages nécessaires pour la réalisation d'une SC. Elle est calculée par: *((nombre de messages de demandes d'entrée en SC + le nombre de messages de redemandes + le nombre de sauts effectués par le jeton)) / nombre d'entrées en SC*. Comme le montre les graphes de la Figure 13, les deux groupes de graphes ont une même allure, i.e. *nb_MSG/SC* est décroissant et donc ''inversement proportionnel'' à la charge. On note également, une certaine stabilisation au niveau des moyennes et fortes charges (*0.5*, *1*). Par ailleurs, nous pouvons remarquer que pour les moyennes et fortes charges l'impact de la variation de la mobilité sur le *nb_MSG/SC* est insignifiant, contrairement à ce qui se passe pour les faibles charges où les tracés présentent des pentes différentes. Tous ces résultats s'expliquent par le fait que plus la charge des demandes est importante, plus le nombre de requêtes valides dans les files d'attentes des nœuds sera élevé; d'où la présence très probable de demandes de nœuds proches les uns des autres (en terme de nombre de sauts). Ceci implique que le rayon de diffusion calculé est assez faible, ce qui réduit le nombre de messages circulants. La Figure 14 montre que le rayon moyen de diffusion varie, selon la charge, entre *1* et *2* sauts et se stabilise à la valeur *1* pour les moyennes et fortes charges. Ceci explique la diminution du nombre moyen de messages par SC pour ces charges.

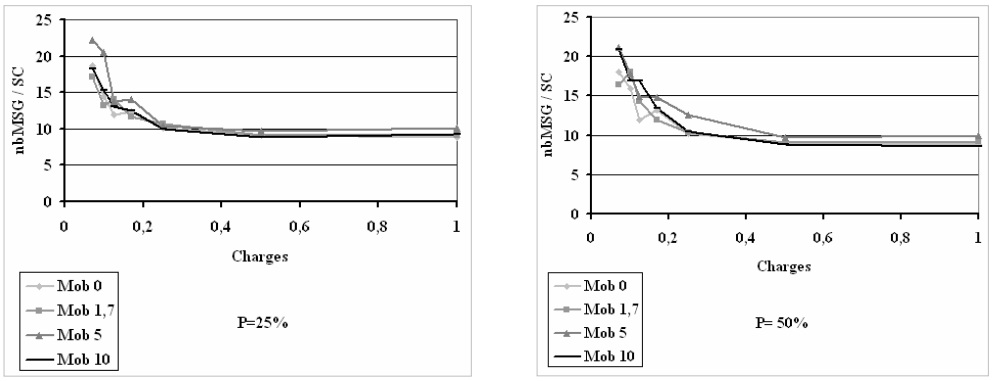


Figure 13: *Les graphes représentant le nombre de messages par SC.*

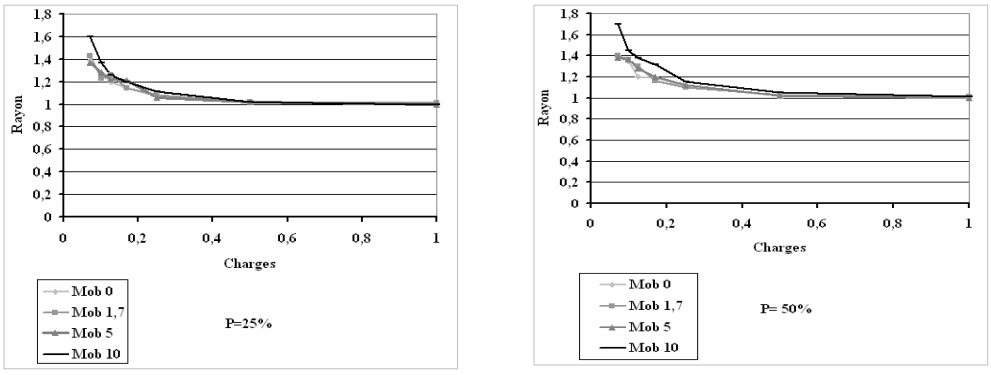


Figure 14: *Les graphes représentant le rayon moyen de diffusion.*

Le délai de synchronisation (nbTokenHops/SC)

Il indique le nombre moyen de sauts effectués par le jeton entre deux SC. Il est calculé par: *nombre de sauts effectués par le jeton / nombre d'entrées en SC*. Les deux groupes de graphes de la Figure 15 ont une même allure. Pour les valeurs de mobilité *0* et *1.7*, les tracés sont presque horizontaux avec des valeurs faibles. Ce qui implique que, pour des mobilités faibles, *nbTokenHops/SC* est assez stable pour toutes les charges. Pour les valeurs de mobilités *5* et *10*, les tracés prennent une allure décroissante et convergente vers les mêmes valeurs que celles des deux premiers tracés. Donc, pour ces mobilités la charge a eu un impact assez fort. De par sa nature, le protocole cherche à satisfaire les nœuds les plus proches (en nombre de sauts). Aussi, chaque nœud demandeur se trouvant sur la route du jeton sera potentiellement satisfait et donc, pour les moyennes et fortes charges, le jeton circule la plupart du temps entre des nœuds demandeurs, ce qui explique le court délai de synchronisation quelle que soit la mobilité. En ce qui concerne les faibles charges, le jeton circule davantage entre des nœuds non demandeurs, afin d'atteindre ceux qui veulent accéder à la SC. Ceci dit, plus la mobilité est élevée, plus le risque de cassures de liens et donc de rupture de routes est plus important, et donc dans ces conditions, le jeton va passer beaucoup plus de temps à chercher les demandeurs de SC avant de les atteindre, ce qui explique les constatations précédentes.

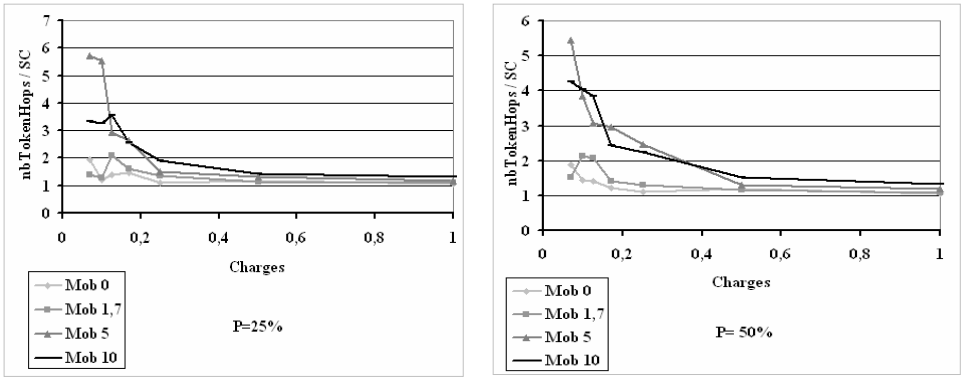


Figure 15: *Les graphes représentant le délai de synchronisation.*

Le nombre d'entrées en SC par nœud (nbSC / nœud)

Cette métrique indique le nombre moyen d'entrées en SC par nœud. Elle est calculée de la manière suivante: *nombre d'entrées en SC/ nombre de nœuds*. Pour cette métrique, on remarque que les deux groupes de graphes de la Figure 16 ont une même allure, les tracés sont croissants, suivent une même pente (allure logarithmique) et sont ''proportionnels'' à la charge. On peut aussi constater que l'impact de la mobilité sur le nombre d'entrées en SC par nœud n'est pas très conséquent. Ces résultats s'expliquent par le fait que plus la charge est importante, plus la fréquence des demandes d'entrée en SC augmente ce qui induit une augmentation du nombre d'entrées en SC. Ceci dit, l'allure logarithmique des courbes montre qu'à partir d'un certain seuil de charge, le nombre de satisfactions par nœud tend à se stabiliser. En effet, dans les moyennes et fortes charges tous les nœuds du réseau demandent le jeton et donc un nœud devra attendre les satisfactions des autres nœuds avant d'être lui-même satisfait.

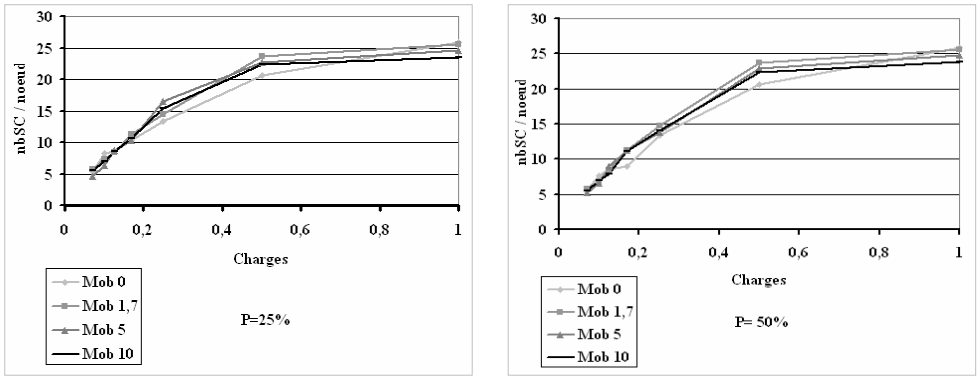


Figure 16: *Les graphes représentant le nombre d'entrées en SC par nœud.*

Le nombre de messages redondants (nb_MsgRedondants / SC)

Cette métrique indique le nombre de messages redondants. Elle est calculée par: *(nombre de messages Request reçus - nombre de messages Request traités) / nombre d'entrées en SC*. Les deux groupes de graphes de la Figure 17 ont une même allure, s'approchant de *0* pour les moyennes et fortes charges, la mobilité n'a pas eu beaucoup d'influence (à un message près). La redondance est due à l'insuffisance de l'information transportée par chaque requête concernant l'ensemble des nœuds visités, i.e. plus le rayon de diffusion est grand (ce qui est le cas pour les faibles charges), plus il y a de redondances dans les messages émis.

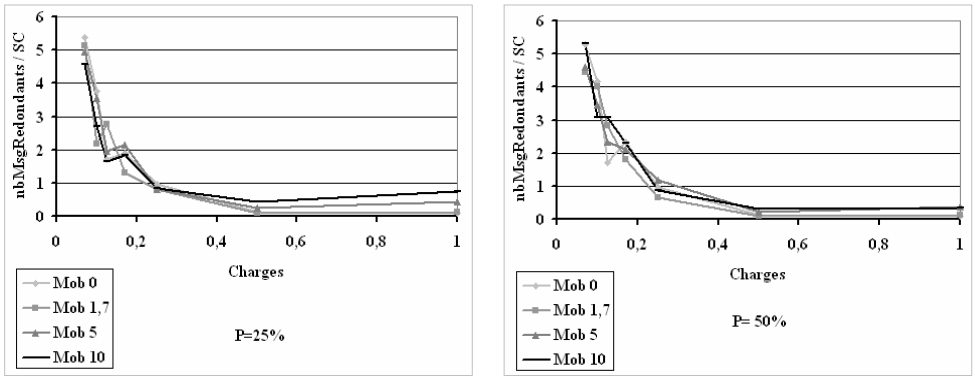


Figure 17: *Les graphes représentant le nombre de messages redondants par SC.*

Taux de redemandes (nb_Redemandes / SC)

Cette métrique indique le taux de redemandes d'entrées en SC lié au nombre total d'entrées en SC. Elle est calculée par: *nombre de redemandes / nombre d'entrées en SC*. Nous pouvons remarquer dans la Figure 18 que, globalement, le taux de redemandes est relativement faible, et nul pour les mobilités faibles à moyennes et fortes charges. Ce taux assez faible montre que la majorité des demandes d'entrées en SC sont satisfaites dès leurs premières tentatives. Ceci revient au fait que pour les moyennes et fortes charges, le nombre de demandes d'entrée en SC simultanées est important et donc puisque chaque nœud va être au courant des demandes des uns ou des autres, le jeton, dans sa circulation, parcoure les nœuds en satisfaisant leurs demandes.

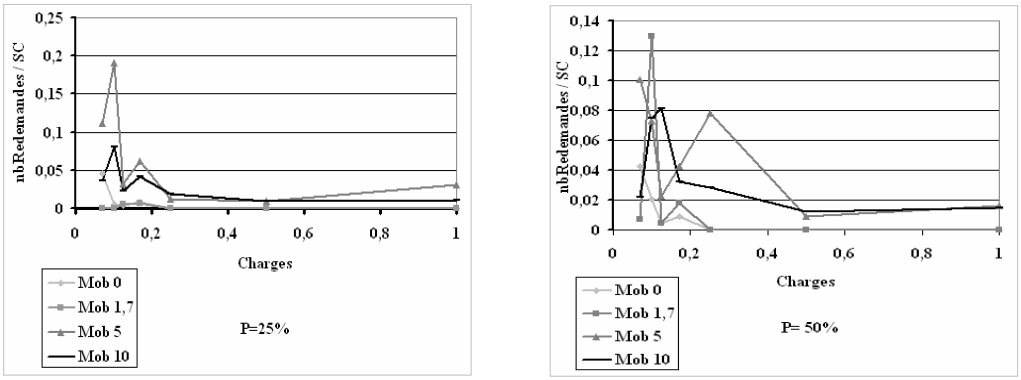


Figure 18: *Les graphes représentant le taux de redemandes d'une SC.*

7.3. Résultats se rapportant au partitionnement

La principale mesure effectuée concerne le nombre de SC exécutées après fusion mais à l'aide du jeton qui va être supprimé. Après chaque fusion de deux ou plusieurs partitions, un seul jeton reste valide et les autres sont supprimés dans des délais assez brefs. Parmi les influences possibles, nous pouvons citer: le nombre de nœuds dans la partition concernée, la vitesse de propagation du message *DeleteToken ()* et l'endroit où se trouve le jeton à supprimer lors de la fusion. La Table 3 montre le nombre de SC exécutées après fusion (à l'aide du jeton qui va être supprimé). Cette table montre que le nombre de SC exécutées après fusion est nul dans la majorité des cas et prend la valeur *1* comme maximum. Ce résultat montre que ce nombre ne dépend pas réellement du nombre de nœuds dans la partition mais de l'endroit du jeton et de la vitesse de propagation du message *DeleteToken ()* par rapport au temps d'exécution d'une SC.

Afin de remédier à ce problème de duplication temporaire de jeton (i.e. de non sûreté), une solution simple peut être adoptée: Avant la fusion (admettons deux partitions), un jeton existe dans chacune des deux partitions concernées. Quand un lien se forme avec conséquence la fusion de ces deux partitions, on peut faire si comme si ''la fusion n'est pas encore réalisée'' jusqu'à la destruction effective du jeton et la sortie du dernier nœud de sa SC dans la partition ciblée. Il s'agit d'interdire la communication entre ces deux partitions, en évitant de véhiculer les messages à travers ce lien pendant une durée limitée par un temporisateur.

D'autres tests ont montré que chaque partition ne possédant pas de jeton, aboutit à la création de ce dernier après consensus. Cette création de jeton survient après des délais limités (directement proportionnels au seuil de propagation).

Nombre de nœuds dans la partition qui contient le jeton à supprimer.	4	7	10	13	17	20
Nombre de SC exécutées au moment de la fusion (la fusion a eu lieu en cours de ces SC)	0	0	1	1	1	1
Nombre de SC exécutées après la fusion (avant que le message *DeleteToken()* n'atteigne le porteur du jeton)	0	1	1	0	0	0

Table 3: *Nombre de SC exécutées au moment et après la fusion.*

Remarquons qu'après toutes les mesures réalisées sur ce protocole, son comportement n'a pas sensiblement changé entre les valeurs *25* et *50%* de la proportion *P*.

7. Conclusion

Dans ce chapitre, nous avons présenté une solution au problème d'EM dans les réseaux mobiles ad hoc. Le protocole présenté se base sur les liens physiques et n'utilise aucune structure logique. Les requêtes sont acheminées suivant un rayon dynamique de diffusion déterminé selon une connaissance locale permettant de couvrir une proportion *P* de nœuds demandeurs. Le jeton est géré selon une méthode qui combine la circulation et la demande. De plus, la politique de satisfaction des nœuds favorise en même temps les nœuds les plus proches et les requêtes les plus anciennes. La mobilité des nœuds fait partie intégrante de la solution proposée grâce aux différents mécanismes entrepris dans le protocole. Aussi, le protocole prend en charge les pannes de nœuds, le partitionnement, la fusion et par conséquent, tous les problèmes qui en résultent (perte de jeton, création de nouveaux jetons, suppression des duplications.). Cependant, le protocole tolère *un seul* retour après panne pour chaque nœud qui tombe en panne; ainsi, il remédie aux problèmes qui surviennent suite aux créations de jetons par des nœuds qui retournent après pannes. Le protocole est distribué, équitable, satisfait les requêtes de manière symétrique, favorise la scalabilité du réseau et considère que le nombre de nœuds est indéfini.

A partir des différents tests ainsi que de l'ensemble des métriques effectuées, nous pouvons constater que le protocole donne des résultats satisfaisants dans sa globalité. Le nombre de messages qu'il génère est assez faible surtout pour les moyennes et fortes charges. Le protocole se montre bien adapté à la mobilité, car cette dernière n'influe pas beaucoup sur ses performances globales. Le nombre de messages émis par SC est assez faible surtout pour les moyennes et fortes charges car le rayon tend vers la valeur *1*. Le délai de synchronisation dépend de la mobilité mais se stabilise au fur et à mesure que la charge du réseau augmente. Le nombre d'entrées en SC est proportionnel à la charge. Aussi, le nombre de messages redondants chute avec l'augmentation de la charge. Le taux de redemandes est relativement faible dans sa globalité puisqu'une demande d'entrée en SC n'est que très rarement non satisfaite dès sa première tentative.

Chapitre IV

La *k*- exclusion mutuelle distribuée : Un état de l'art

1. Introduction

L'allocation de ressources est un domaine essentiel dans les systèmes en général et dans les systèmes distribués en particulier. Le point d'entrée vers ce domaine est le problème de l'exclusion mutuelle qui consiste à gérer l'accès à une ressource commune et exclusive. Ce problème a été largement abordé dans la littérature et plusieurs protocoles ont été proposés (voir Chapitre II). La plupart de ces protocoles répondent aux contraintes liées à ce problème et essaient de fournir des solutions efficaces et fonctionnant en présence des pannes de noeuds et de liens de communication. Quand la ressource est présente en plusieurs exemplaires, le problème devient comment permettre à *k* processus d'utiliser simultanément chacun un exemplaire (parmi *k*) de ressource. Il ressort que chaque ressource peut seulement être accédée par un processus à la fois, et chaque processus peut accéder à au plus une ressource à la fois, c'est donc le problème de la *k- exclusion mutuelle* (ou de la *section critique multiple*).

Plusieurs protocoles de la *k*- EM ont été proposés dans la littérature selon les deux approches classiques connues dans l'EM. Quelques solutions sont orientées jeton [27, 82, 94, 127] avec soit *un* soit *k* jetons; d'autres utilisent l'approche à permission d'arbitre (moyennant le concept de *k*- coterie, extension du concept de coterie) pour les solutions de [67, 100] et permission individuelle pour la solution de [109] par exemple.

La première solution de *k*- EM connue des systèmes distribués est due à K. Raymond [109], par extension de l'algorithme de G. Ricart et A.K. Agrawala [117]. Par la suite, P. K. Srimani et R. L. N. Reddy dans [127] proposent une amélioration à ce protocole sur la base de la notion de privilèges de I. Suzuki et T. Kasami [133] en utilisant *k* jetons. Leur solution réduis à moitié le nombre de messages requis pour une entrée en SC, mais elle n'est pas résistante aux pannes. Dans [94], M. Naimi développe un protocole distribué simple qui généralise la solution d'EM de M. Naimi et M. Trehel [93] à *k* entrées simultanées en SC en se basant sur l'utilisation de *k* jetons et sur une structure logique *dynamique* de graphe orienté. Dans [82], K. Makki et *al* proposent un simple protocole de *k*- EM orienté jeton dans lequel

un jeton circule sur demande à travers les processus et permet aux seuls processus le recevant avec un sémaphore non nul d'accéder à leurs SC. Dans [27], S. Bulgannawar et N. H. Vaidya proposent un protocole de *k*- EM qui étend celui d'EM de M. Maimi et M. Trehel [93] en utilisant *k* jetons et une structure de forêt *dynamique* pour chaque jeton. Avec l'introduction de la notion de *k*- coterie, plusieurs solutions ont été proposées [100, 67, 57, 34, 1, 73], elles admettent un même principe mais diffèrent sur la manière de construction des structures de *k*-coteries. Ainsi, dans le but d'ajouter de nouvelles propriétés qui augmentent l'efficacité des *k*-coteries, d'autres concepts ont apparus; citons par exemple, les *k*- coteries non dominées [100], le concept de *k*- majority [34], les structures de cohortes [57], l'arbre de quorums [34] etc… La majorité de ces solutions sont tolérantes aux pannes de processus et possèdent un degré de disponibilité de plus en plus haut.

La seule solution de k-EM connue dans les réseau mobiles ad hoc est due à Walter et *al* [147]. Elle est orientée jeton et est une généralisation du protocole d'exclusion mutuelle développée dans [150], ce qui lui permet d'hériter de ses principaux mécanismes. Néanmoins, cette solution n'est pas tolérante aux pannes et au partitionnement.

Ce chapitre est dédié à la présentation des solutions sues citées. Une simple comparaison est alors réalisée sur la base d'un tableau (Table 4) qui récapitule les éléments importants concernant ces protocoles. Ce chapitre est structuré de la manière suivante: La Section 2 présente les solutions les plus connues de la *k*- EM dans les systèmes distribués selon les deux approches principales connues. La Section 3 expose plus ou moins en détail la seule solution de *k*- EM connue dans les réseaux mobiles ad hoc. Finalement, la Section 4 conclue le chapitre.

2. Solutions classiques de *k*- EM dans les systèmes distribués

Dans cette section, nous présentons les solutions classiques de *k*- EM connues dans la littérature des systèmes distribués. Nous insistons sur les éléments importants des solutions ciblées.

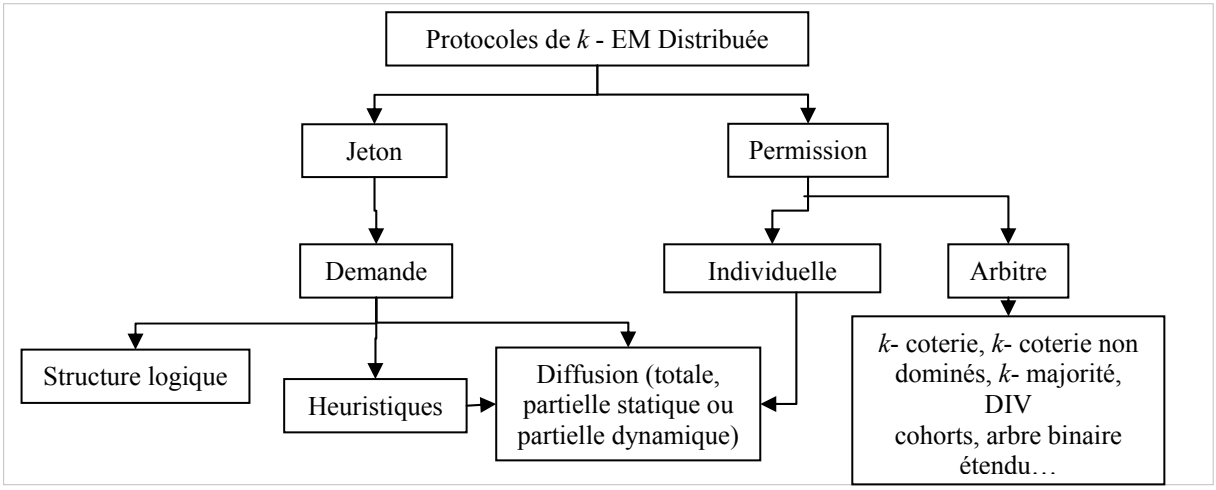


Figure 19 : *Une classification des protocoles de k- EM distribués.*

La Figure 19 donne une proposition de classification des ces solutions et la Table 4 donne un récapitulatif avec classement des solutions classiques. Les hypothèses communes à toutes les solutions sont: Le réseau sous jacent est connexe et composé de *N* nœuds fiables numérotés de *1* à *N*, les liens sont fiables, bidirectionnels et FIFO (le cas d'ordre arbitraire

sera indiqué), chaque nœud connaît ses voisins. Les cas de tolérances aux pannes seront indiqués. Par la suite, nous nous plaçons au niveau d'un nœud désigné par *i*.

2.1. Solution de K. Raymond

Dans [109], K. Raymond a développé la première solution de *k*- EM connue des systèmes distribués. Son protocole orienté *permission individuelle à diffusion totale* est une extension de celui de G. Ricart et A. K. Agrawala [117] afin de permettre à *k* processus du système d'exécuter leurs SC simultanément. La différence essentielle avec le protocole de [117] est le coût d'une SC qui est réduit à un minimum nécessaire de (*N-k*) messages de réponses. En effet, ce nombre de réponses est suffisant pour accéder à une SC. Il en découle une borne inférieure de coût total pour une SC de (*2*N-k-1*) messages et une borne maximale de *2*N* messages, puisque chaque message de requête requière un message de réponse.

Un processus désirant accéder à sa SC est amené à envoyer sa requête estampillée [76] aux *N-1* autres processus et à attendre la réception de (*N-k*) réponses pour lui permettre d'y accéder. Un processus recevant une requête, répond immédiatement s'il n'est pas intéressé par la SC ou son estampille est plus récente que celle de la requête reçue; autrement, il la défère jusqu'à la sortie de la SC. Ce comportement montre clairement qu'au plus *k* processus peuvent exécuter leurs SC en même temps. Un processus qui a obtenu suffisamment de permissions et qui est en SC, ou qui sort de sa SC peut recevoir des réponses parmi celles manquantes; un mécanisme spécial est prévu pour ignorer ces réponses.

Dans ce protocole, un processus est amené à envoyer sa requête à tous les autres processus alors que (*N-k*) réponses sont nécessaires pour exécuter sa SC. Une variante de ce protocole permet de réduire dans les meilleurs cas le nombre de messages de demandes de SC à un minimum de *N-k* (sur la base des réponses non reçues des anciennes demandes); mais, suite à des tests effectués le gain est insignifiant.

Le protocole de K. Raymond [109] hérite des mêmes avantages et inconvénients de son prédécesseur [117]. En effet, il est simple d'implémentation mais reste coûteux en messages et non tolérant aux pannes.

2.2. Solution de M. Naïmi

Dans [94], M. Naimi a développé un protocole distribué simple qui généralise la solution d'EM de M. Naimi et M. Trehel [93] à *k* entrées simultanées en SC en se basant sur l'utilisation de *k* jetons et sur une structure logique *dynamique* de graphe orienté. Ce protocole requière au maximum *2*(N-1)* messages pour accéder à une SC. Le graphe dirigé est maintenu par une structure de pointeurs distribués sur les processus. Initialement, le graphe est défini par une arborescence orientée vers un processus arbitraire qui détient tous les jetons. Le protocole suppose un réseau connexe. Chaque processus *i* entretient un *multi-ensemble* (i.e. un même élément peut y apparaître plusieurs fois,) de pointeurs $pred_i$ qui indique les processus auxquels la demande d'accès en SC doit être véhiculée, et une file d'attente de requêtes $suivant_i$ dont le premier élément est le processus auquel le jeton, disponible suite à la sortie de SC, doit être véhiculé. L'essentiel de cette solution réside dans le mécanisme de mise à jour de la structure pour servir la séquence de requêtes des processus.

Un processus *i* qui demande la SC, s'il dispose d'un jeton libre accède directement à sa SC. Dans le cas contraire, il envoie sa requête à tous ses *''prédécesseurs''* dans le graphe orienté (dont il devient sa racine), met son identité en tête de sa file $suivant_i$ et attend le jeton. Le graphe est alors transformé en un autre graphe comme dans [93]. Quand un processus *i* reçoit une requête d'un processus voisin *j*, s'il dispose d'un jeton libre il le lui sera immédiatement

véhiculé. Autrement, *i* enfile la requête localement et transmet une requête sous son nom à ses prédécesseurs (autres que *j*) qu'il supprime de $pred_i$, ce qui engendre la réorganisation du graphe. Dans les deux cas, *j* devient le prédécesseur de *i*. Lorsqu'un processus *i* sort de sa SC, il envoie le jeton au premier élément de sa file $suivant_i$ s'il en existe et l'enlève de cette file. La réception d'un jeton au niveau d'un processus *i* engendre son entrée en SC si l'identité de *i* est en tête de file $suivant_i$, sinon le jeton sera véhiculé au processus voisin en tête de file; l'identité du bénéficiaire est alors supprimée de cette file.

La Figure 20 qui est prise de [94] illustre le fonctionnement du protocole de M. Naimi. La Figure 20(a) montre un graphe de 5 processus reliés entre eux pour former un réseau connexe. La Figure 20(b) montre l'arborescence, construite à l'état initiale, dont la racine est le nœud *1* qui détient les deux jetons du système. Dans cette figure, $Pred_1$= {}; $Pred_2$= {1,1}; $Pred_3$= {2}; $Pred_4$= {2} et $Pred_5$= {1}. Supposons que les processus *1*, *3* et *5* désirent accéder à leurs SC. Le processus *1* détient un jeton libre, il entre alors immédiatement à sa SC. Le processus *3* met son identité en tête de sa file et envoie une requête au processus *2*. Le processus *5* fait de même que *3*, mais demande la SC à *1*. Le processus *2* reçoit la requête de *3*, il l'enfile et transmet une requête à *1*. Le processus *1* reçoit une requête de *5*, puisque *1* détient un jeton libre il le lui envoie. Le graphe orienté devient alors celui de la Figure 20(c). Le processus *1* reçoit une requête de *2*, il l'enfile et envoie une requête à *5*. Le processus *5* reçoit un jeton de *1* et entre en SC. Le processus *5* reçoit une requête de *1* qu'il enfile localement. Le graphe orienté devient alors celui de la Figure 20(d).

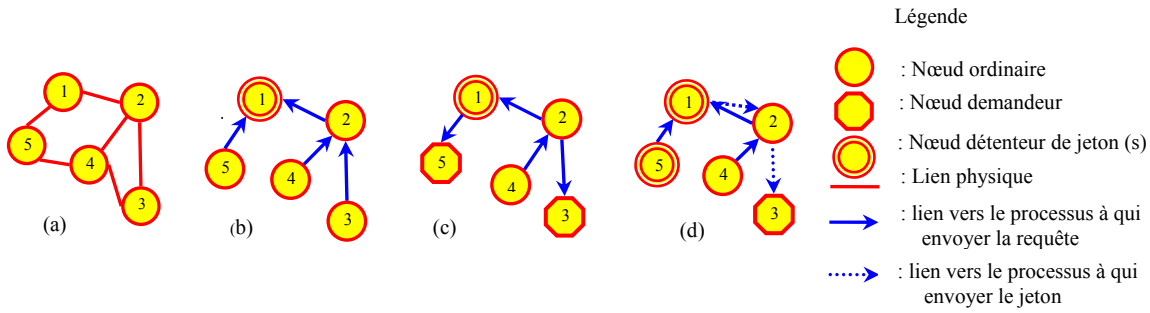


Figure 20: *Un exemple illustrant l'exécution du protocole de k- EM de M. Naimi*

2.3. Solution de P.K. Srimani et R. L.N. Reddy

Dans [127], P. K. Srimani et R. L. N. Reddy ont développé un protocole d'accès multiple et simultané à une SC qui se base sur le protocole d'EM de I. Suzuki et T. Kassami [133] en utilisant *k* jetons (où *1<=k <N, N* étant le nombre de processus du système). Chaque processus entretient un certain nombre d'informations, notamment sur les requêtes reçues (vecteur RN) et servies (vecteur PLN). Chaque jeton transporte des informations sur l'état du système (i.e. une file de requêtes pour le jeton, un vecteur de numéros de séquences des requêtes servies et un vecteur de booléens qui sert à la synchronisation (voir plus loin dans cette section)). Les informations entretenues localement et celles des jetons sont mutuellement mises à jour à différentes occasions (à la sortie de la section critique, à la réception de requête…) par un processus qui détient au moins un jeton. A cause des délais de communication qui sont imprévisibles et l'ordre arbitraire des messages, les informations véhiculées par les jetons sont naturellement différentes. Une bonne partie du protocole est consacrée au maintient à jour des informations entretenues localement en plus de celles des jetons. Comme un processus détenant un jeton n'est pas au courant si un autre processus a envoyé son jeton à un même processus demandeur, les mises à jour décrites ci-dessus

permettent d'éviter d'envoyer ce jeton supplémentaire. Sachant que les numéros de séquences des requêtes sont bornées à la valeur $L>N$, chaque L requêtes d'un nœud i représente un tour. Des opérations de synchronisations sont souvent invoquées pour assurer que les informations sur toutes les requêtes issues d'un processus i durant le tour précédent sont collectées au niveau de chaque jeton.

Quand un processus i désire accéder à sa SC et ne dispose d'aucun jeton (dans le cas contraire, il est libre d'accéder à sa SC), il diffuse sa requête avec le nouveau numéro de séquence à tous les autres processus en initialisant ses variables pour le prochain tour, si nécessaire, et attend la réception d'un jeton (il peut recevoir plusieurs jetons) auquel cas il occupe un jeton et accède à sa SC. A la sortie de la SC, il marque la satisfaction de sa requête dans les structures du (ou des) jeton(s), se synchronise éventuellement avec les autres processus pour un prochain tour concernant i et envoie éventuellement un jeton au prochain processus si la file du jeton n'est pas vide; autrement, il garde le jeton au repos. Quand un processus i reçoit une requête d'un autre j, il marque cette nouvelle requête dans les structures appropriées. Si nécessaire, le processus i mis à jour les structures des jetons pour entamer un nouveau tour pour j et lui envoie un message d'acquittement. Si les conditions sont favorables, un jeton est ensuite transmis à j après avoir mis à jour les structures transportées. Quand un jeton est reçu au niveau de i, celui-ci met à jour les informations concernant les nouveaux tours pour tous les processus concernés y compris lui-même. Sachant qu'un processus peut recevoir plusieurs jetons pour une même requête, il accède à sa SC seulement s'il ne l'a pas déjà fait; autrement, il met à jour les informations du jeton et l'envoie à un prochain demandeur s'il en existe sinon, il est gardé au repos.

Ce protocole requière 0 ou $N+k-1$ messages pour accéder une SC (on supposant que L est une valeur très grande); ce qui améliore presque de moitié les performances du protocole de K. Raymond [109].

2.4. Solution de K. Makki et *al*

Dans [82], K. Makki et *al* proposent un protocole de k- EM orienté jeton. Le jeton circule sur demande à travers les processus et permet d'accéder en SC aux seuls processus le recevant avec un sémaphore non nul ou avec un sémaphore nul. Ce dernier, avant d'accéder en SC, doit attendre de recevoir plus tard un message de libération d'un processus précédent déjà permis d'accéder à sa SC. Le protocole suppose que chaque processus peut communiquer directement avec tout autre processus. Le jeton transporte une file qui contient la liste des processus qui vont recevoir, dans l'ordre d'apparition, le jeton et un sémaphore général. Le dernier processus dans la file du jeton est considéré comme un ''*bon processus*''; les requêtes de SC sont adressées à ce processus. Les requêtes reçues par ce dernier sont placées dans une file locale. Le jeton passe par chacun des processus dans la file qu'il transporte; chaque processus le recevant s'enlève de la file du jeton, décrémente le sémaphore de un et envoie le jeton au prochain processus et accède à sa SC si le sémaphore est positif sinon il garde le jeton localement jusqu'à l'arrivée d'un message de libération auquel cas il fait passer le jeton au prochain et accède à sa SC. Lorsque le jeton arrive au bon processus, celui-ci à la sortie de sa SC, attache sa file locale à la fin de la file du jeton. A ce point, un nouveau bon processus est choisi; l'ancien bon processus envoie ainsi un message d'information sur l'identité du nouveau bon processus à tous les processus qui ne figurent pas dans la nouvelle file du jeton. Un délai fixé d'attente (i.e. temps maximum d'échange de messages entre deux sites) de requêtes retardataires (en transit) est alors observé. Ce délai d'attente permet de s'assurer de la réception des requêtes en transit. (A ce stade, l'ensemble des processus est réparti en deux catégories, ceux ayant une requête pendante (ils ne peuvent envoyer de nouvelles requêtes

qu'à la sortie de leurs SC) et ceux informés de l'identité du nouveau bon processus, leurs prochaines requêtes seront envoyées au nouveau bon processus).Après ce délai, le jeton reprend alors un nouveau tour qui se termine lorsqu'il sera reçu par le dernier processus de sa file. La réception du jeton permet à chaque processus de prendre acte du nouveau bon processus, i.e. le dernier processus de la file du jeton. A la sortie de la SC, un message de libération doit être envoyé au *k* ieme processus dans la file du jeton, c'est le processus qui va recevoir le jeton avec un sémaphore nul! Si le nombre de processus de la file du jeton est inférieur à *k*, le message de libération est adressé au dernier processus de la file. Ce qui permet au bon processus de recevoir un message de libération de chacun des *k* processus précédents dans la file du jeton. Ces messages seront utilisés pour incrémenter le sémaphore à la réception du jeton.

Dans tous les cas, à l'exception des charges de demandes très faibles, le nombre de messages par SC peut être exprimé par une valeur constante qui s'approche de la valeur *trois* pour les hautes charges de demandes.

Ce protocole est simple de conception mais suppose que chaque processus communique directement avec tout autre processus et fait recours à la diffusion quand il s'agit d'informer les processus de l'identité du bon processus.

2.5. Solution de S. Bulgannawar et N.H. Vaidya

Dans [27], S. Bulgannawar et N. H. Vaidya proposent un protocole de *k*- EM qui étend celui d'EM de M. Maimi et M. Trehel [93]. Ce protocole se base sur l'utilisation de *k* jetons et une structure de forêt *dynamique* pour chaque jeton. Le réseau est supposé de topologie complète. Les jetons sont identifiés de *1* à *k*; initialement, le jeton *t* est détenu par le processus *t*. Chaque processus *i* détient un tableau $pointeur_i$, dont l'entrée *t* correspond au père de *i* sur le chemin du jeton *t* dans la forêt (i.e. collection d'arbres) correspondante, où $1<=t<=k$; initialement, $pointeur_i[t]=t$, ce qui signifie que le processus *t* est la racine de l'unique arbre de la forêt *t*. En général, les processus en attente du jeton *t* ou détenant ce jeton sont les racines des arbres de la forêt *t*. Rappelons qu'un processus demandeur de SC doit identifier le jeton souhaité. Chaque processus *i* détient une file d'attente de requêtes dont chaque élément se compose de l'identité d'un processus demandeur de SC et d'un flag utile à l'entretient de la requête. Si *i* est demandeur de jeton, cette file contient les requêtes du même jeton que *i*. Chaque jeton transporte les identités des processus auxquels le jeton doit être envoyé dans l'ordre d'arrivée.

Un nœud *i* désirant accéder à sa SC, y accède directement s'il dispose d'un jeton; autrement, il choisi un jeton *t* selon une heuristique donnée (choix aléatoire, dernier jeton reçu ou le dernier jeton détenu par un autre processus dont il est informé) et envoie une requête pour ce jeton au processus père dans la forêt *t* désigné par $pointeur_i[t]$ et attend la réception d'un jeton *quelconque* pour accèder à sa SC. Cette requête suit le chemin défini par cette forêt et tous les processus rencontrés pointent *i* (qui devient la racine de cette forêt). Un processus *j* recevant la requête de *i* l'enfile localement, s'il est détenteur d'un jeton ou demande le même jeton; autrement, il se contente de la faire suivre à son père sur la forêt. Dans le cas où le processus *j* détient un jeton libre, il le lui envoie immédiatement et le considère comme père dans la forêt du jeton envoyé. La réception d'un jeton au niveau d'un processus (nécessairement demandeur de SC) lui permet de devenir la racine de la forêt de ce jeton et d'accéder à sa SC après quelques mises à jour. Celles-ci concernent la file du jeton à l'aide de la file locale et quelques autres mises à jour (liées essentiellement aux flags des éléments de la file d'attente et au père dans la forêt du jeton attendu) en dépendant de l'identité du jeton reçu relativement à celle du jeton attendu. A la sortie de la SC, si la file du jeton libéré n'est pas

vide, le processus *i* envoie le jeton au premier processus de cette file et met à jour le père dans la forêt de ce jeton en fonction des flags dans la file du jeton ; autrement, il garde le jeton localement mais informe *v* (paramètre défini selon une heuristique donnée) processus (, déterminés en fonction des flags de la file du jeton,) de la détention d'un jeton libre. Ces derniers vont alors pointer ce processus dans la forêt de ce jeton, ce qui leur permet de réduire la distance vers ce jeton pour d'éventuel besoin futur.

Une étude de simulation a été réalisée afin de permettre la comparaison du protocole de [27] aux protocoles de K. Raymond [109], P. K. Srimani et R. L. N. Reddy [127] et Makki et *al* [82]. Les mesures de performances utilisées sont le temps moyen nécessaire pour accéder à une SC, le nombre de messages moyens requis pour accéder à une SC et la taille de l'information transportée par les messages de contrôle. Les résultats de simulations montrent que le protocole de [27] engendre de meilleures performances pour les trois premiers paramètres au prix d'une légère augmentation de la taille des messages transportés.

2.6. Solution de H. Kakugawa et *al*

Dans [67], H. Kakugawa et *al* développent une solution de *k*- EM à permission, qui se base sur l'utilisation du concept de *k*- coterie, qui généralise celui de coterie, pour permettre à *k* processus d'accéder simultanément à leurs SC. La solution proposée [67] est de type M. Maekawa [80] mais avec l'utilisation de *k*- coterie.

Rappelons que le concept de coterie (voir Chapitre II Section 2.3.8) donne une base permettant la gestion d'un exemplaire de ressource de manière exclusive. L'EM est garantie puisqu'il n'existe pas de quorums disjoints dans une coterie. Ainsi, par extrapolation *intuitive*, *k* processus peuvent être dans leurs SC respectives s'il existe *k* quorums *disjoints*. De plus, pour ne pas avoir *k+1* processus simultanément dans leurs SC respectives, on ne doit pas trouver *k+1* quorums distincts. De cette manière, on arrive à la définition du concept de *k*-coterie (définie dans [67]) dont *1*- coterie est une coterie.

Concept de k- coterie

Soient *U* un ensemble de *N* nœuds, *C* un ensemble non vide constitué de plusieurs sous ensembles non vides *Q* appelés quorums,

C est dite une *k*- coterie [100] si et seulement si les conditions suivantes sont vérifiées :

- Non intersection : $\forall h / (1 \leq h \leq k-1)$ et $\forall Q_1, \ldots, Q_h \in C$ tel que $Q_i \cap Q_j = \emptyset$ $(i \neq j)$ pour $1 \leq i,j \leq h$, $\exists Q \in C$ tel que $Q \cap Q_i = \emptyset$ $\forall 1 \leq i \leq h$,
- Intersection : $\forall k+1$ éléments distincts $Q_1, \ldots, Q_{k+1} \in C$, $\exists$ une paire Q_i et Q_j / $Q_i \cap Q_j \neq \emptyset$ $\forall 1 \leq i, j \leq k+1$.
- Minimalité : $\forall Q_i, Q_j$ éléments de C, $Q_i \not\subseteq Q_j$.

Un processus demandeur de SC peut accéder à sa SC en sélectionnant un quorum ''approprié''. A l'aide de la propriété d'intersection, *k* processus peuvent accéder simultanément à leurs SC.

L'algorithme

Dans le protocole de H. Kakugawa et *al* [67], Chaque processus *i* entretient deux ensembles, *Yes* et *NotNow* qui contiennent respectivement l'ensemble des processus qui répondent par *Ok* et ceux qui répondent par *Wait* et un ensemble *Perm* qui contient la requête du processus auquel *i* a donné sa permission. Quand un processus *i* désire accéder à sa SC, il envoie une

requête estampillée à tous les membres de son quorum Q_i dont *i* fait partie. Ensuite, attend la réception de messages *Ok* ou *Wait*. Si *i* reçoit *Ok* de chaque processus de Q_i, il accède à sa SC. Lorsque *i* reçoit un premier message *Wait*, il sélectionne un autre quorum *Q'* (, il serait aberrant d'attendre les messages *Ok* qui peuvent durer longtemps alors qu'il existe *k* quorums distincts ! Il s'agit de la différence essentielle avec M. Meakawa [80],) qui minimise */Q' ∩ Yes/* et qui n'intersecte pas avec *NotNow* s'il existe et envoie une requête à tous les processus de *Q' – Yes*. Si un tel quorum n'existe pas, *i* se contente d'attendre la réception des messages *Ok*. Notons que durant cette opération, *i* peut recevoir un message *Ok* d'un processus *j* de *NotNow*. Dans ce cas, il déplace *j* de *NotNow* vers *Yes* et si un quorum est formé par *Yes*, il accède à sa SC.

Quand *j* reçoit une requête de *i*, il lui retourne *Ok* si *Perm* est vide et affecte *i* à *Perm*. Si *Perm* contient la requête de *s*, la requête de *i* est enfilée ensuite si la requête de *i* est moins prioritaire par rapport à celle de *Perm* et celles de la file locale, un message *Wait* est retourné à *i* sinon un message *Query* est envoyé au processus dont la requête est dans *Perm* et attend un message *Release* ou *Relinquish* de ce processus. Si un message *Query* a été déjà envoyé, alors il n'est pas nécessaire d'envoyer un autre. Si *j* reçoit un message *Relinquish*, la requête du processus dans *Perm* est déplacée vers la file locale et celle de *i* est déplacée de la file vers *Perm*; ensuite, *j* envoie un message *Wait* aux processus de la file locale auxquels il n'a pas envoyé ce message depuis le dernier envoi de *Query* et finalement, un message *Ok* est envoyé à *i*. Lorsqu'un processus *p* reçoit un message *Query* de *q*, s'il n'est pas en SC et *q* est dans *Yes*, alors *p* déplace *q* de *Yes* vers *NotNow* et retourne à *q* un message *Relinquish*. Lorsqu'un processus *i* sort de sa SC, il envoie un message *Release* à tous les processus de *Yes ∪ NotNow*.

Quand *j* reçoit un message Release de *i*, il supprime la requête de *Perm*. Si la file locale n'est pas vide, *j* supprime la requête de *i* de la file s'il y a lieu, déplace la requête la plus prioritaire de cette file vers *Perm* et envoie un message *Ok* au propriétaire de cette requête et un message *Wait* à tous les processus de la file locale auxquels il n'a pas envoyé ce message depuis le dernier envoi de *Query*.

Le protocole engendre *3/Q/* messages dans les meilleurs cas pour accéder à une SC où */Q/* la taille du plus grand quorum construit, puisqu'un processus envoie des requêtes et reçoit des *Ok* et envoie des *Release* de et vers tous les membres de *Q*; étant donné que la taille d'un Quorum est $O(\sqrt{n} \log n)$. Dans le pire cas (i.e. le cas où un processus est amené à envoyer des messages *Query* et à recevoir des messages *Relinquish*), la complexité est de 6/Q/. La construction des quorums est alors d'un lien direct avec cette complexité.

2.7. Autres Solutions de *k*- EM

Afin d'améliorer la complexité en messages, divers autres solutions de *k*- EM basées *k*-coteries existent dans la littérature [100, 57, 34]. Ces solutions admettent un principe similaire à celui de [67] et profitent des avantages des coteries en terme de tolérances aux pannes. Elles différent entre elles par la manière de constructions des structures de *k*- coteries. Ainsi, dans le but d'ajouter de nouvelles propriétés qui augmentent l'efficacité des *k*- coteries, d'autres concepts ont apparus. Citons par exemple, les *k*- coteries non dominées [100], le concept de *k*-majority [34], les structures de cohortes [57], l'arbre de quorums [34], etc... Dans l'ensemble, les performances des protocoles qui utilisent les *k*- coteries dépendent de la taille des quorums construits. La suite de cette section décrit ces nouveaux concepts ainsi que leurs apports.

Concept de k- coterie non dominées

Soient *Q1* et *Q2* deux *k*- coteries sous *U*. *Q1* domine *Q2* si :

1. $Q1 \neq Q2$,
2. $(\forall H \in Q2),\ \exists G \in Q1$ *tel que* $G \subseteq H$.

Une *k*- coterie *Q* sous *U* est dominée s'il existe une autre *k*- coterie sous *U* qui domine *Q*. Ce type de *k*- coterie fournie un haut degré de tolérance aux pannes et par conséquent de *disponibilité*. Dans ce contexte, la disponibilité est mesurée par la probabilité de construction d'un quorum en présence de pannes.

Deux méthodes sont utilisées [100] pour construire les *k*- coteries non dominées:

- Vote pondérée [46]: A chaque processus est associé un nombre de votes spécifique. Un quorum est formé par l'obtention d'au moins la majorité de votes. Une autre manière permettant d'obtenir un quorum est l'utilisation du *consensus majoritaire.* A chaque noeud est associé un seul vote, et un quorum est formé par obtention de la majorité de votes [137].
- Composition [99] : Une *k*- coterie non dominée peut être construite en concaténant plusieurs *k*- coteries non dominées de la manière suivante:
  Soient *k* un entier $>= 2$ et deux entiers positifs k_1 et k_2 / $k=k_1+k_2$. Soient U_1 et U_2 deux ensembles non vides de noeuds/ $U_1 \cap U_2 = \phi$. Admettons que Q_1 est une k_1- coterie non dominée sous U_1 et Q_2 est une k_2- coteries non dominées sous U_2. Soit $U=U_1 \cup U_2$ et $Q=Q_1 \cup Q_2$. Alors, *Q* est une *k*- coterie non dominée sous *U*.

Structures de cohortes

Un Cohorte [57] $Coh(k, n) \equiv (C_1,\ldots,C_n)$ est une liste d'ensembles deux à deux disjoints; chaque ensemble C_i est appelé un cohorte. La structure de Cohortes *Coh(k, n)* doit vérifier les conditions suivantes :

- $/C_1/=k$,
- $\forall i : 1<i<= n : /C_i/>max (2k-2, k)$.

Autrement dit, le premier Cohorte possède *k* membres et les autres Cohortes possèdent plus que *2k-2* membres. Par exemple, *({1, 2}, {3, 4, 5}, {6, 7, 8, 9, 10})* est *Coh(2,3)* puisqu'il possède trois Cohortes disjoints avec deux membres (=*k*) pour le premier et plus de deux (=*2k-2*) pour les autres.

Un ensemble *Q* est dit un *quorum sous Coh(k, n)* si un cohorte C_i dans *Coh(k, n)* est *cohorte primaire* de *Q* et chaque cohorte $C_j, j>i$ est *cohorte supportant Q* où :

- Un cohorte *C* est cohorte primaire de *Q* si $/Q \cap C/=/C/-(k-1)$ (i.e. *Q* contient Tous les éléments de *C* sauf *k-1*) et,
- Un cohorte *C* est cohorte supportant de *Q* si $/Q \cap C/=1$ (i.e. *Q* contient exactement un élément de *C*).

Les quorums sous *Coh(k, n)* sont appelés les *quorums de cohortes* et il peuvent former une *k*-coterie.

Dans [57], un algorithme est défini pour construire efficacement les quorums de cohortes. Ce type de *k*- coterie est résistant au partitionnement et aux pannes de nœuds où il est toujours possible de construire une autre *k*- coterie sous le même cohorte en isolant les nœuds disparus.

Structures logiques

Arbre binaire

Un *arbre binaire* est un ensemble fini de nœuds tel que :

- un nœud spécifique désigné est appelé la racine et est au niveau *0*.
- les autres nœuds sont partitionnés en S_1 et S_2 où chacun de ces ensembles est un arbre binaire. S_1 et S_2 sont appelés les sous arbres de nœuds.

Il existe 2^i nœuds au niveau *i* de l'arbre. Par conséquent, un arbre binaire de niveau $(h+1)$ contient $(2^{h+1} - 1)$ nœuds. Chaque nœud dans l'arbre binaire est numéroté du haut en bas (voir Figure 21)

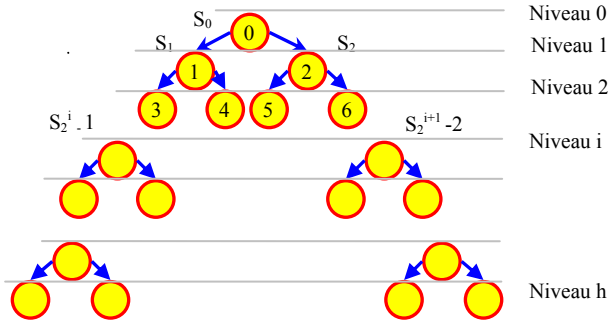


Figure 21: *Un arbre binaire.*

Quorum d'arbre binaire

La stratégie de quorums d'arbre binaire [6] organise logiquement les nœuds dans un système comme une structure binaire de quorums. Un *quorum d'arbre binaire* pour *1*-exclusion mutuelle consiste (récursivement) en :

- la racine et un quorum d'arbre binaire du sous arbre gauche, ou
- la racine et un quorum d'arbre binaire du sous arbre droit, ou
- un quorum d'arbre binaire du sous arbre gauche et un quorum d'arbre binaire du sous arbre droit.

Exemple: Pour un arbre binaire à *4* niveaux, des exemples de quorums sont : *{0, 1, 3, 7}, {0, 1, 4, 10}, {0, 7, 8, 9, 10}, {0, 11, 12, 13, 14}, {7, 8, 9, 10, 11, 12, 13, 14}*.

Quorums d'arbre binaire généralisé [34]

Soit un arbre binaire de niveau $(h+1)$ constitué d'un ensemble de *n* nœuds organisés par niveaux où $h>=0$ et $n=2^{h+1}-1$. Un ensemble *Q* est dit *quorum d'arbre binaire généralisé* pour la *k*- exclusion mutuelle si la condition suivante est vérifiée: Pour un niveau *i*, tel que $2^i <= k < 2^{i+1}$, $0<=i<=h$, *Q* contient un quorum d'arbre binaire pour la *1*- exclusion mutuelle pour chacun des sous arbres $S_{k-1}, S_k, \ldots, S_{2k-2}$.

L'ensemble de tous les quorums *Q* obtenus forme une *k*- coterie et est appelée une *k-coterie d'arbre binaire généralisé*. Plus précisément, cette *k*- coterie est une *k*- coterie non dominée.

Exemple: Pour un arbre binaire à *4* niveaux, des exemples de quorums d'arbre binaire généralisé pour la *3*- exclusion mutuelle sont ceux de la *1*- exclusion mutuelle des sous arbres S_2, S_3 et S_4 dont certains sont : *{3, 7},{1, 3, 7},{2, 5, 11}*.

Quorums d'arbre binaire étendu [34]

Il est à noter que dans le quorum d'arbre binaire généralisé, certains nœuds (i.e. les nœuds *j* tel que $j<=k-2$) ne sont pas utilisés dans la construction des quorums de la *k*- coterie. Cependant, il est intéressant de faire participer tous les nœuds pour augmenter la disponibilité du système. D'où l'introduction du *quorum d'arbre binaire étendu* qui est en réalité une extension du quorum d'arbre binaire généralisé par l'introduction de nouveaux nœuds fictifs. Dans l'arbre étendu, il existe alors deux types de nœuds: Nœuds *fictifs* et nœuds *réels*. La structure d'un arbre binaire étendu de niveau $(h+1)$ est la même que celle d'un arbre binaire généralisé sauf que les nœuds *0, 1, ... (k-2)* sont fictifs, où $2^i <=k<2^{i+1}$ et $0<=i<=h$ (voir Figure 22).

Soit un arbre binaire étendu de niveau $(h+1)$ constitué d'un ensemble de n' nœuds organisés par niveaux où $h>=0$ et $n'=2^{h+1}-1= n+ (k-1)$, *où* $k>=1$. Un ensemble *Q* est dit *quorum d'arbre binaire étendu* pour la *k*- exclusion mutuelle si la condition suivante est vérifiée: Pour un niveau *i*, tel que $k= n'-n+1$, $2^i <=k<2^{i+1}$, $0<=i<=h$, *Q* contient un quorum d'arbre binaire pour la *1*- exclusion mutuelle pour chacun des sous arbres S_{k-1}, S_k,… , S_{2k-2}.
L'ensemble des tous les quorums *Q* obtenus forme une *k*- coterie. Plus précisément, cette *k*-coterie est une *k*- coterie non dominée.

Exemple: Pour un arbre binaire à *4*, des exemples de quorums d'arbre binaire généralisé pour la *4*- exclusion mutuelle sont ceux de la *1*- exclusion mutuelle des sous arbres S_2, S_3 et S_4 dont certains sont : *{3, 7}, { 9, 10},{5, 11}, {13, 14}*.

La stratégie des quorums de l'arbre binaire étendu engendre une haute disponibilité par rapport aux stratégies de *k*- majority, de cohortes et de DIV dans la plupart du temps [34].

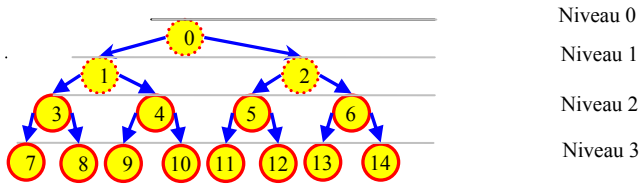


Figure 22: *Un arbre binaire étendu.*

Approche	Outils &mécanismes		Autres	Protocoles	Objectif	Messages	Observations
Jeton	Demande	Diffusion totale	*k* jetons	Srimani-Reddy [127], 1992	Améliore Raymond [109]	*0 ou N+k-1(si L est très grand)*	-Se base sur Suzuki-Kazami [133]. -Topologie complète.
		Structure logique: graphe orienté	*k* jetons	Naimi [94], 1996		*0 à 2(N-1)*	- Se base sur Naimi-Trehel [93]. - Topologie connexe.
		Structure logique: ensemble de forêts	*k* jetons	Bulgannawar –Vaidya [27], 1995	Généralise le protocole d'EM de Naimi-Trehel [93]	Résultats par simulation	-Topologie complète. - Comparé avec Raymond [109], Srimani-Reddy [127] et Makki et *al* [82]. IL engendre de meilleurs résultats.
		Pas de structure logique	-*1* jeton -Recours à la diffusion partielle	Makki et *al* [82], 1992	-	N/m+2 où m est le nombre de requêtes par *cycle* (voir [82])	- Tous les processus sont accessibles directement.
Permission	Individuelle		Diffusion totale	Raymond [109], 1989	Adapte Ricart-Agrawala [117]	*>=(2N - k -1)* et *<= 2(N-1)*	-Topologie complète -Estampillage de Lamport
	Arbitre		-*k*- coterie -Diffusion partielle	Kakugawa et *al* [67], 1993	-	>= 3/Q/ et <= 6/Q/ où Q est le plus grand Quorum	- Topologie complète
			-k- coterie non dominées (vote pondéré, composition) -diffusion partielle	Neilsen-Mizuno [100], 1994	-		- Tolérance aux fautes
			- Cohorts - diffusion partielle	Jiang et al [57], 1997	-	Complexité est proportionnelle à la taille d'un quorum qui est 2 (si k=1) ou entre k et O(n) (si k>1)	- Tolérance aux fautes de nœuds et aux partitionnements
			-k- coterie d'arbre binaire étendu - diffusion partielle	Chang-Chen [34], 1997	-	Complexité est proportionnelle à la taille d'un quorum qui est entre $[\log_2(n/k)]$ et $[(n+k)/2k]$	Tolérance aux fautes de nœuds entre $k(\log_2 (n/k) - 1$ et $n - k\log_2(n/k)$ nœuds en pannes - Meilleurs résultats que Cohorts, k-majority et DIV.

Table 4: *Classement des solutions classiques de k- EM distribuée.*

3. La *k*- EM dans les réseaux mobiles ad hoc

Dans cette section, nous décrivons le seul protocole de *k*- EM connu dans les réseaux mobiles ad hoc.

3.1. Solution de Walter et *al*

Dans [147], J.E. Walter et *al* ont proposé un protocole de *k*- exclusion mutuelle orienté jeton pour les réseaux mobiles ad hoc. Ce protocole nommé *KRL* est une généralisation du protocole d'exclusion mutuelle *RL* [150] (voir Chapitre II Section 3.3); ce qui lui permet d'hériter de ses principaux mécanismes. *KRL* utilise une structure de *DAG* de pointeurs orientés jetons mappées sur les liens physiques. Chaque nœud du réseau entretient seulement des informations liées à son voisinage immédiat. Chaque nœud détient une file d'attente des identités des nœuds voisins desquels des requêtes pour un jeton ont été reçues. *KRL* maintiens *k* jetons, initialement répartis entre les nœuds *0..k-*. Si *k=1*, le nœud de plus basse élévation (i.e. nœud puits) est constamment le détenteur du jeton. Si *k>1*, tous les nœuds détenant au moins un jeton sont constamment des noeuds puits. Le protocole *KRL* assure que tous les nœuds qui ne détiennent pas de jeton possèdent constamment un chemin à un nœud détenteur de jeton(s). Autrement dit, chaque nœud non détenteur de jeton doit assurer continuellement l'existence d'au moins un voisin avec une élévation moindre puisque sa requête doit être envoyée à travers ce nœud (qui forme son lien sortant vers le détenteur d'un jeton). Dans le cas où ce nœud se trouve sans ce genre de voisin, il invoque la technique d'inversions de liens afin de lui assurer l'existence de nœuds sortants. Chaque nœud choisi dynamiquement son voisin de plus petite élévation comme nœud prochain préféré sur le chemin du détenteur d'un jeton pour lui envoyer sa requête. Toutes les requêtes qui atteignent un nœud détenteur de jeton sont traitées d'une manière symétrique et servies continuellement. La structure de *DAG* est réorganisée quand il est nécessaire (i.e. lors de déplacement de jeton ou lors de rupture de liens physiques) et les requêtes bloquées sont re-routées. Etant donné qu'un nœud détenteur d'au moins un jeton doit avoir constamment au moins un voisin avec lien entrant (i.e. avec une élévation plus grande), s'il se trouve sans un tel nœud, il invoque la technique d'inversements de liens. Une requête qui arrive au niveau d'un nœud en SC est immédiatement servie si ce noeud détient plus d'un jeton, ce qui permet d'augmenter la concurrence d'accès en SC.

Le protocole suppose que les liens sont bidirectionnels et FIFO, les nœuds détenant au moins un jeton ne tombent pas en panne. Le partitionnement peut se produire mais chaque partition du réseau ainsi formée doit contenir au moins un jeton pour continuer à fonctionner.

La Figure 23 montre un exemple d'exécution du protocole *KRL* dans un réseau statique en utilisant *2* jetons. Dans la Figure 23(a), les nœuds *2*, *3* et *4* possèdent des requêtes en cours, les files Q_2, Q_3 et Q_4 contiennent leurs requêtes respectives et les nœuds *2* et *3* ont envoyés leurs requêtes au nœud *0* qui enregistre leurs requêtes dans leur ordre d'arrivée. Le nœud *4* envoie sa requête au nœud de plus basse élévation i.e. le nœud *1*. La Figure 23(b) montre l'état du système après que le nœud envoie le jeton au nœud *4* et a demandé le jeton au nœud *1* (sa requête est enfilée dans Q_1 et Q_4). Le nœud *0* a aussi envoyé le jeton au nœud *3* et le suit d'une requête pour le compte du nœud *2* (la requête de *0* est enfilée dans Q_3). Remarquons que la direction des liens logiques, entre le nœud *0* et *3* et entre le nœud *1* et *4*, a été inversé. De même que les nœuds prochains sur le chemin du jeton du nœud *0* et *1* qui deviennent respectivement le nœud *3* et *4* et ceux des nœuds *3* et *4* qui deviennent eux-mêmes). Ainsi, les élévations des nœuds *3* et *4* ont été diminuées pour devenir les nouveaux nœuds puits. Dans la

Figure 23(c), le nœud *4* a quitté sa SC et a envoyé le jeton au nœud *1*. De même, le nœud *3* sort de sa SC et envoie le jeton au nœud *0* qui le transmis au nœud *2*. A ce niveau aucune requête n'est pendante. Dans la Figure 23(d), le nœud *4* a exprimé une requête et l'envoie à son voisin de plus basse élévation, le nœud *2*. Dans la Figure 23(e), le nœud *4* reçoit le jeton du nœud *2*, diminue son élévation et accède à sa SC. Le nœud *1* a reçu un message d'information du nœud *4* et se rend compte qu'il a perdu son dernier lien entrant. Dans la Figure 23(f), le nœud *1* diminue son élévation pour devenir la plus petite que celle de tous ses voisins; ce qui lui permet d'être atteignable par de futures requêtes.

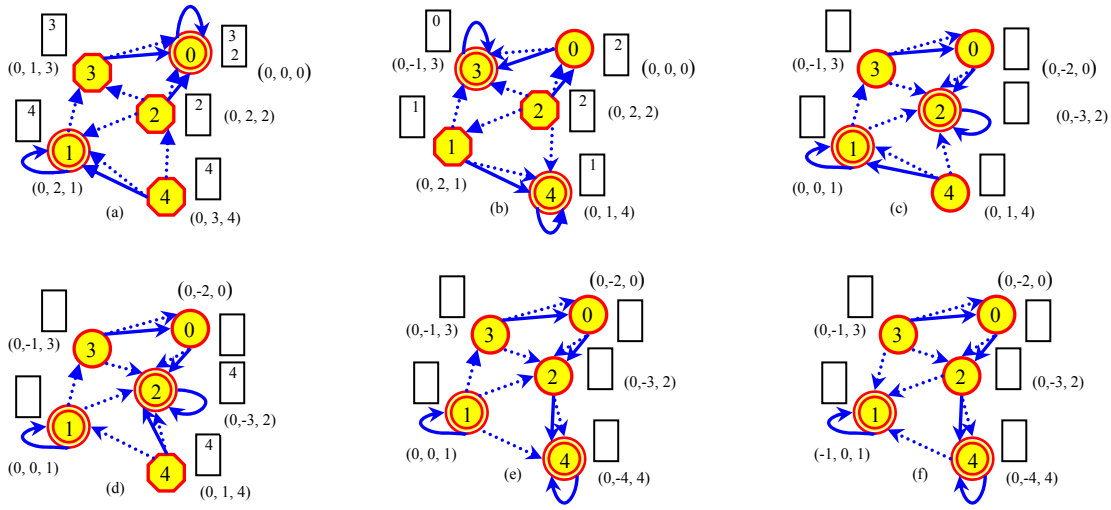
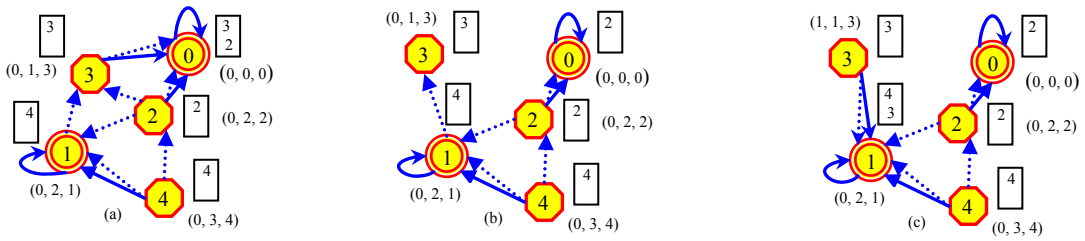


Figure 23: *Exemple de déroulement de KRL dans un réseau statique.*

La Figure 24 illustre le fonctionnement de *KRL* dans un réseau dynamique en utilisant *2* jetons. La Figure 24(a) donne le même schéma d'exécution que celui de la Figure 23(a) mais avec augmentation du temps entre la Figure 24(a) et la Figure 24(c). Dans la Figure 24(b), le nœud *3* s'est déplacé en causant la rupture de liens avec les nœuds *0* et *2* et dont la conséquence le graphe qui n'est plus orienté jeton puisque le nœud *3* n'a plus de liens sortants. Le nœud *0*, seulement, supprime la requête du nœud *3* de sa file puisqu'il a toujours des liens entrants. Le nœud *2* n'a pas de mise à jour à réaliser puisque son prochain sur le chemin du jeton est toujours présent. Dans la Figure 24(c), le nœud *3* s'est adapté au changement du lien avec le nœud *0* en s'orientant vers le nœud privilégié *0* en lui adressant sa requête. Les Figures 24(d) et 24(e) montrent respectivement l'état du système après l'envoi d'un jeton par le nœud *0* au nœud *2* et après l'envoi d'un jeton par le nœud *4* au nœud *1* qui l'a transmis au nœud *3*.



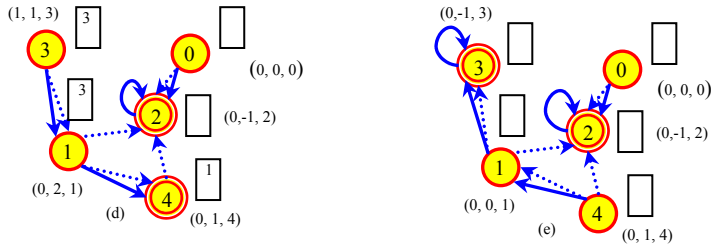


Figure 24: *Exemple d'exécution de KRL dans un réseau dynamique.*

Le protocole *KRL* présente un problème de sous utilisation de jetons. En effet, dans certaines situations la file d'attente d'un nœud détenteur d'un jeton peut contenir plusieurs requêtes en attente alors qu'un autre nœud détient des jetons libres. Ce problème est du au fait qu'un nœud non détenteur de jeton n'a aucune information ni sur le nombre de jetons détenus par les nœuds privilégiés ni sur les requêtes enfilées par ces derniers. De ce fait, une nouvelle variante du protocole *KRLF* (*K* Reverse Link with token Forwarding) [146] a été proposée. Elle permet de faire circuler les jetons libres. Un nœud sortant de sa SC et ayant une file d'attente libre, envoie le jeton rendu libre à l'un de ses voisins afin de lui permettre d'être éventuellement utilisé. La Figure 25 illustre ce problème de sous- utilisation de jetons. Dans cette figure, les noeuds *4* et *6* sont demandeurs de SC. Ils ont adressé leurs requêtes au nœud *5* qui les enfile localement dans l'ordre indiqué. A sa sortie de SC, le nœud *5* envoie le jeton au nœud *6* et le nœud *4* reste en attente. Parallèlement, le nœud *3* détient un jeton libre puisque sa file est vide!

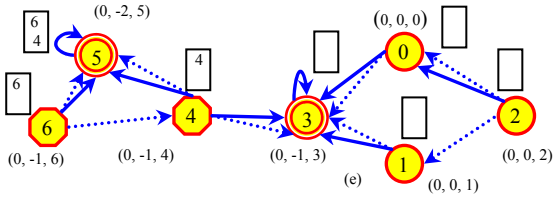


Figure 25: *Problème de sous- utilisation de jetons dans KRL.*

La technique de renvoi du jeton introduite dans la variante *KRLF* essaye d'envoyer le jeton libre à une autre partie du réseau dans le cas où aucun nœud ne l'a demandé. La stratégie consiste à choisir le nœud voisin de plus petite élévation, ce qui résulte en un minimum d'inversions de liens logiques. Pour éviter d'envoyer le jeton à un nœud qui vient de renvoyer un autre pour la même raison, celui-ci doit être marqué comme *''visité''* et par lui-même et par le nœud qui le lui a envoyé.

L'exemple de la Figure 26 illustre cette modification. Dans la Figure 26(a), le nœud 3 est en SC. Dans la Figure 26(b), le nœud 3 sort de sa SC mais sa file est vide. Il est alors contraint de l'envoyer à un voisin. Ce jeton va traverser les nœuds 0, 2, 1 et 3. Arrivant au nœud 3, celui-ci va marquer tous ses voisins non visités et reprend un autre renvoi.

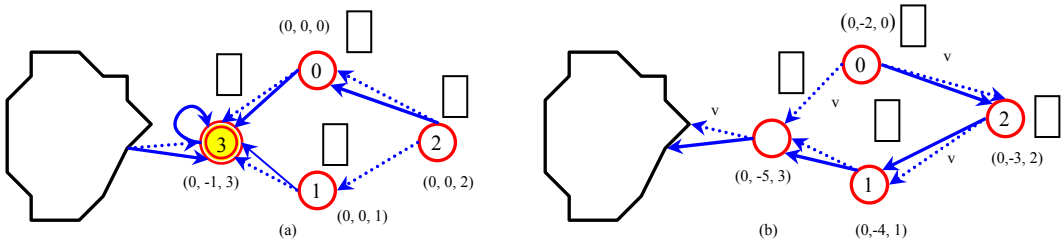


Figure 26: *Exemple de renvois de jetons libres dans KRLF.*

Discussion

Le protocole *KRL* de Walter et *al* [147] est convenable aux réseaux mobiles ad hoc pour les mêmes raisons discutées au Chapitre II, Section 3.3 concernant le protocole *RL* [150]. Néanmoins, le protocole défini dans [147] suppose d'une part, un nombre de changements de liens réduit et d'autre part, ne tolère pas la panne de nœuds détenant au moins un jeton (i.e. non perte de jetons) ni la déconnexion de nœuds. Lors du partitionnement, le protocole ne régénère pas les jetons manquants; de plus la fusion n'est pas considérée. Le nombre de nœuds du réseau est supposé alors fixe et connu de tous les nœuds du réseau. Néanmoins, la simulation effectuée montre une amélioration significative du temps d'attente par SC (sans messages supplémentaires que le protocole de base) lors de l'utilisation de la technique de renvois de jetons. Cependant, un problème majeur apparaît: Il s'agit de la circulation inutile de jetons lors de charges de requêtes très faibles.

4. Conclusion

Ce chapitre fait un tour d'horizon sur les solutions de *k*- EM connue dans la littérature des environnements distribués statiques et mobiles ad hoc. Ces solutions aussi variées sont en général des adaptations ou des généralisations des solutions développées pour l'exclusion mutuelle. La plus part des solutions développées pour les systèmes distribués se basent sur l'utilisation de structures logiques et posent des problèmes de scalabilité. De plus, elles supposent en majorité (à l'exception de celle de M. Naimi [94]) que chaque processus peut communiquer directement avec tout autre processus. Ce qui sous entend l'utilisation de la couche de routage pour l'acheminement des messages. Les solutions orientées quorums admettent un principe similaire mais différent sur la méthode de construction des quorums; elles profitent des propriétés des quorums pour la résistance aux pannes mais leurs performances dépendent de la taille des quorums construits. Il est délicat de penser à utiliser ce concept dans les réseaux mobiles ad hoc en regardant sa dynamicité. La seule solution connue des réseaux mobiles da hoc est due à walter et *al*. Elle définie des mécanismes adéquats aux réseaux mobiles ad hoc mais ne tient pas compte de certaines caractéristiques de ces réseaux et présente un problème de scalabilité.

Dans le Chapitre V, nous proposons une solution au problème de *k*- EM qui adopte une approche complètement différente de celle du protocole dans [147].

Chapitre V

Un protocole de *k*- exclusion mutuelle pour les réseaux mobiles ad hoc

1. Introduction

L'exclusion mutuelle est un service utile dans le cas d'une ressource unique et à accès exclusif. Cependant une ressource exclusive peut exister en plusieurs exemplaires, il s'agit donc de permettre l'utilisation de ces ressources exclusives de manière parallèle mais chacune par un processus différent. Il s'agit alors du problème de la *k*- exclusion mutuelle, où *k* étant le nombre d'exemplaires de cette ressource. Plusieurs protocoles de la *k*- exclusion mutuelle ont été proposés dans la littérature des systèmes distribués selon les deux approches classiques. Quelques solutions sont orientées jeton [27, 82, 94, 127] avec soit *un* soit *k* jetons. Ces solutions, en général, adaptent ou généralisent certaines solutions classiques d'EM [117, 133, 93]. Parmi les solutions à permission, on distingue la permission individuelle; il s'agit de la solution développé par K. Raymond [109] qui étend celle de G. Ricart and A. K. Agrawala [117] afin de délivrer *k* autorisations d'entrées en SC. Les autres solutions se basent sur le principe d'arbitre [34, 57, 67, 100] moyennant l'utilisation du concept commun et de base de *k*- coterie, extension du concept de coterie. Toutes ces solutions admettent un principe similaire mais diffèrent sur la méthode de construction des quorums (i.e. les *k*- coteries non dominées [100], le concept de *k*- majority [34], les structures de cohorts [57], arbre de quorums [34] etc..). Ces solutions profitent des propriétés de tolérances aux pannes et de disponibilité des quorums. Un seul travail est connu dans les réseaux mobiles ad hoc, le protocole *KRL* de J. Walter et *al* [147]. *KRL* est une généralisation du protocole d'exclusion mutuelle *RL* [150] (voir Chapitre II Section 3.3) avec *k* jetons; ce qui lui permet d'hériter de ses principaux mécanismes. *KRL* utilise une structure de DAG de pointeurs orientés jetons mappées sur les liens physiques. Tous les nœuds détenteurs de jeton sont des nœuds puits. Chaque nœud du réseau entretient seulement des informations liées à son voisinage immédiat. Le protocole *KRL* [147] est convenable aux réseaux mobiles ad hoc pour les mêmes raisons discutées au Chapitre II, Section 3.3 concernant le protocole *RL* [150]. Néanmoins, *KRL*

suppose d'une part, un nombre de changements de liens réduit et d'autre part, ne tolère pas la panne de nœuds détenant au moins un jeton (i.e. non perte de jetons) ni la déconnexion de nœuds. Lors du partitionnement, le protocole ne régénère pas les jetons manquants; de plus la fusion n'est pas considérée. Le nombre de nœuds du réseau est supposé fixe et connu de tous les nœuds du réseau.

Le protocole décrit dans ce chapitre adopte une méthode différente de celle de J. Walter et *al* [150]. En effet, aucune structure logique n'est imposée sur le réseau et la couche de routage n'est pas sollicitée pour l'acheminement des messages. Le nombre de nœuds du réseau est variable. Les k jetons ($k > 0$) définis dans le système sont détenus initialement par un seul nœud (le nœud d'identité *0*). Le protocole est orienté demande de jeton. Les requêtes suivent les traces des jetons et sont acheminées au mieux de manière linéaire mais en générale de la même manière qu'une exploration en profondeur d'une arborescence. La trace de chaque requête est sauvegardée de manière distribuée afin de pouvoir reconstituer son chemin qui va servir à véhiculer le jeton dans le sens inverse. Seuls les nœuds en SC ou demandeurs d'accès sont habilités à construire une file d'attente (ceux-la sont considérés comme des futurs détenteurs de jetons libres). Afin d'augmenter la disponibilité du service de *k*- EM, différents types de jetons (*single*, *collector* et *cession*) sont utilisés. Afin de pallier aux effets de l'absence d'une structure logique entretenue, un certain nombre de concepts et outils sont utilisés. En particulier, des heuristiques liées au cheminements de requêtes, à la gestion des files d'attentes et des jetons.

Ce chapitre est structuré comme suit : La Section 2 décrit les idées sur lesquelles repose le protocole et les structures de données et de communication utilisées. Ensuite, la Section 3 décrit le fonctionnement du protocole à travers les différents événements qui le composent. Un simple exemple d'exécution suivi d'une discussion sur le protocole est donné dans la Section 4. La Section 5 propose une extension du protocole pour supporter le partitionnement et la panne de nœuds. La Section 6 décrit une étude de correction du protocole. La Section 7 traite des différents résultats de simulation obtenus ainsi que leurs interprétations. Finalement, la Section 8 conclue le chapitre.

2. Eléments généraux du protocole

2.1. Modèle du réseau

Le service de *k*- exclusion mutuelle fonctionne sur un réseau composé d'un nombre variable de nœuds mobiles, chaque nœud est identifié de manière unique. Les mouvements des nœuds sont libres (i.e. mobilité aléatoire) mais ne tombent pas en panne. Les nœuds utilisent un support de communication sans fil, les liens de communications sont bidirectionnels, Fifo et fiables. Les nœuds du réseau forment en général une topologie connexe en tolérant des partitionnements et isolement brefs. La couche réseau fournie à tout moment aux noeuds les informations sur la formation et la rupture de liens. Un nœud ne peut pas lancer de nouvelle requête que s'il n'a pas de requête en cours. Chaque nœud ne connaît que ses voisins directs et ne connaît pas la taille du réseau. Dans la suite de l'article et afin de décrire notre protocole, nous nous plaçons au niveau du nœud désigné par i.

2.2. Idées de base

Le protocole décrit dans ce chapitre tente de contracter un comportement adaptatif en interagissant avec le comportement dynamique du réseau. Ce comportement ne peut être

assuré que par la disponibilité d'informations sur l'état du système; la récolte de ces informations est alors nécessaire. Afin d'éviter la surcharge du service de *k*- EM par des messages propres à la récolte d'informations nécessaires, il s'avère utile d'utiliser les messages de service pour le transport de ces informations. En effet, étant donné le changement fréquent de la topologie du réseau, les messages de services sont amenés à traverser plusieurs nœuds avant d'arriver à leurs destinations; ce qui assure ce transport. Les informations ainsi rendues disponibles ne sont jamais complètes. De ce fait, le recours à des *heuristiques* dans certains cas de décisions s'avère intéressant. Les heuristiques sont alors utilisées pour aider à choisir les bonnes décisions mais sans les garantir. Les décisions sont fondées sur des suppositions de ce qui *semble être meilleur*. Ainsi, le protocole peut être *paramétré* dès que l'on défini des fonctions pouvant délivrer un résultat (une décision) en faisant abstraction de leurs contenus qui peuvent être modélisés de plusieurs manières différentes.

Sur la base de décisions locales en fonctions des informations rendues disponibles, plusieurs *outils* et *mécanismes* ont été développés pour réaliser le service de *k*- EM.

2.2.1. Acheminement d'une requête

Un nœud désirant accéder à sa SC est amené à émettre une requête. Le problème de base est comment trouver un nœud détenteur de jeton en l'absence de structure logique qui mène vers un jeton. Afin d'éviter de faire recours à la diffusion, nous avons préféré de faire suivre la requête sur les traces d'un jeton. De ce fait, il est donc nécessaire de conserver localement l'historique des envois de jetons. Ceci pour répondre à l'heuristique suivante : *Le nœud voisin qui conduira la requête vers un nœud détenteur de jeton(s) le plus sûrement est celui qui vient de recevoir un jeton le plus récemment*. Une autre question se pose : Lors de l'acheminement de la requête, lequel des nœuds est autorisé à garder la requête localement? L'idéal est d'atteindre directement un détenteur de jeton, mais pour des raisons d'optimisation, la requête séjourne chez le *premier* nœud rencontré sur le chemin du jeton qui est, soit demandeur de SC, soit en SC. Autrement dit, *seuls les nœuds demandeurs de jeton ou détenteurs de jeton sont habilités à entretenir une file d'attente*. Un nœud demandeur est considéré comme un futur détenteur de jeton. La conséquence en est la non participation au service de *k*- EM des nœuds non intéressés par la SC. Une requête d'un nœud séjourne chez un seul autre nœud.

Lors de la réception d'une requête par un nœud *i*, un certain nombre d'actions sont envisageables selon les informations détenues ou récoltées. Si on s'intéresse aux informations locales à *i*: En supposant que *i* est demandeur de SC et a effectué un nombre *faible* de *relances* (voir Section 2.2.3), il peut enfiler localement cette requête sans condition. Si le nombre de relances dépasse un certain seuil et si on admet que les chances d'acquérir un jeton deviennent minimes, il serait préférable d'acheminer la requête pour ne pas la compromettre. Si on s'intéresse à l'état de la file locale, on peut définir un seuil du nombre de requêtes à pouvoir insérer dans la file. Par exemple, on enfile une seule requête et les autres doivent être réacheminées. L'avantage en est de rendre les requêtes réacheminées plus actives; car au lieu de rester en attente dans la file au risque d'être purgées (voir Section 2.2.6) au bout d'un délai préfixé, elles vont à la recherche d'un autre jeton (moyennant un coût en terme de nombre de massages générés au profit de la vivacité). Par conséquent, le nombre de requêtes à garder dans la file locale peu être variable ou fixe. D'autre part, tous les jetons du système ont la même chance d'être sollicités.

Si l'on s'intéresse aux informations collectées, on peut agir comme suit : Si le nombre de sauts effectués par la requête jusque là est supérieur à un pourcentage fixe (par exemple *50%* ou *30%*) du nombre de messages moyen par entrée en SC (ce nombre peut être fixe ou calculé dynamiquement par le système), la requête doit être insérée dans la file d'attente sinon

réacheminée. Aussi, si le nombre de nœuds en SC ou demandeurs d'accès ayant refusé d'insérer une requête dans la file d'attente atteint un certain seuil (déterminé de manière dynamique en fonction du nombre de sauts effectués par la requête jusque là et du nombre total de jetons dans le système), celle-ci doit être enfilée.

2.2.2. Collecte d'informations

La collecte d'informations fiables est d'une nécessité évidente pour la prise de décisions locales. Les messages de services de *k*- EM représente un bon véhicule de ces informations qui sont souvent récoltées de manière distribuée. Plusieurs types d'informations doivent être rendues disponibles pour une meilleure prise de décisions. Si on s'intéresse par exemple au sort d'une requête qui vient d'être reçue, cela dépendra des informations disponibles localement. Par exemple, le nombre de nœuds traversés par la requête peut renseigner sur l'ancienneté de la requête et sur le nombre de messages générés par celle-ci jusque là, le nombre de fois que l'enfilement d'une requête est refusé. Plusieurs autres types d'informations peuvent être utiles. Par exemple, le nombre de fois qu'un jeton est exploité avant d'atteindre son destinataire (voir Section 2.2.5) sert à éviter la famine au nœud destinataire.

Toutes ces informations sont collectées sans *aucune investigation* particulière. En outre, ces informations ne nécessitent pas d'être conservées, la plus part peuvent être *aussitôt obtenues aussitôt exploitées*. Dans la suite de ce chapitre, les informations utiles seront explicitées dans leurs contextes.

2.2.3. Les relances

Dans certaines situations extrêmes, un nœud ayant émis une requête peut attendre longtemps l'arrivée d'un jeton sans le recevoir. Ceci peut se produire si la requête n'aboutit pas ou si le jeton perd le chemin vers la requête. Les raisons possibles sont : L'isolement de nœud ou le partitionnement temporaires, la file où la requête est insérée contient plusieurs requêtes et sa satisfaction dure longtemps par manque de jetons, changements fréquents des liens dans le réseau dus à la mobilité.

Dans le but d'éviter la famine à ce nœud, la requête est relancée au bout d'un temps borné. Cette relance est considérée comme une nouvelle tentative d'une même requête. Autrement dit, elle est générée avec la *même* estampille afin de lui assurer son ancienneté dans le système. Le nombre de relances ne doit pas être illimité. En effet, au-delà d'un certain seuil, un *échec* est observé.

2.2.4. Chemin d'une requête

Dans ce cadre, on fait appel aux nombres premiers vu leur propriété intéressante suivante : *La décomposition d'un nombre entier en produits de nombres premiers est unique*. De ce fait, à chaque nœud i est affecté un attribut A_i représentant un nombre premier *unique* (de numéro $(i+2)$ dans l'ordre croissant des nombres premiers). Le produit T des attributs de tous les nœuds traversés par une requête et calculé de manière distribuée admet une décomposition unique en produit de nombres premiers. Ceci permet d'avoir la liste de tous les nœuds ayant participé à l'acheminement de la requête. Ainsi, grâce à cette valeur, le nœud qui doit acheminer la requête choisit un voisin dont l'attribut ne divise pas T afin d'éviter à la requête de revenir sur un chemin déjà visité (chemin acyclique). Si tous les voisins sont explorés, le nœud i retourne la requête à son prédécesseur sur le chemin de cette requête (la requête rebrousse chemin). Avec cette technique, la recherche d'un jeton se fait de la même manière que l'exploration d'un arbre. Cependant, T ne décrit pas l'ordre des nœuds suivis par la

requête. Le fait que chaque nœud garde trace de son prédécesseur sur le chemin de la requête remédie à ce problème. Ceci permet de tracer de manière distribuée le chemin du jeton qui est dans ce cas l'inverse de celui de la requête. Néanmoins, il est possible d'optimiser ce chemin car un nœud qui doit acheminer le jeton peut avoir plusieurs voisins ayant acheminé la même requête. Dans ce cas, *le voisin qui acheminera le jeton de la manière la plus optimale est celui dont la position sur le chemin de la requête est la plus petite* ; d'où l'intérêt d'introduire l'information liée à cette position. En effet, chaque nœud est appelé à conserver sa propre position sur le chemin des requêtes qu'il a transité mais aussi celle de tous ses voisins relativement aux requêtes communes (voir plus loin).

2.2.5. Types de jetons

Le service de *k*- exclusion mutuelle entretient *k* jetons initialement détenus par le nœud d'identité *0*. Les jetons sont de trois types: Le type S*ingle*, le type C*ollector* et le type *Cession*. Un jeton de type S*ingle* est un jeton ordinaire destiné à satisfaire une seule requête en attente. Il 'voyage' seul sur le réseau et n'est pas appelé à revenir au nœud source. Un jeton *Singl*e peut être exploité en cours de route par des nœuds intermédiaires demandeurs de SC. Afin d'éviter de léser le nœud destinataire de ce jeton, un seuil d'utilisation est défini. Dans certaines situations, principalement lorsqu'un nœud sort de sa SC en ayant une file d'attente contenant plusieurs requêtes, celui-ci peut envoyer son jeton à l'élément du sommet de la file en demandant son retour afin de satisfaire les requêtes en attente. Il peut aussi attribuer à ce jeton une qualification qui lui accorde le droit de collecter tous les jetons stationnaires et oisifs rencontrés au cours de son parcours *aller* et *retour*. Ainsi plusieurs jetons peuvent être véhiculés par un même message; d'où le type de jeton *Collector*.

Un jeton *Collector* permet au nœud source, non seulement de satisfaire un grand nombre de requêtes dans sa file en fonction du nombre de jetons collectés mais aussi, de faire circuler les jetons oisifs vers des 'endroits' où ils sont les plus sollicités. Si le nombre de jetons collectés est plus grand que le nombre de jetons nécessaires au nœud source, le jeton *Collector* peut *céder* les jetons excédants de sa collection, *mais un à la fois* au profit de nœuds intermédiaires demandeurs de SC.

Le troisième type de message de jeton est un message généré par un nœud qui 'craint' son isolement dans un futur proche en ayant à sa possession des jetons libres. Ce nœud, se trouvant lié au réseau par un seul lien, envoie ses jetons (jetons de *Cession*) vers son unique voisin afin de permettre, non seulement de les préserver d'un isolement même momentané mais aussi, leur circulation. Comme le message de jeton *Collecto*r, un message de jeton de *Cession* peut transporter plusieurs jetons simultanément.

2.2.6. Les files d'attente

Rappelons que seuls les nœuds demandeurs de SC ou en SC sont habilités à construire une file d'attente et sont donc des cibles pour les requêtes: Les nœuds demandeurs de SC sont probablement des futurs acquéreurs de jetons. Ceux qui sont en SC vont dans un futur proche libérer la SC et donc posséder un jeton libre. Les autres, qui ne sont pas intéressés par la SC, leur intervention n'est pas de rigueur. La file est triée par ordre croissant des numéros de séquence (selon les horloges de L. Lamport [76]) des requêtes reçues.

Si un jeton est disponible, la requête se trouvant au sommet de la file est la prochaine à satisfaire. Cependant, le chemin de cette requête peut être perdu suite à une rupture de lien. Cette dernière sera (en fonction des informations locales existantes telle que la durée de séjour de la requête dans la file, le nombre de sauts effectués par la requête, …etc.) insérée dans une file temporaire et sera traitée à la prochaine formation d'un lien. Nous distinguons ainsi deux

types de files d'attentes: une file d'attente *principale*, qui permet la mise en attente des requêtes en voie de satisfaction, et une file d'attente *temporaire* qui maintient en attente des requêtes ayant perdu leur lien avec le chemin de jeton, dans l'espoir que ce chemin soit renoué dans un futur proche (grâce à une formation de lien).

2.2.7. La cession de jetons

Cette opération vise à satisfaire un besoin spécifique et à rendre actifs les jetons stationnaires sans aucun coût supplémentaire. Dans ce protocole, il existe plusieurs situations où un nœud est appelé à céder un ou plusieurs jetons:

- Lors du passage d'un message de jeton de type *Collector*, en cas d'existence de jetons libres au niveau du nœud transité. Celui-ci les cède au profit du destinataire du message.
- Lors du passage d'un message de jeton de type *Collector*, dans le cas où le message comprend un nombre de jetons en surplus et le nœud transité est demandeur de SC. Un seul jeton de la collection sera cédé au profit de ce nœud transité.
- Lors du passage d'un message de jeton de type *Single*, dans le cas où le nœud transité est demandeur et *le seuil d'utilisation du jeton est non encore atteint* ou *le nombre de relances de la requête atteint un certain seuil*. Dans ce cas, il s'agit d'un 'emprunt'.
- Lorsqu'un nœud se *trouve* lié au réseau par un seul lien, il cède tous ses jetons libres à son unique voisin.

2.2.8. La trace

Elle est entretenue par chaque nœud *i* et sert à garder des informations nécessaires à la prise de décisions dans différentes situations. Ces informations peuvent être, par exemple, la position du noeud sur le chemin des requêtes qu'il a transité et celle de tous ses voisins relativement aux requêtes communes (sera vu dans la Section 3.4). Celle-ci permet l'optimisation du chemin du jeton destiné à satisfaire cette requête. En effet, si le nœud *i* vient de se lier au nœud *j* et se trouve sur le même chemin que lui d'une ou plusieurs requêtes, et si *i* se trouve à une position pos_i sur le chemin telle que: $pos_i < pos_j$ (pos_i et pos_j sont respectivement les positions relatives des nœuds *i* et *j* sur le chemin traversé par une requête commune), il serait intéressant de sauvegarder cette information afin que *j* soit le prochain nœud sur le chemin du jeton. A cet effet, lorsque ce cas se présente, les nœuds *i* et *j* s'échangent leurs positions respectives relativement à toutes les requêtes communes.

2.3. Structures

2.3.1. Structures de données locales à un nœud *i*

Chaque nœud *i* entretient un certain nombre de structures de données permettant de conserver les informations nécessaires au fonctionnement du service de *k*- EM.

- $Status_i$: indique l'état de *i* par rapport à la CS. Il peut être non demandeur (*free*), demandeur (*need*), demandeur isolé (*unable*) ou en section critique (*cs*);
- A_i : constante indiquant l'attribut de *i*, le nombre premier de numéro *(i+2)* dans l'ordre croissant des nombres premiers;
- NS_i: estampille locale gérée selon les horloges de L. Lamport [76];
- Req_NS_i: Numéro de séquence de la dernière requête envoyée par *i*;
- $Nb\_attemps_i$: Nombre de tentatives de la requête de *i* en cours;

- N_i: Information sur les voisins de *i*. C'est une liste dont le format de chaque élément est : *Id*, *A* et *NS* qui représentent respectivement l'identificateur d'un voisin, son attribut et l'estampille du dernier jeton envoyé à ce voisin;
- R_i: Nombre de relances de la dernière requête du nœud *i*;
- $Nb\_free\ \_token_i$: Nombre de jetons libres détenus par le nœud *i*;
- Way_i: Identité du nœud auquel *i* a envoyé un jeton le plus *récemment*. Si $Way_i = i$ cela signifie que le nœud *i* ignore le chemin du jeton;
- Q_i: Une file de requêtes triées selon l'ordre croissant de leurs numéros de séquence. Le format de chaque élément est : $j, Req\_NS_j, SN_j, T_j, Type, Nb\_token\_need$ qui représentent respectivement l'identité du nœud source, le numéro de séquence de sa requête, l'estampille logique d'arrivée de la requête, le produit des attributs de tous les nœuds traversés par la requête jusque là, le type de jeton demandé et le nombre de jetons sollicités par la requête. Le dernier champ ne sert que lorsque *i* insère dans sa file un message de jeton *Collector* qui n'a collectionné aucun jeton en cours de son parcours;
- Q_tmp_i: Une file temporaire de requêtes, utilisée lors du traitement de requêtes dont le chemin du jeton est rompu (impossible d'envoyer le jeton). Ces requêtes sont traitées lors de la formation d'un nouveau lien. Ces requêtes seront supprimées après un Timeout;
- $Trace_i$: Une liste qui sert lors de l'acheminent des requêtes, elle est de format: $(j, id, Req\_NS_j, T_j, Pos, Prd)$ qui représentent respectivement l'identité du nœud source de la requête, l'identité du nœud émetteur de la requête, le numéro de séquence de la requête, le produit des attributs des nœuds déjà traversés par la requête, la position de *i* sur le chemin de la requête et le produit des attributs des nœuds par lesquels est passé un jeton après l'acheminement de la requête. Le rôle de la trace est de maintenir de manière distribuée le chemin traversé par chaque requête;
- S_Trace_i: Une sous trace liée à $Trace_i$. Elle contient les informations sur des éventuels chemins optimaux et qui sont construits lors de la formation de liens et détruits lors de la rupture de liens. Une entrée de $Trace_i$ (qui représente une requête) correspond à plusieurs entrées de S_Trace_i dont chacune représente un éventuel chemin qui servirait à acheminer le jeton de la requête correspondante. Cette structure a le format suivant: *Idf*, qui représente l'identificateur du nœud voisin sur le chemin optimal et *pos* qui est la position de ce nœud voisin sur le même chemin.

2.3.2. Types de communications

Le service de *k* exclusion mutuelle utilise pour sont fonctionnement les messages suivants :

- Le message de requête, il est de la forme $Request\ (j, Req\_NS\ , NS_j, T_j, pos)$. Les éléments de ce message représentent respectivement l'identité du nœud source, le numéro de séquence de la requête, l'estampille enregistrée chez l'émetteur de la requête en cours de transit, le produit de tous les attributs des nœuds visités par la requête et la position du nœud émetteur sur le chemin de la requête.
- Le message de jeton, il est de la forme $Token\ (j, Target, Req\_NS, Type, Nb\_token\_need, Nb\_token\_grant, Pos, pass, util, Time, T)$. Les éléments de ce message représentent respectivement l'identité du nœud source (, cette information est utile pour le cas de retour du jeton, le nœud source devient destinataire), l'identité du nœud destinataire, le numéro de séquence de la requête pour laquelle ce jeton est envoyé, le type du jeton, le nombre de jetons demandés à travers ce message (, ce paramètre est utilisé pour le cas d'un jeton *Collector*), le nombre de jetons collectés par le message, la position du nœud émetteur sur le chemin du jeton (, cette information est utile pour la mise à jour du chemin de retour du jeton), le numéro de passe du jeton (sa première passe ou sa deuxième, si le jeton effectue sa première passe, la trace est mise à jour pour préparer le retour, s'il effectue sa deuxième

passe la trace est supprimée. Si ce paramètre est à *0*, il s'agit d'un jeton *Single*), le nombre de fois (, si le jeton est de type *Single,*) que ce jeton en transit est utilisé par les nœuds intermédiaires, l'estampille locale à laquelle le jeton est transféré, le produit des attributs de tous les nœuds par lesquels est passé le jeton (, cette information est utile seulement pour le cas de jeton *Collector* en première passe).

- Le message d'information, il est de la forme *Link_Info (Trace$_i$)* où *Trace$_i$* représente la structure de trace enregistrée au niveau de *i*. Ce message est échangé entre deux nœuds, à la formation d'un lien entre eux, à titre prévisionnel afin que ce lien puisse former un chemin de jeton potentiel qui permettra d'avoir un choix plus large quant à l'acheminement d'un jeton; ce qui aide à favoriser le chemin le plus optimal.

3. Description du protocole

Le texte du protocole est formé d'un ensemble de primitives liées aux différents événements le constituant. Au niveau de chaque nœud, les primitives s'exécutent de manière atomique. Cependant, les événements qui se produisent durant l'exécution de la SC peuvent être traités. Dans la Section 2.2, plusieurs possibilités de décisions sont exposées pour chaque problème donné. Afin de profiter de cette richesse, il est possible de présenter le protocole de manière *paramétrable* en supposant qu'il existe une procédure (i.e. sous forme de boite noir) pour chaque point de décision à laquelle sont fournie toutes les informations nécessaires et qui retourne une décision donnée. Cependant, des choix sont nécessaires pour des besoins de simulation, ils seront décrits dans la partie simulation. Le déroulement du processus de *k*-exclusion mutuelle est décrit dans les sections qui suivent.

3.1. Initialisation d'un nœud i

Lors de l'exécution de la procédure d'initialisation, toutes les variables locales sont initialisée. Chaque nœud *i* du système génère son attribut A_i dont la valeur correspond au *(i+2)* $^{\text{ième}}$ nombre premier, calcule localement les attributs de tous ses voisins. Le nœud d'identité *0* génère *k* jetons dont il restera détenteur tant qu'il n'a pas été sollicité ou qu'il n'est pas lié au réseau par un seul lien (ce qui suppose le risque d'un isolement futur si ce lien est rompu); auquel cas, il va léguer ses jetons à son voisin.

3.2. Acheminement d'une requête

Lorsqu'un nœud *i* désire entrer en SC, s'il possède au moins un jeton libre, il peut accéder à sa SC. Dans le cas contraire, il envoie sa requête à travers le dernier nœud voisin auquel il a transmis récemment un jeton. Si le nœud *i* n'a jamais acheminé de jetons précédemment, il envoie sa requête vers un voisin selon un autre critère prédéfini; par exemple: *le voisin dont l'identité est la plus petite* ou un voisin quelconque. Ainsi, en suivant le même principe, chaque nœud recevant la requête va tenter de l'acheminer jusqu'à un nœud détenteur de jeton(s) libre(s). Si le nœud *i* a perdu tous ses liens avec le réseau (il est isolé, état *unable*), il attend la formation d'un nouveau lien pour lancer sa requête.

Le jeton est attendu durant une période définie par un temporisateur *Request_timer*. A son expiration, la requête est considérée sans réponse. Les causes possibles sont par exemple le blocage de la requête dans une file d'attente ou la perte de chemin du jeton à cause d'une rupture de lien qui empêche l'acheminement de celui-ci vers son destinataire. Dans tous ces cas, une nouvelle tentative de relance de la requête est effectuée. Le nombre de relances est

limité par un seuil *RequestThreshold* après lequel, un échec (i.e. *Fail*) peut être observé pour le service de la requête.

Si un nœud *j* reçoit une requête d'un nœud voisin *i*, s'il détient au moins un jeton libre, il le transmet au nœud source (c'est-à-dire le propriétaire) de la requête. Si le chemin vers celui-ci est rompu, la requête peut être insérée dans la file d'attente temporaire dans l'attente d'être traitée ultérieurement. Si le nœud *j* se trouve en SC sans détenir de jetons libres ou en état demandeur de SC, il détermine en fonction des informations disponibles (nombre de nœuds traversés par la requête, nombre de fois où la requête a été rejetée, état de la file, nombre de relances effectuées par le nœud *j* pour sa propre requête en cours (en cas de l'état *Need*), …etc.) s'il doit insérer la requête dans sa file d'attente ou l'acheminer. Si la requête est insérée dans la file d'attente, elle l'est pour une durée limitée définie par le temporisateur *Valid_Timer(i)*; au-delà, si elle n'est pas satisfaite elle est purger de la file d'attente. La définition de *Valid_Timer* doit donc être judicieux et doit tenir compte de celle de *Request_Timer*. Parfois, une requête est effacée à la source pendant qu'elle se trouve en cours d'acheminement sur le canal, elle est donc ignorée. Si la requête est à acheminer, la valeur de *T* reçue est multipliée par l'attribut A_j du nœud *j*, ensuite, le nœud *j* choisit parmi les nœuds voisins dont l'attribut ne divise pas *T*, celui qui a reçu un jeton du nœud *j* le plus récemment ou à défaut, un autre comme précisé précédemment et lui envoie la requête. Cette stratégie de sélection permet d'éviter à la requête de revenir sur un chemin déjà visité. Néanmoins, il existe un cas où un nœud déjà exploré est sollicité: C'est le cas où un nœud a reçu un jeton du nœud *j* après avoir traité la requête.
Si tous les voisins sont explorés (tous leurs attributs divisent *T*), le nœud *i* retourne la requête à son prédécesseur sur le chemin de cette requête (la requête rebrousse chemin). Il convient de préciser qu'en suivant le cheminement d'une requête, celle-ci trace un chemin au moins linéaire et au plus arborescent.
Le nœud *j* peut être lui-même le propriétaire de la requête reçue. S'il est toujours à l'état demandeur (i.e. *Need*), il tente de la réacheminer sur une autre voie. Si tous les voisins sont explorés, le nœud *j* attend l'expiration du délai *Request_Timer* relatif à la requête afin d'effectuer une nouvelle tentative.

Une relance de requête est traitée comme une nouvelle requête. Cependant, la requête garde son numéro de séquence afin de maintenir son ancienneté dans le système.

3.3. Acheminement de jetons

A la réception d'un message de jeton, un nœud *i* doit distinguer plusieurs situations à savoir si le jeton lui est destiné, si le jeton est de type *Single*, *Collector* ou *Cession*, s'il est appelé à l'exploiter (en entrant en SC ou en traitant son éventuelle file d'attente), s'il doit l'acheminer vers son destinataire ou bien le garder. Si le nœud *i* est le destinataire du jeton reçu, et est toujours à l'état *Need* et si le jeton est de type *Single*, le nœud *i* peut entrer en SC et éventuellement traiter sa file dès sa sortie de la SC. Dans le cas où le nœud *i* n'en est pas le destinataire et est à l'état *Need*, il peut exploiter le jeton reçu et entrer en SC si le seuil d'utilisation du jeton n'est pas encore atteint. (Le seuil d'utilisation d'un jeton en cours de route peut être calculé en fonction du délai attribué à une requête pour être satisfaite et du nombre de nœuds traversés par la requête et le jeton.) Dans ce cas, à sa sortie de la SC, le nœud *i* doit réacheminer le jeton comme prévu sans aucun traitement de la file d'attente. Si le nœud *i* est le destinataire du jeton reçu et si le jeton est de type *Collector*, il faut déterminer s'il se trouve dans sa première ou sa deuxième passe (c'est-à-dire sur son chemin d'aller ou de retour). S'il se trouve dans sa première passe, le nœud *i* peut, s'il y a un besoin (c'est-à-dire: le nœud *i* est à l'état *Need* ou la file d'attente n'est pas vide), se *servir* d'un jeton et renvoyer les

autres à condition que le nombre de jetons collectés soit supérieurs à *1*; s'il est égal à *1*, le message de jeton est considéré comme une requête et sera inséré dans la file d'attente. S'il n'y a pas de besoins, les jetons libres qui pourraient se trouver chez le nœud *i* sont cédés à la collection qui sera réacheminée dans sa deuxième passe.
Si le jeton *Collector* se trouve déjà dans sa deuxième passe (le nœud *i* étant le destinataire), le nœud *i* se sert des jetons (s'il y a un besoin) sinon la collection de jetons reste localement jusqu'à une nouvelle demande. Si le nœud *i* n'est pas le destinataire du jeton *Collector* reçu, s'il est à l'état *Need*, et ce jeton détient dans sa collection plus de jetons qu'en a demandé le destinataire, un jeton sera 'consommé' et le reste acheminé comme prévu.

Rappelons que le chemin du jeton est tracé à l'avance par la requête qui l'a atteint. En effet, chaque nœud traversé par cette requête garde dans ses structures de données des informations sur le passage de cette requête et notamment, la valeur de *T* actuelle ainsi que sa position sur le chemin. Le jeton est destiné à suivre le chemin inverse à celui emprunté par la requête et ce, grâce à la valeur *T*. En premier lieu, le nœud détenteur du jeton, avant de l'envoyer, détermine si le nœud demandeur est l'un de ses voisins auquel cas il sera le destinataire direct du jeton (première optimisation possible); sinon, il sélectionne parmi ses voisins un nœud, dont l'attribut est un diviseur de la valeur *T* (cela signifie qu'il a participé à transiter la requête) et dont la position sur le chemin de la requête est la plus petite, et lui envoie le jeton après avoir divisé *T* par son propre attribut (opération inverse). On peut remarquer que plusieurs voisins peuvent être candidats. Or, la requête n'est venue que par un seul chemin. Ceci est du au fait que le système enregistre tous les chemins possibles afin de déterminer le plus optimal (cela permet d'optimiser le chemin du jeton). En effet, après réception d'une requête d'un nœud *j* par un nœud *i*, il se peut qu'un lien se forme avec un autre nœud (autre que *j*) qui a lui-même participé à son acheminement. Toutes les informations nécessaires (identité du nœud, sa position sur le chemin de la requête, …etc.) sont enregistrées localement afin de les exploiter au moment opportun. Il est clair que certains nœuds ayant participé à l'acheminement de la requête peuvent ne pas être sollicités pour l'acheminement du jeton. Ils seront appelés à purger leurs structures de données relatives à cette requête dans des délais fixés dans *Valid_Timer*.
Si le jeton est de type *Collector*, le même principe d'acheminement est appliqué pour l'aller et le retour; sauf que dans l'allée, le chemin doit être inversé pour assurer le retour. Enfin, si le message de jeton est de type *Cession*, les jetons reçus seront préservés jusqu'à une nouvelle demande.

A la sortie de sa SC, le nœud *i* sauvegarde selon le besoin toutes les informations relatives au jeton exploité telles que les circonstances d'obtention (détenu, cédé, exploité en cour de route, …), les informations véhiculées avec le message de jeton, etc… Ensuite, ce nœud vérifie les paramètres en sa possession; s'il est le propriétaire du jeton qu'il vient d'exploiter, il peut procéder à la satisfaction des requêtes en attente dans sa file: si sa file est vide, le jeton reste stationnaire. Si la file contient une seule requête, le jeton est envoyé à l'élément du sommet de file avec la propriété *Single* sinon, sa propriété sera *Collector*; ce qui signifie que ce jeton a non seulement le rôle de satisfaire la requête se trouvant au sommet de la file, mais aussi il sera appelé à revenir au nœud *i* afin de pouvoir satisfaire les requêtes restantes en collectant tous les jetons stationnaires qui se trouvent sur son chemin dans les deux sens (aller et retour). Si le jeton exploité par le nœud *i* n'est pas sa propriété cela signifie que le nœud *i* vient d'exploiter un jeton en cours de transit; on peut distinguer deux situations possibles:

- le jeton exploité possède la mention *Single*: dans ce cas, la file n'est pas traitée et le jeton sera envoyé à son propriétaire après incrémentation du nombre d'utilisations du jeton avec la même mention *Single*;

- le jeton exploité a été cédé par un message de jeton *Collector* sur son passage par le nœud *i*: dans ce cas, le nœud *i* agit comme s'il était le propriétaire du jeton, c'est-à-dire, il traite sa file d'attente, car un jeton *Collector* n'est cédé que lorsqu'il est en surplus dans sa collection.

Lors de l'exécution d'une SC, les évènements de formation et de rupture de liens peuvent se produire et être traités. Dans ce cas, à sa sortie de la SC, le nœud *i* peut se retrouver lié au réseau par un seul lien (possibilité d'isolement dans un futur proche), celui-ci procède à la cession de ses jetons libres après avoir traité sa file d'attente.

3.4. Réactions aux changements de liens

A la formation d'un nouveau lien avec un nœud *j*, le nœud *i* procède à la mise à jour de la liste de ses voisins et envoie un message *Link_Info ()* contenant des informations relatives aux requêtes se trouvant dans la file d'attente (telle que l'identité du nœud source de la requête, la position du nœud sur le chemin de la requête, …etc.). Cela peut servir à optimiser le chemin du jeton. En fonction de la situation du nœud *i*, une ou plusieurs actions peuvent être entreprises comme suit:

- Si le nœud *i* se trouve dans l'état *unable*, sa requête est relancée.
- Si le nœud *i* possède des jetons en ayant une file temporaire non vide, il vérifie si ce nouveau lien peut lui servir pour acheminer au moins un jeton vers un destinataire se trouvant dans la file temporaire et envoie le jeton.
- Si ce nœud se trouve en état isolé et détient des jetons libres, ceux-ci seront acheminés sur le nouveau lien (jetons de *Cession*).

Lorsque le nœud *j* reçoit le message *Link_Info ()*, il vérifie si le nœud *i* se trouve à une position plus petite que la sienne sur les chemins des requêtes communes. Dans ces cas, le nœud *i* est considéré comme un chemin potentiel optimum.

Dès qu'un nœud *i* détecte la rupture d'un lien avec un nœud *j*, il procède à la mise à jour de la liste de ses voisins. S'il ne lui reste qu'un voisin et qu'il est détenteur d'un ou plusieurs jetons libres, ces derniers sont immédiatement envoyés vers ce voisin; cela va permettre de protéger les jetons libres d'un éventuel isolement du nœud *i*, ce qui pourrait les rendre inactifs même pour un laps de temps.

4. Exemple d'exécution et discussion

4.1. Exemple d'exécution

Cette section décrit un exemple simple permettant d'illustrer le fonctionnement du protocole de *k*- exclusion mutuelle. Le scénario décrit permet de mettre en évidence certains aspects clés tels que la technique de recherche du jeton (cheminement d'une requête), détermination du chemin du jeton, …etc. Les jetons considérés pour l'exécution sont au nombre de trois. Les décisions à prendre à chaque occasion seront les plus simples, c'est-à-dire ne considèrent (ne dépendent) pas énormément des informations récoltées par les messages de service de *k*- exclusion mutuelle. La technique de recherche appliquée dans le cas où un nœud ne connaît aucun chemin de jeton est de solliciter le voisin dont l'identité est la plus petite parmi les nœuds non visités par cette requête. D'autre part, une requête est enfilée sans condition au niveau d'un nœud demandeur ou en SC.

Supposons une topologie à *huit* nœuds. A l'état initial, les trois jetons sont détenus par le nœud *0* (Figure 27(a)). Le nœud *3* formule une requête qui, n'ayant pas de chemin préliminaire, est adressée au nœud *1*. Le nœud *1* choisi parmi ses voisins celui qui a la plus petite identité (le nœud *2*) et lui adresse cette requête. Le nœud *2* ne dispose d'aucun jeton, il retourne la requête au nœud *1*. La requête du nœud *3* suit alors le chemin *1, 2, 1, 5, 4, 0*. Supposons maintenant que le noeud *7* formule une requête; celle-ci suit le chemin *1*, *2*, *1*, *3* et sera enfilée au niveau du nœud *3* étant donné que celui-ci est demandeur de SC. Entre temps, le nœud *1* désire entrer en SC et formule une requête qu'il adresse au nœud *2*. Cette requête lui revient et l'adresse au nœud *3* qui l'insère de même dans sa file.

La Figure 27(b) montre un chemin optimisé du jeton véhiculé depuis le nœud *0* vers le nœud *3*. En effet, le chemin inverse de la requête est *0, 4, 5, 1, 3*. Etant donné que le nœud *4* est le voisin direct du nœud *3*, ainsi que le chemin *0, 4, 3* est préféré. Le nœud *3* entre en SC. Le nœud *3*, après sa sortie de la SC, envoie le jeton au nœud se trouvant au sommet de sa file (i.e. le nœud *7*) avec la mention *Collector*. Ce jeton est appelé à revenir vers le nœud *3* puisqu'il reste encore la requête du nœud *1* à satisfaire. Arrivé à destination, le message de jeton est considéré comme une requête et est inséré dans la file du nœud *7* jusqu'à sa sortie de la SC. Entre temps, le nœud *6* désire entrer en SC et envoie sa requête au nœud *0*. Le nœud *6* reçoit aussi son jeton et entre en SC et en sortant, il le garde locallement.

Dans la Figure 27(c), le nœud *7*, en sortant de sa SC, envoie le jeton vers le nœud *3*, qui n'étant pas demandeur, satisfait la requête qui se trouve au sommet de sa file. Le nœud *1* reçoit ainsi le jeton et entre en SC. Rappelons que chaque nœud garde trace du nœud auquel il a envoyé un jeton la dernière fois.

Partant de la Figure 27(c), un lien est crée entre les nœuds *0* et *5* puis le lien entre les nœuds *0* et 6 est rompu. Le nœud *6* se trouvant lier au réseau par un seul lien, cède son jeton au nœud *3* qui le garde au repos. Le nœud *7* envoie une requête de SC au nœud *3* (i.e. le nœud auquel il a envoyé récement un jeton). Celui-ci lui renvoie le seul jeton disponible localement. Ensuite, le lien entre *2* et *1* est rompu et un lien est crée entre *2* et *7*. Supposons aussi que le nœud *2* demande le jeton, il envoie sa requête au seul voisin, le nœud *7*. Celui-ci garde la requête localement puisqu'il est en SC. En sortant de sa SC, le nœud *7* envoie le jeton rendu disponible au nœud *2* qui entre en SC (Figure 27(d)).

Maintenant (Figure 27(d)), les nœuds *6*, *4* et *3* demandent successivement la SC, leurs requêtes respectives passent par les chemins *3*, *7*, *2*; *3*, *7*, *2* et *7*, *2*. Elles sont toutes enfilées au niveau du nœud *2* qui est en SC.

En supposant que le nœud *1* a déjà fini d'utiliser sa SC et il vient de perdre successivement ses liens avec les nœuds *3* et *5*, il cède alors le jeton au nœud *7* (Figure 27(e)). Celui-ci garde le jeton localement puisqu'il n'a enfilé aucune des requêtes qui l'ont traversés. A sa sortie de SC, le nœud *2* prenant acte du nombre de requêtes présentes locallement, envoie le jeton à destination du nœud *6* avec la mention *Collector*. Le long de son parcours de la première passe, ce message récolte un jeton oisif au niveau du nœud *7*. Arrivant au nœud *3*, celui-ci, malgrès qu'il est demandeur, ne peut pas exploiter de jeton puisque le nombre de jetons transportés par le message de jeton reçu n'est pas suffisant. A l'arrivée à destination, le nœud *6* exploite un jeton et retourne (i.e. deuxième passe) au nœud *2* le jeton *Collector* avec un seul jeton. Entre-temps, un lien est crée entre le nœud *2* et le nœud *3*. Le jeton *Collector*, arrivant au nœud *2*, sera ensuite envoyé comme jeton *Single* pour satifaire le nœud *4*. Celui-ci est envoyé directement au nœud *3* sans passé par le nœud *7* étant donné que les nœuds *2* et *3* sont sur le chemin d'une même requête, celle du nœud *4* (, il s'agit d'une optimisation de chemin grace à la création du lien entre les nœuds *2* et *3* !). Mais, avant d'arriver au nœud *4*, le nœud *3* exploite ce jeton et ensuite le passe à sa destination.

Dans la Figure 27(f), un certain nombre d'vénements se produisent, le lien entre les nœuds *7* et *3* est rompu; ensuite, le nœud *1* envoie une nouvelle demande de SC au nœud *7*, le seul nœud voisin. Le lien entre les nœuds *2* et *3* se rompe; ensuite, la requête du nœud *1* passe au nœud *2* qui la retourne au nœud *7* puis passe au nœud source *1*. Celui-ci la garde locallement et passe à l'état *unable* en attendant l'établissement d'un nouveau lien pour véhiculer sa requête ou le passage d'un jeton. On suppose maintenant que le lien entre le nœud *2* et *3* est réétabli. Le nœud *3* envoie une nouvelle requête au nœud *4* (c'est le nœud auquel il a envoyé recement un jeton). Le nœud *4* la garde locallement puisqu'il est en SC , ensuite le lien entre *3* et *4* est rompu. Le nœud *4*, en sortant de sa SC, perd le chemin vers le nouveau destinataire du jeton (qui vient d'être rendu disponible). Il met alors la requête du nœud *3* dans la file temporaire. Après expiration du *RequestTimer* au niveau du nœud *3*, celui-ci renouvelle sa requête qui suit le nouveau chemin *2*, *7*, *1*. Le nœud *1* la garde locallement puisqu'il est demandeur. Malgrès que le nœud *3* a déjà véhiculé un jeton au nœud *6*, il ne s'en souvient pas!

Dans la Figure 27(g), le nœud *6*, en sortant de sa SC, cède le jeton au nœud *3* puisqu'il se trouve lier au réseau par un seul lien. Le nœud 3 exploite le jeton ainsi reçu. Supposons que par la suite les liens se forment respectivement entre les nœuds *4* et *3* et *4* et *1*. Le nœud *4* envoie alors le jeton au nœud *3* dont la requête est présente dans sa file temporaire. Ce jeton est gardé au repos au niveau du nœud *3*. Le nœud *1*, en profitant du nouveau lien, envoie sa requête sur le chemin *4*, *3*. Le nœud *3* lui renvoie le jeton disponible sur le chemin inverse.

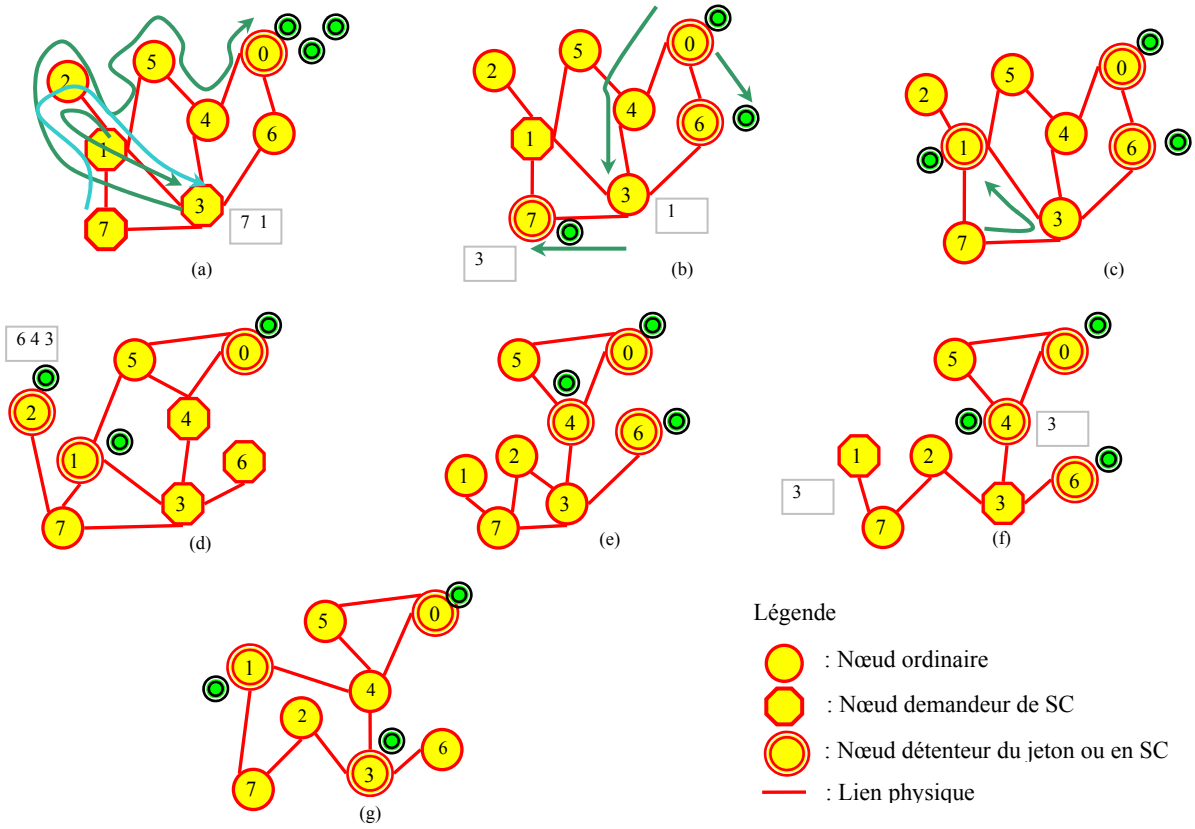


Figure 27: *Un exemple d'exécution du protocole de k- exclusion mutuelle.*

4.2. Discussion

Dans ce protocole, deux outils de base liés au fonctionnement ont été utilisés, en l'occurrence heuristiques et les nombres premiers.
Les heuristiques ont été utilisées à deux niveaux :

- Recherche de jeton: A ce niveau, la technique de recherche arborescente est utilisée. Elle permet d'assurer l'existence d'un chemin entre deux nœuds différents du graphe. Son avantage est l'abstraction de la topologie du réseau
- Prise de décision quant au comportement à adopter: Les actions à exécuter par un nœud ont été paramétrées. Ceci a pour objectif de rendre le comportement de chaque nœud plus adaptatif et plus interactif.

Les nombres premiers ont déjà étaient exploités dans le problème d'exclusion mutuelle distribuée par M. Raynal [112] qui a démontré leur puissance quant à la détermination de l'état global d'un système distribué à l'instar des horloges logiques. Mais tous les calculs qu'il propose sont effectués de manière isolée et locale. Dans notre protocole, tous les calculs sont distribués, il y a une véritable coopération entre les processus du système. Les nombres premiers servent pendant l'acheminement d'une requête au cours duquel le produit des attributs des nœuds visités est calculé de manière distribuée. Les nombres premiers se sont avérés très efficaces compte tenu de leur propriétés [47]: Il existe une infinité de nombres premiers et la factorisation d'un nombre en facteurs premiers est unique. Cependant, lorsque le nombre de nœuds visités augmente, le produit de leurs attributs augmente très rapidement au risque de causer un dépassement de capacité mémoire. Pour palier ce problème, il est utile d'utiliser plusieurs variables pour stocker dans chacune, une partie du produit.

5. Extension du protocole

Avant d'entamer cette extension, il est nécessaire de fixer un certain nombre d'éléments de bases utiles pour cette opération. Pour cela, on s'inspire des solutions définies précédemment. Tout d'abord, tous les jetons d'une même partition doivent être identifiés de manière unique. L'identification retenue se compose évidemment d'une identité (*Tid*, celle du nœud qui les a créé) et d'une estampille (*TidNS*, un numéro d'ordre local au nœud qui les a créé). Chaque partition est connue par l'identification des jetons qui y circulent. Cette information est connue de tous les nœuds de la partition.

5.1. Partitionnement et perte de jeton

Le manque de certains jetons (perte ou déplacement dans d'autres partitions) est soupçonné lorsqu'un nœud demandeur de jeton atteint le seuil de relance ($Nb\_attempts_i = Req\_Threshold$). En ce moment, il change la manière de demander le jeton. Il diffuse un message *IsThereToken ()* dans toute la partition courante. Ce message est à double objectif: Demande aux processus qui détiennent des jetons de les supprimer même s'il sont en SC et proposition de création des *k* jetons. Le nœud *i* se *bloque* ensuite (i.e. met $Search\_Token_i$ à *true*, ce qui le même à ne plus générer de demandes) en attente de la régénération de jetons pendant une durée de temps limitée (i.e. *SuspicionTimer).* Au delà, le traitement du partitionnement (i.e. la régénération de jetons) est appliqué. Etant donné que plusieurs nœuds peuvent réaliser cette opération simultanément*, un seul* doit se charger de cette création. Il peut être, par exemple, celui de plus petit identificateur. *Celui-ci s'identifie par le fait qu'il trouve l'identité suggérée aux jetons égale à la sienne.* Les *k* jetons créés par ce nœud extremum (, dont il devient détenteur,) auront comme identité celle de ce nœud et comme

estampille le numéro de séquence proposé par ce nœud et le protocole *redémarre* pour cette partition. Les nœuds qui ne créent pas de jetons prennent alors acte de l'identification des nouveaux jetons.

Il est à constater que cette création de jetons ne se produit qu'après un délai fixé, ce qui permet de temporiser les nœuds en SC d'en sortir. A leur sortie de SC, ils n'ont pas à véhiculer de jeton étant donné qu'ils ont étaient détruits! Ce qui permet d'assurer la sûreté du service de *k*- EM.

Quand un nœud *j* reçoit un message *IsThereToken ()*, il détruit *immédiatement* les éventuels jetons en sa possession et considère ce message comme une exploration pour l'élection à la création des *k* jetons. Ceci le mène à participer à cette élection en mettant à jour les variables appropriées, et continue éventuellement la propagation de ce message et se bloque (, en déclenchant son propre temporisateur *SuspicionTimer,*) s'il ne l'était pas.

Deux questions principales se posent dans ce contexte. La première est: Pourquoi ne pas limiter la régénération *aux seuls jetons perdus* au lieu de tous les jetons? Il est délicat de remonter l'information sur la présence de ces jetons pour deux raisons principales. La première, est à qui la remonter étant donné qu'il s'agit d'une élection qui est en cours et le nœud élu n'est pas encore connu. La seconde, est même si on attend la fin de l'opération (qui fait perdre du temps), il est difficile, voire pratiquement impossible de faire remonter cette information sans procéder à de nouvelles diffusions, ce qui est coûteux. La deuxième question est : Pourquoi *ne pas éviter de régénérer* les jetons s'il n' y a pas perte? Il faut attendre la remontée d'informations sur la présence de jeton(s) pour déduire celui ou ceux perdu(s), ce qui coûteux pour les mêmes raisons que la première question.

5.2. Fusion de partitions

La fusion est détectée à l'occasion de créations de liens. A chaque création de lien, les deux nœuds le formant s'échangent les identifications de leurs partitions respectives. En cas de duplications de jetons, seuls ceux de la partition *prioritaire* restent valides et les autres seront supprimés.

Dès qu'un nœud *i* détecte un nouveau voisin, il lui transmet dans le message *LinkInfo (),* déjà prévu dans la solution de base, des informations sur son état de la manière suivante: Si *i* n'est pas bloqué, seules les informations (identité et estampille) de la partition de *i* sont transmises. Si *i* est bloqué, le message *LinkInfo ()* inclus aussi les informations du message *IsThereToken ()* traversant ou généré par *i*. Ce mécanisme permet de remédier aux cas de déconnexions momentanées: Le nœud correspondant peut être le seul relais vers l'autre portion de la même partition qu'il faut informer d'un recherche en cours de jetons!

Quand un nœud *j* reçoit un message *LinkInfo ()* de la part de *i*, s'il trouve qu'il sont associés aux mêmes jetons (i.e. le lien créé est alors interne à la partition) et selon les informations reçues, *j* peut exécuter un traitement équivalent à celui exécuté lors de la réception d'un message *IsThereToken ()*. Par contre, si les jetons sont différents alors il y a eu une fusion et les jetons d'une partition doivent être supprimés. (Le choix porte par exemple sur les jetons les plus récents dans le cas de jetons créés par le même nœud, ou sinon sur ceux avec l'identité la plus grande). Si le jeton à supprimer est celui de la partition de *j*, celui-ci diffuse le message *DeleteToken ()* dans sa partition après avoir exécuté la même tache de celui qui reçoit ce même message.

Quand un nœud *r* reçoit un message *DeleteToken ()*, si le jeton de la partition locale est moins prioritaire, alors il met à jour ses informations locales par celles reçues, réinitialise son contexte (vide ses files, se débloque s'il était bloqué, désactive ses temporisateurs éventuels,

etc…) et continue la diffusion de ce message. De plus si *r* est détenteur de jeton(s), il le(s) supprime même s'il est en SC, et cela ne pose pas de problème car à la sortie de la SC, il se trouvera sans jeton. Dans tous les cas, si *r* est demandeur de jeton, il arme le temporisateur *Request_Timer* pour relancer sa requête plus tard.

6. Etude de correction

Le but du service de *k*- exclusion mutuelle est fondamentalement la satisfaction des deux propriétés, en l'occurrence la sûreté et la vivacité. En premier, on s'intéressera à la sûreté. Cette propriété rappelons le, implique que chaque processus ne peut accéder qu'à une seule ressource à la fois et au maximum *k* processus exécutent leurs SC simultanément. Rappelons que *k* jetons sont gérés au niveau de chaque partition. De plus, seuls les nœuds détenant des jetons sont habilités à satisfaire les requêtes, une par jeton. Si le protocole assure qu'à tout moment au maximum (étant donné que certains nœuds détenteurs de jetons peuvent tomber en panne) *k* jetons résident au niveau de chaque partition, la propriété de sûreté est alors vérifiée. Evidemment, dans l'absence de la création de jeton et du fusionnement de partitions (i.e. dans la solution sans extension), cette propriété est assurée étant donné qu'au maximum *k* jetons circulent dans la partition courante. Nous allons étudier la sûreté dans deux situations: A la création de nouveaux jetons après perte, à la suppression de jeton(s) durant le fusionnement de partitions.

Pour la première situation (voir aussi la Section 5.1), supposons qu'un ou plusieurs nœuds sont en cours de recherche du jeton. Tous les noeuds de la partition courante seront informés au même titre que ceux en déconnexion temporaire (par l'utilisation des temporisateurs qui attendent leurs reconnexions et des messages d'informations *LinkInfo ()*). Si un nœud détient au moins un jeton (ou le reçoit pendant qu'il est dans l'état de recherche du jeton), il le détruit immédiatement. A l'expiration de son temporisateur (*SuspicionTimer*), un nœud doit créer les *k* jetons avec une nouvelle identification. Ce *seul* nœud a été choisi par un processus d'élection qui a accompagné la recherche de jetons. Comme résultat de cette élection, tous les nœuds ont déjà détruits leurs jetons éventuels, sont sortis de leurs SC et sont informés de l'identification des nouveaux jetons. Par conséquent, *k* jetons existent dans la partition ce qui vérifie la sûreté.

Pour la seconde situation (voir aussi la Section 5.2), supposons deux partitions chacune avec ses propres jetons identifiés de manière unique. Pour faciliter la présentation, considérons que chaque partition est colorée différemment (en *bleu* ou en *rouge*) et cette couleur correspond à ses jetons. Admettons maintenant qu'un lien est établi entre les nœuds *i* et *j* appartenant chacun à l'une de ces deux partitions. Quelque soit la politique qui gère la priorité des jetons, les jetons d'une partition deviennent moins prioritaires par rapport aux autres, par exemple ceux de la couleur *bleu*. Par conséquent, le nœud *j* va diffuser dans sa partition le message *DeleteToken ()*. Ainsi, à la fin de cette dessimination, tous les nœuds vont être de couleur *bleu*. Un délai peu s'écouler entre le moment du fusionnement et la suppression effective des jetons en dépendant de la vitesse de propagation du message de suppression du jeton. Des mécanismes similaires à ceux du Chapitre VII, Section 7.3 peuvent être adoptés pour y remédier. La propriété de sûreté est alors assurée.

Considérons maintenant la propriété de vivacité qui implique que chaque nœud demandeur de ressources arrive à les acquérir. Plusieurs mécanismes ont été appliqués pour favoriser cette propriété. Le premier mécanisme est le suivi des traces d'un jeton par une requête (voir Section 2.2.4). En effet, celle-ci voyage pour atteindre dans les meilleurs cas un nœud détenteur de jeton. Dans ce cas, le jeton fait le chemin inverse de cette requête grâce au

chemin tracé par celle-ci. La requête peut aussi être hébergée chez le premier nœud rencontré demandeur de SC car il est considéré comme un futur acquéreur de jeton. Ce qui implique qu'un nœud détenteur d'au moins un jeton est indirectement au courant des requêtes lointaines. Mais, ce mécanisme est insuffisant pour satisfaire toutes les requêtes. Le second mécanisme est la relance de requêtes (voir Section 2.2.3). En effet, un nœud qui ne reçoit pas le jeton durant une période fixe doit relancer sa requête. Afin de garder son ancienneté, cette requête est relancée avec la même estampille. Dans le cas échéant, ce processus est répété un certain nombre de fois (i.e. *Req_Threshold*) avant de suspecter la perte de jeton(s). Le troisième mécanisme est évidemment la recherche et la régénération des jetons (voir Section 5.1). Quand la perte de jeton(s) est suspectée, tous les *k* jetons seront régénérés et par suite les nœuds demandeurs régénèrent leurs requêtes après un délai fixe. Le quatrième mécanisme est la politique de satisfaction de requêtes au niveau d'une file locale (voir Section 3.3) qui utilise l'ancienneté des requêtes. Au court du temps, les anciennes requêtes deviennent nécessairement plus prioritaires et par suite, servies en premier. Le cinquième mécanisme est l'utilisation du jeton *Collector* qui permet de faire circuler des jetons stationnaires et satisfaire certains nœuds demandeurs sur le chemin du destinataire à l'aide des jetons en surplus (voir Section 3.3). Le sixième mécanisme est l'utilisation de files temporaires qui gardent des requêtes pour lesquels le jeton a perdu le chemin vers celle-ci en attendant la création de nouveaux chemins qui éventuellement mèneront aux propriétaires de ces requêtes. D'autres mécanismes ont aussi été utilisés, ils permettent d'éviter la famine de nœuds. Ces mécanismes sont liés aux différentes heuristiques liés aux décisions de *ce qui semble être meilleur* qui se basent sur la récolte d'informations.

Pratiquement, si on considère les résultats de simulations (qui seront détaillés dans la Section 7), nous constatons qu'il n' y a pas de temps infini d'attente d'entrée en SC et que le nombre de relances et le temps d'attente d'entrée en SC sont bornés. Ces résultats, qui ne considèrent pas bien sur le partitionnement, montrent que tous les nœuds demandeurs arrivent à accéder à leurs SC.

7. Etude de simulation

L'évaluation des performances du protocole est effectuée à l'aide de l'outil NS2 (*Network Simulator 2*) [136] version 2.26 sous le système d'exploitation LINUX MANDRAKE 9.0. La codification du protocole est effectuée à l'aide du Langage objet Otcl. Cette simulation vise deux objectifs:

- Connaître le fonctionnement et les performances de notre protocole dans des situations divers et qui prennent en charge ses aspects dynamiques. *Toutes les simulations réalisées ne considèrent pas la tolérance aux pannes*.
- Situer notre protocole par rapport à son seul prédécesseur, i.e. le protocole *KRL* [147].

7.1. Environnement de simulation et paramètres

Dans cette simulation, nous avons sélectionné les mêmes paramètres que ceux utilisés dans [147] afin de fournir des éléments de comparaison dans la mesure du possible. Néanmoins, des scénarios de simulation propres à notre protocole ont été réalisés afin d'observer son comportement et d'évaluer ses performances face à des situations préfixées. Les paramètres de simulation incluent le nombre de nœuds qui est fixé à *30*, le nombre de jetons qui varie entre *1*, *3*, *5* et *8*, la durée d'une SC est de *1* unité de temps et la durée de transmission d'un message est également de *1* unité de temps. Le temps d'intervention du système est considéré comme nul. Les autres paramètres pris en compte sont:

- La mobilité: Elle est donnée par le modèle aléatoire se basant sur le mouvement relatif des nœuds (voir Chapitre I). Ainsi, les mobilités retenues sont: La mobilité nulle: les nœuds sont soit stationnaire soit dans un état de mouvement qui n'affecte pas les liens ni en terme de formation ni en terme de rupture. Par exemple, le cas de mouvement relatif. La faible mobilité: le nombre de changements de liens varie entre *2* et *11* par unité de temps. La forte mobilité: le nombre de changements de liens est supérieur à *30* changements par unité de temps.
- La charge: Elle correspond à une variable aléatoire exponentielle avec une moyenne de $1/\lambda_{req}$ unités de temps. Les valeurs de λ prises pour la simulation sont: *0.001*, *0.01*, *0.1* et *1*.
- La connectivité: Elle est calculée en termes de pourcentage des liens existant par rapport au nombre de liens possibles présents dans le graphe. Les connectivités considérées sont de *20%* et *80%* correspondant respectivement aux faibles et fortes connectivités.
- Conditions générales de simulation : Outre les paramètres cités ci-dessus, certaines conditions doivent être préparées afin d'assurer une simulation cohérente et qui s'approche le plus de la réalité. Par exemple, le lancement des requêtes. Dans notre simulation, nous avons lancé les requêtes aléatoirement pendant les *30* premières unités de temps à partir du commencement de la simulation.

Concernant la gestion de la file d'attente: Nous avons exécuté trois variantes de protocole selon la manière de gérer la file d'attente:

- Dans la première, une seule requête est acceptée en file d'attente. Cette contrainte a été appliquée dans le cas où le nombre de jetons est suffisamment grand (i.e. *5* et *8*).
- Dans la deuxième, un nombre fixe de requêtes est accepté en file d'attente. Cette contrainte a été appliquée dans le cas où le nombre de jetons est égal à *3*.
- Dans la troisième, ayant un seul jeton, toutes les requêtes arrivées à un nœud en SC ou demandeur d'accès sont admises dans la file.

7.2. Résultats et interprétation

Afin d'étudier les performances du protocole, les quatre métriques suivantes ont été prises en compte:

- Temps moyen d'attente par entrée en SC (*Tatt/ SC*): C'est le nombre moyen d'unités de temps écoulées entre la demande d'accès en SC et l'accès lui-même.
- Nombre moyen de messages générés par entrée en SC(*Nb_Msg/ SC*): C'est le nombre moyen de messages émis (messages de requête, messages de jeton, messages *LinkInfo ()* et messages de *Cession*) entre la demande d'accès en SC et l'acquisition du jeton.
- Délai de synchronisation (*DélaiSynch*): C'est le temps moyen écoulé en terme de nombre moyen de messages générés entre deux entrées successives en SC.
- Nombre de relances (*Nb_Relances/ SC*): C'est le nombre moyen de renouvellements d'une demande d'acquisition d'un jeton relativement à une même requête.

Dans ce qui suit, nous allons présenter les résultats obtenus des différentes simulations en considérant séparément chacun des cas de mobilité (nulle, faible et forte). Nous intégrons dans cette étude, les résultats obtenus par les simulations du protocole *KRL* afin de permettre une comparaison même approximative des deux protocoles. Il est à noter que vu la mobilité telle qu'elle est considérée par J. Walter et *al* dans [147] et les valeurs de mobilité considérées par nos simulations, les mobilités définies dans [147] sont très faibles. Pour cela, les résultats issus des simulations du protocole *KRL* dans le cas de forte mobilité sont comparés à ceux issus de nos simulations dans le cas de faible mobilité, le cas de faible mobilité des

simulations du protocole *KRL* est ignoré. Le cas de mobilité nulle ne pose aucun problème. En outre, dans les simulations du protocole *KRL*, seuls les deux premiers critères de performances (à savoir le temps d'attente et le nombre de messages pour une entrée en SC) sont pris en considération. Dans la suite (principalement au niveau des graphes), notre protocole sera qualifié de protocole *KEM*.

7.2.1. Résultats se rapportant aux différentes métriques avec trois jetons.

Nombre de messages par entrée en SC (Nb_Msg/ SC)

Similairement aux courbes issues des simulations du protocole *KRL* (Figures 28(a) et 28(b)), celles issues de notre protocole ont une allure plutôt décroissante et semblent se confondre. Ces allures montrent d'une part, que les meilleurs résultats en terme de nombre de messages par entrée en SC sont enregistrés lorsque la charge augmente et d'autre part, que la connectivité n'a pas une grande influence sur les performances. Notons aussi que ces courbes expriment que notre protocole offre les meilleures performances relativement à cette métrique.

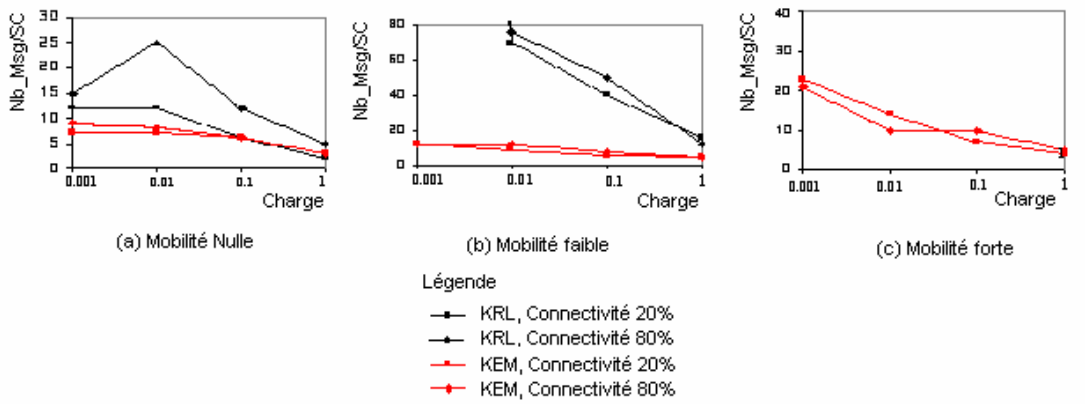


Figure 28 : *Nombre moyen de messages par entrée en SC.*

En général, il est à constater à travers les courbes de *KEM* (Figure 28) que les résultats sont meilleurs (dans le cas de mobilité nulle (tracé des courbes dans l'intervalle *[0, 10]* de la Figure 28(a)) que les cas dynamiques; et dans le cas avec faible mobilité (tracé des courbes dans l'intervalle *[0,20]* de la Figure 28(b)) meilleurs que ceux du cas avec forte mobilité (tracé des courbes dans l'intervalle *[0,25*] de la Figure 28(c)).

Temps d'attente moyen par entrée en SC (Tatt/ SC)

Tous les graphes de la Figure 29 montrent que les courbes de *KEM* ont gardé l'allure décroissante contrairement à celles de *KRL* (Figures 29(a) et 29(b)).

Les résultats de notre protocole semblent être meilleurs que celles du protocole *KRL* dans les cas de forte charge (Figures 29(a) et 29(b)). Les délais d'attente dans le cas de faible connectivité (20%) sont plus importants que ceux de la forte connectivité (80%), mais les courbes restent relativement proches (Figure 29).

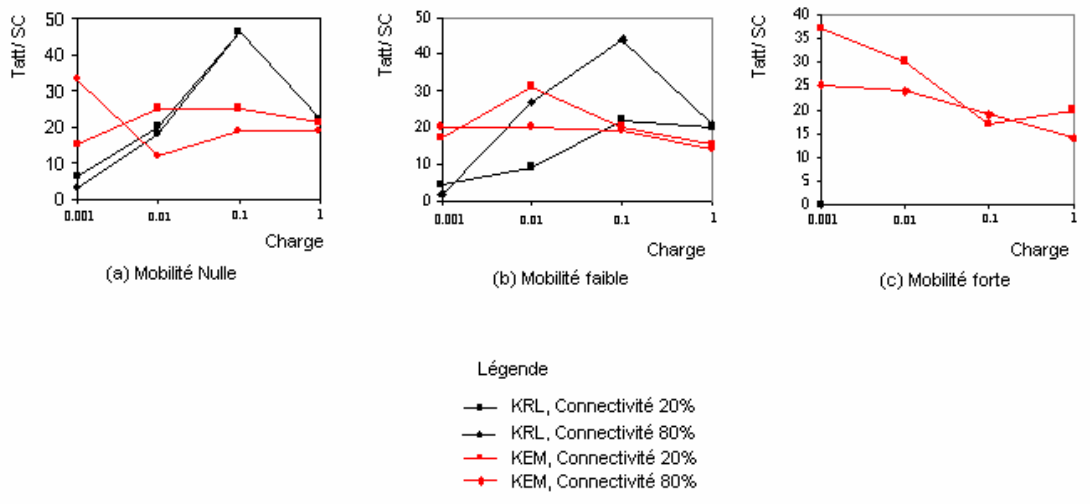


Figure 29 : *Temps d'attente moyen par entrée en SC.*

Délais de synchronisation (DélaiSynch)

Les graphes de la Figure 30 montrent l'existence d'un écart mineur entre les courbes en faisant varier la connectivité. L'allure de celles-ci étant décroissante avec un léger avantage des conditions de faible connectivité par rapport à la forte connectivité.

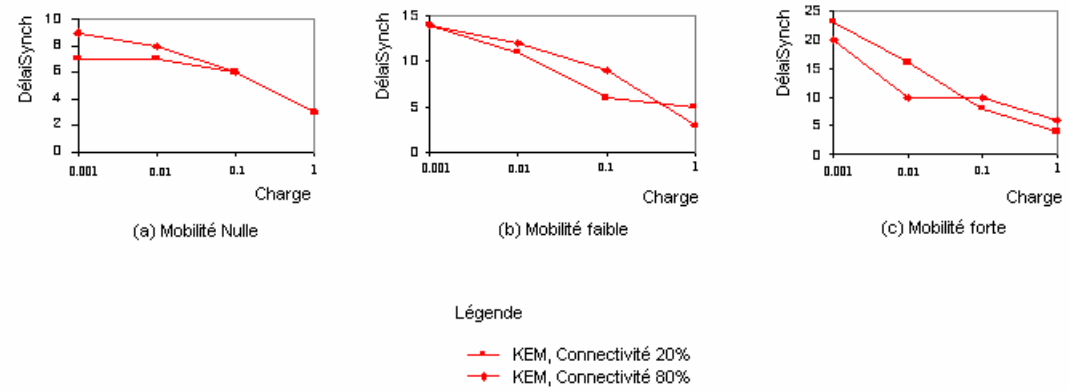


Figure 30 : *Délai moyen de synchronisation par entrée en SC.*

Nombre de relances (Nb_Relances/ SC)

Il est à constater en général, à partir des graphes de la Figure 31 que le nombre de relances par entrée en SC est plus important lorsque la connectivité est faible, ce qui est prévisible.

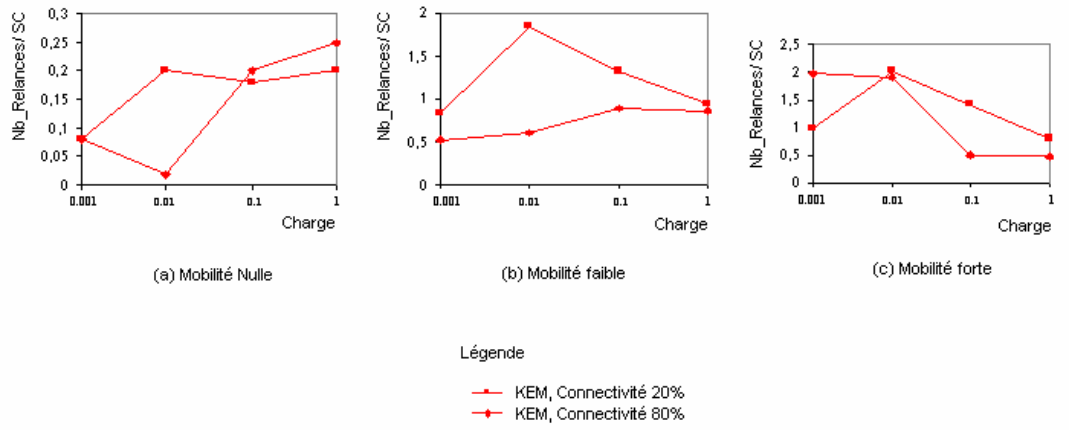


Figure 31 : *Nombre moyen de relances par entrée en SC.*

Analyse des résultats

En général, les courbes obtenues des mesures relevées pour toutes les métriques (sauf le nombre moyen de relances pour une entrée en SC) ont une allure décroissante, ce qui révèle la corrélation existant entre ces métriques. La lecture de ces résultats permet de dire que plus forte est la charge, meilleures sont les performances. Ce qui peut être expliqué par l'apport considérable des solutions d'optimisation apportées par le protocole à savoir: l'utilisation de différents types de jetons, l'optimisation du chemin du jeton, l'utilisation des files temporaires, l'exploitation des jetons en cours de circulation, etc... Toutes ces solutions sont fortement sollicitées et mises en œuvre au fur et à mesure que la charge augmente.

Par ailleurs, nous constatons que les performances dans le cas de mobilité nulle 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 *LinkInfo ()* avec des risques plus élevés de perte de chemin (notamment celui du jeton) avec multiplication du nombre de nœuds isolés (partiellement), ce qui cause plus de relances et par conséquent des temps d'attente de plus en plus élevés. Néanmoins, les performances obtenues sont acceptables comparées à celles du protocole *KRL*.

7.2.2. Résultats se rapportant à la variation du nombre de jetons

Afin de montrer l'influence de la variation des jetons sur les performances de notre protocole, nous avons pris successivement *8*, *5*, *3* et *1* jeton (s). Nous avons construis des graphes pour chacune des métriques à évaluer dans les trois cas de mobilité (nulle, faible et forte mobilité) sans omettre de représenter les courbes issues des simulations du protocole *KRL* sachant que seuls les systèmes à *5, 3* et *1* jeton (s) sont présents.

Vu l'écart étroit observé sur les valeurs des mesures effectuées dans le cas de faible connectivité (*20%*) et forte connectivité (*80%*), nous avons représenté seulement les courbes issues des mesures avec une connectivité de *20%* afin de ne pas nuire à la lisibilité des graphes.

Nombre moyen de messages par entrée en SC (Nb_Msg/ SC)

Comme dans le protocole *KRL* (Figure 32), le nombre de messages par entrée en SC est inversement proportionnel au nombre de jetons ce qui est prévisible. Néanmoins, il existe des cas de chevauchement.

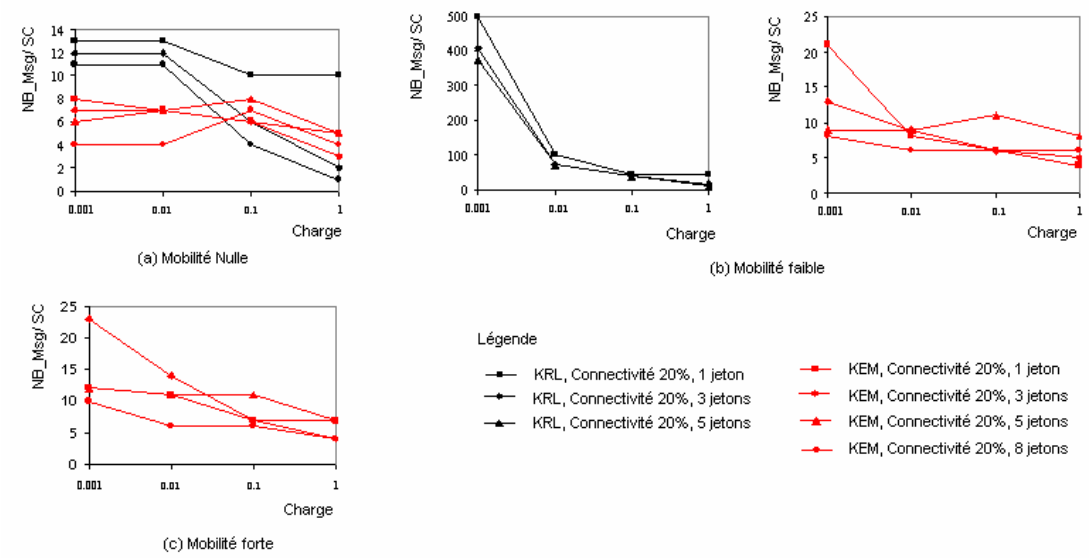


Figure 32 : *Nombre moyen de messages par entrée en SC.*

Temps d'attente moyen pour une entrée en SC (Tatt/ SC)

En général, le résultat attendu est tel que le temps d'attente est inversement proportionnel au nombre de jetons; mais dans certains cas (Figure 33) nous observons des chevauchements. Par exemple, le cas de faible mobilité avec une charge de *1* et *1000*; ou dans un autre cas plus apparent, celui de la forte mobilité où l'on observe que les temps d'attentes soulignés dans un système à *1* jeton sont meilleurs que ceux à *3* jetons et dans les deux derniers cas, les résultats sont meilleurs que ceux du système à *5* jetons dans des conditions de forte charge.

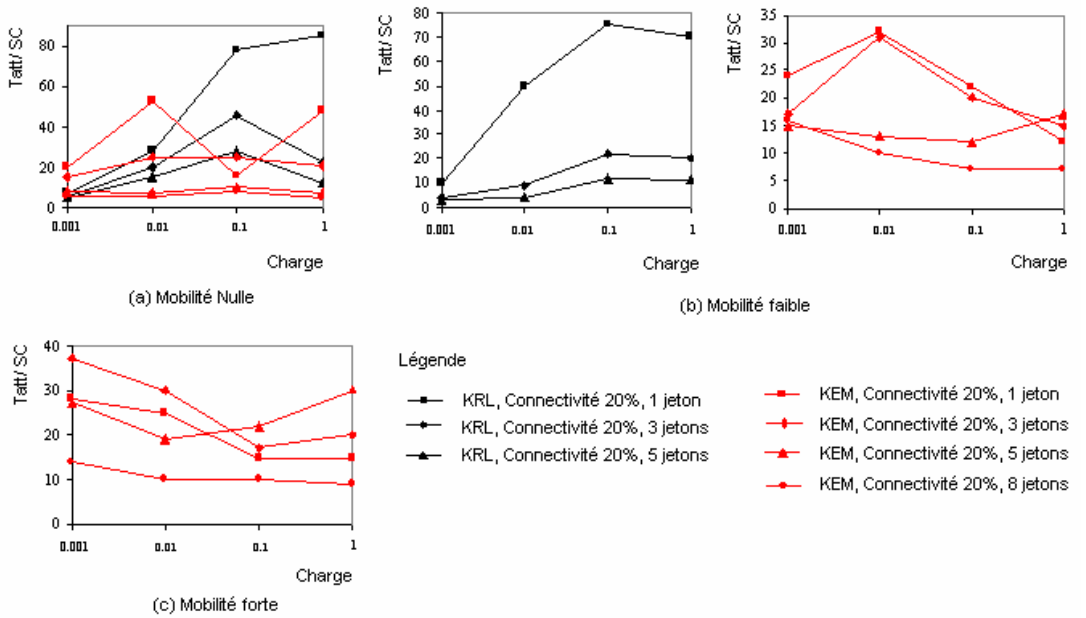


Figure 33 : *Temps moyen d'attente par entrée en SC.*

Délai moyen de synchronisation (DélaiSynch)

Les valeurs des mesures effectuées pour cette métrique appartiennent à un intervalle de valeurs réduites (Figure 34(a)). Dans tous les cas de mobilité, les graphes obtenus ont la même allure décroissante. Les cas de *1* et *3* jetons offrent relativement de faibles

performances dans les cas de faible charge et de très bonnes performances dans les cas de forte charge voir même des performances qui dépassent celles des cas où le système dispose de *5* et *8* jetons.

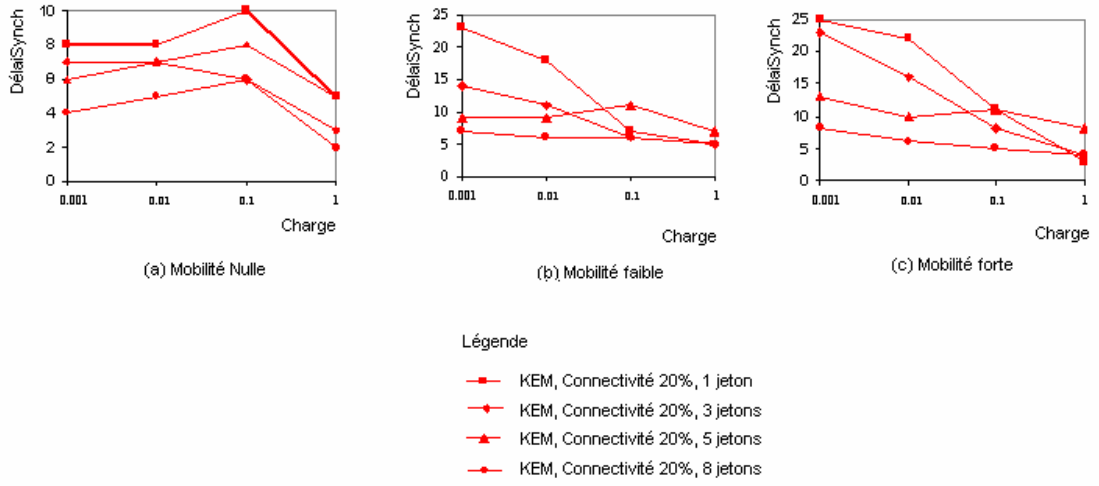


Figure 34 : *Délai moyen de synchronisation.*

Nombre moyen de relances par entrée en SC (Nb_Relances/ SC)

Dans le cas de mobilité nulle (Figure 35(a)), le nombre de relances est négligeable lorsque le nombre de jetons est de *3*, *5* et *8*. Dans le cas de *1* jetons le nombre de relances est plus important, ce qui est prévisible. Dans les deux Figures 35(b) et 35(c), nous constatons que plus le nombre de jetons augmente plus le nombre de relances diminue, ce qui est prévisible. En outre, il est à noter que les mesures de relances dans les cas de forte charge sont meilleures en général que dans les cas de faible charge.

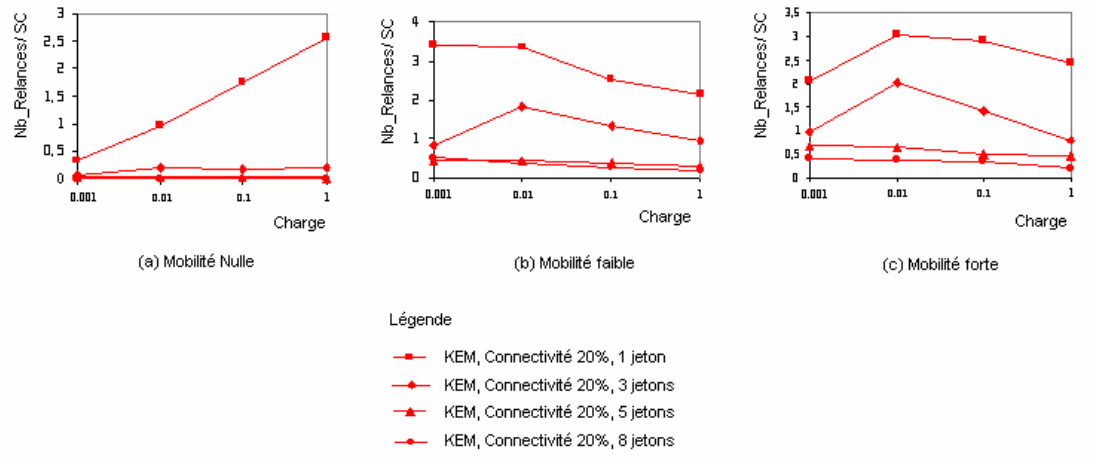


Figure 35 : *Nombre moyen de relances par entrée en SC.*

Analyse des résultats

Les courbes obtenues des différentes simulations montrent clairement que les performances sont meilleures lorsque le nombre de jetons augmente. Le nombre de relances (Figure 35) est plus important lorsque le nombre de jetons diminue. L'observation des courbes issues des mesures sur le nombre moyen de messages par entrée en SC pour le cas de faible mobilité

montre que les performances dans le cas de faible charge sont meilleures lorsque le nombre de jetons est de plus en plus grand. Plus la charge augmente, plus les courbes ont l'air de se rapprocher pour donner des valeurs de performances voisines. L'influence de la variation du nombre de jetons est plus évidente dans le graphe des temps d'attente par entrée en SC en fonction de la charge, où plus le système dispose de jetons moins un nœud attend pour entrer en SC, ce qui est prévisible. Dans les cas de *5* et *8* jetons, les performances sont bonnes de part le fait que le système et fortement actif avec disponibilité suffisante de jetons.

Contrairement à nos prévisions, le cas où le système dispose d'un seul jetons a donné des performances plus fortes et dans beaucoup de situations, meilleures que les cas de *3*, *5* et *8* jetons. Ceci peut s'expliquer comme suit :

- Le temps de relances pour le cas d'un seul jeton peut être très réduit, ainsi que le temps de séjour dans les files d'attente, ce qui réduit considérablement les messages *LinkInfo ()*, contrairement aux autres cas, ce qui peut aussi expliquer que le nombre de relances dans le cas de *1* jeton est plus grand que celui de *3*, *5* et *8* jetons.
- Le nombre de nœuds qui craignent l'isolement futur est réduit quand le nombre de jetons est *1*. Par conséquent, le nombre de messages de cession de jetons est très réduit.

Les courbes du graphe donnant le délai moyen de synchronisation pour une entrée en SC montrent le même comportement en général.

Comparées aux mesures effectuées dans le cas de mobilité nulle, celles du cas dynamique avec une faible mobilité montrent des performances légèrement inférieures et celles du cas dynamique avec une forte mobilité sont encore inférieures sans être dégradée. Ceci est prévisible.

7.3. Discussion

Les courbes représentant chaque métrique: délai d'attente, nombre de messages et délai de synchronisation en fonction de la charge ont en général une allure plutôt décroissante pour tous les cas de mobilité. Ceci montre que notre protocole réagit de manière efficace dans les conditions de forte charge où les solutions d'optimisation proposées pour ce protocole (utilisation de jetons *Collector*, la cession de jeton en cas de crainte d'isolement, exploitation de jetons Single en cours de transit, …etc.) sont fortement sollicitées et mises en œuvre de manière intense.

L'augmentation du nombre de jetons est en faveur du protocole en termes de performances. Cependant, certaines situations présentent des chevauchements entre les courbes contrairement à nos prévisions. En général, l'ajustement des délais des relances est de rigueur car son influence n'est pas négligeable. Comme dans *KRL*, la variation de la connectivité ne semble pas avoir une grande influence sur les performances de notre protocole. En effet, les écarts entre les valeurs des mesures prises respectivement pour la connectivité *20%* et *80%* ne sont pas très grands pour les métriques délai d'attente, nombre des messages et délai de synchronisation et pour tous les cas de mobilité. Dans le cas de forte mobilité, nous constatons une légère baisse des performances comparée à celles du cas de faible mobilité. Ces dernières étant moins bonnes que celles du cas de mobilité nulle; ce qui est prévisible.

Comme il a était évoqué dans les Sections précédentes, la comparaison avec *KRL* n'était possible que dans certains cas. D'après les graphes obtenus des différentes simulations, il semble que les résultats offerts par notre protocole sont meilleurs que ceux de *KRL*. En effet, les courbes issues de nos simulations sont en général inférieures à celles des courbes issues

des simulations de *KRL*.

D'autres part, avec les choix de décisions considérés dans la simulation, les résultats actuels de performances intrinsèques à notre protocole sont acceptables. Néanmoins, étant donné que le protocole est conçu de manière paramétrée, d'autres choix de décisions avec de nouvelles simulations permettront de mieux se rendre compte des performances du protocole face à des situations variées. A titre d'exemples nous citons: La variation du nombre de nœuds, la variation dynamique de la taille de file en fonction des informations collectées, etc.

8. Conclusion

Dans ce chapitre, nous avons présenté un nouveau protocole de *k*- exclusion mutuelle dans les réseaux mobiles ad hoc. Ce protocole s'écarte de toutes les solutions qui le précèdent: Il ne généralise ni n'adapte aucune autre solution, il est indépendant de toute topologie et est paramétré en fonction de l'état du système afin d'interagir avec tous les mouvements qui peuvent survenir. Il est orienté demande de jeton et suppose l'existence de *k* jetons dans le système. Face aux contraintes imposées par l'environnement mobile ad hoc, plusieurs techniques ont été proposées. Celles-ci sont sollicitées et mises en œuvre par le système chaque fois qu'il est nécessaire dans le but de favoriser la sûreté et la vivacité du protocole. Parmi ces techniques, nous citons: les relances en cas d'expiration du délai d'attente du jeton, l'utilisation de la file d'attente temporaire dans l'*espoir* de traiter les requêtes ayant perdu le chemin du jeton correspondant, l'utilisation de jetons de type *Collector* pour favoriser la satisfaction d'un grand nombre de requêtes tout en activant des jetons oisifs, l'utilisation des jetons en cours de transit (avec certaines restrictions), la cession de jetons afin d'éviter leur perte même momentanée, la collecte d'informations pour construire une connaissance locale du système sans aucun coût, …etc. En outre la participation des nœuds non intéressés par la SC dans le service de *k*- exclusion mutuelle n'est que partielle.

En général, le protocole que nous avons conçu offre ses meilleures performances dans les conditions de forte charge comparées à celles de faibles charges. Néanmoins, dans les conditions de faible charge, il réagit de manière efficace surtout lorsque les jetons sont en nombre suffisant. La variation de la connectivité ne semble pas avoir une grande influence sur les performances du protocole. Cependant, l'observation des états d'avancement des différentes simulations concernant des cas de forte connectivité (*80%*), montre une lenteur de calcul essentiellement à la réception ou à l'émission de requêtes. En effet, les nombres premiers étant très élevés, le coût du calcul de leur produit ou décomposition est important; ce qui nécessite une grande puissance de calcul.

D'après les fichiers résultats générés par les différentes simulations, notre protocole offre une certaine équité quant au partage des ressources. En effet, sur toute une durée de simulation, les processus exécutent relativement chacun, le même nombre de SC avec un nombre négligeable de relances.

La comparaison avec le protocole *KRL* a été possible seulement dans le cas de mobilité nulle et le cas mobile avec une faible mobilité en prenant comme critères, le nombre moyen de messages générés par entrée en SC et la durée d'attente moyenne pour une entrée en SC. D'après les graphes obtenus des différentes simulations, il semble que les résultats offerts par notre protocole sont meilleurs que ceux du protocole *KRL* relativement aux métriques communes.

Chapitre VI

La *h-k* exclusion mutuelle distribuée: Un état de l'art

1. Introduction

L'allocation de ressources est l'une des fonctions essentielles dans tout système informatique. En effet, chaque processus s'exécutant au niveau d'une machine donnée nécessite l'acquisition de ressources. Etant donnée que celles-ci sont en nombres limités et beaucoup d'entre elles ne tolèrent pas l'accès simultané, le contrôle d'acquisition s'avère alors vital. Le problème de base est connu par l'exclusion mutuelle qui consiste à gérer l'accès à une ressource commune et exclusive. Quand la ressource existe en *k* exemplaires, il s'agit du problème de la *k*- exclusion mutuelle dans lequel chaque processus ne peut utiliser qu'une seule ressource à la fois; donc, au maximum *k* processus exécutent en même moment leurs SC . Cependant, un processus peut vouloir demander *h* (*h* <=*k*) exemplaires de ressources ; il s'agit du problème de la *h parmi k exclusion mutuelle* (ou *h*-*k* EM). Le nombre total de ressources alloué au même moment aux différents processus ne doit pas dépasser *k*. La *h* parmi *k* EM est alors une généralisation des deux problèmes précédents. En effet si *h=1*, on retrouve le problème de la *k*- EM et si de plus *k=1*, on retrouve l'exclusion mutuelle. Par conséquent, ces trois problèmes sont étroitement liés.

Dans le but de résoudre le problème de la *h* parmi *k* EM, plusieurs solutions ont été proposées dans la littérature des systèmes distribués. Le premier protocole est du à M. Raynal [111]. Ce protocole requière à un nœud de récolter *k* permissions afin d'utiliser les *h* ressources demandées (ou par abus de langage, d'accéder à sa SC). Toutes les autres solutions utilisent le concept de structures de quorums, sont du type M. Meakawa [80], engendrent un coût de communication bas et sont souvent tolérantes aux pannes. Dans [9], R. Baldoni a développé une solution se basant sur le concept d'ensembles d'arbitres. Cette solution souffre du problème d'augmentation des ensembles avec l'augmentation du nombre *k*. Dans [85], Y. Manabe et *al* ont proposé un schéma général se basant sur les *k*- *ar*bitres, deux types de *k*-arbitres ont été définis, à savoir les *k*- arbitres symétriques (qui présentent des propriétés de disponibilité et d'équitabilité) et non dominés qui sont tolérants aux pannes. Dans [69], H. Kakugawa et M. Yamashita ont utilisé le concept de coterie local. Cette solution permet d'accéder à des ressources nommées. Contrairement aux autres protocoles connus, Y. Manabe et N. Tajima [86] ont introduit le concept de *(h, k)- arbitres* en en définissant deux types : un

(h, k)- arbitre uniforme et un *(h, k)*- arbitre *(k+1)* cube. J. R. Jiang [56] améliore son ancienne solution [56] (quoique tolérante aux pannes), qui utilise le concept de *k*- coterie, par l'utilisation du concept de *k*- *coterie de cohortes* ; ce qui la rend parmi les meilleures solutions utilisant les *k*- coteries en terme de haute disponibilité. Les solutions à base de *k*- arbitres bénéficient d'une grande tolérance aux fautes et d'un faible coût de messages de communication. Cependant, à cause de leurs propriétés, les *k*- arbitres sont difficiles à constituer. Y-C. Kuo a définit alors deux opérations de composition. L'opération de jointure des *k*- arbitres [75] permet la construction de nouveaux *k*- arbitres à partir des *k*- arbitres connus. L'opération de composition [74] permet de construire des *k*- arbitres à partir des coteries et/ou des *k*- coteries.

La majorité des solutions utilisant les quorums ont le même principe [80]: Pour réaliser une SC, un nœud doit recevoir une autorisation de tous les membres du quorum sélectionné. La différence entre ces différentes solutions réside dans la manière de construire ces quorums.

Dans les réseaux mobiles ad hoc, deux solutions ont été développées. La première est due à J-R. Jiang [55] qui est une adaptation du protocole d'exclusion mutuelle *RL* [150] (voir Chapitre II Section 3.3) pour la *h* parmi *k* exclusion mutuelle; ce qui lui permet d'hériter de ses principaux mécanismes. La deuxième solution est due à C-Z. Yang [152]. Cette solution utilise *k* jetons et s'inspire des algorithmes d'exclusion mutuelle de R. Baldoni et *al* [12] et J. Walter et al [150]. Néanmoins, ces solutions ne sont pas tolérantes aux pannes et au partitionnements.

Ce chapitre est dédié à la présentation des solutions sues citées. Une simple comparaison est alors réalisée sur la base d'un tableau (Table 5) qui récapitule les éléments importants concernant ces protocoles. Ce chapitre est structuré de la manière suivante : La Section 2 fait le tour d'horizon sur les solutions les plus connus de la *h* parmi *k* EM dans les systèmes distribués. La Section 3 expose plus ou moins en détail les deux seules solutions de *h* parmi *k* EM connue dans les réseaux mobiles ad hoc. Finalement, la Section 4 conclue le chapitre.

2. Solutions classiques de *h* parmi *k* EM dans les systèmes distribués

2.1. Solution de Raynal

Dans [111], M. Raynal a proposé la première solution au problème de *h* parmi *k* exclusion mutuelle pour les systèmes distribués. Cette solution, simple d'implémentation, est une généralisation de la solution d'exclusion mutuelle de G. Ricart et A. K. Agrawala [117]. Elle est donc orientée *permission individuelle avec information d'état* et avec *diffusion totale*. Cette solution se base sur une connaissance locale dont dispose chaque processus sur l'état l'allocation de ressources d'autres processus. L'idée de base de cette solution est comme suit: Soit un processus *i* qui demande m_i ressources et soit $used_i[j]$ la vue locale de *i* sur le nombre de ressources utilisées par chacun des autres processus *j*. Si elle est *supérieure ou égale* au nombre de ressources actuellement utilisées par *j*, alors si la condition suivante est assurée:

$$\sum_{1<=j<>i<=n} used_i[j] + m_i <= k$$

la sûreté est assurée et donc le processus *i* peut être satisfait. La propriété de vivacité est assurée en créant un ordre total sur les requêtes des processus en utilisant le mécanisme de datation des requêtes de L. Lamport [76]. La solution est construite en se basant sur ces deux principes et en considérant comme squelette la solution développée dans [117]. Un processus *i* désirant utiliser des ressources ne connaît pas naturellement le nombre exact de ressources

utilisées par un autre processus *j* ; ce nombre est compris entre *0* et *k*. Par conséquent, il augmente les variables $used_i[j]$ de *k* (dans le but d'avoir une valeur supérieur ou égale à la valeur courante) puis diffuse une requête estampillée (mais sans indication sur le nombre de ressources demandées!) à tous les autres processus. Quand un processus *i* reçoit une requête de *j*, il lui retourne un message *free(k)* s'il n'est pas intéressé par les ressources ou si sa requête est moins prioritaire que celle de *j*; (ce message indique, du point de vue de *i*, que *j* peut utiliser toutes les ressources du système). Autrement, le processus *i* retourne à *j* un message $free(k - m_i)$ (où m_i est le nombre de ressources demandées ou utilisées par *i*); (ce message indique, du point de vue de *i*, que *j* peut utiliser $(k – m_i)$ ressources du système). De plus, *i* mémorise *j* dans un ensemble $delayed_i$, ce qui lui permet d'envoyer à *j* et à tous les processus de cet ensemble un message $free(m_i)$ dès qu'il termine d'utiliser ces m_i ressources. Dans le cas particulier où $j \in delayed_i$ lorsque *i* reçoit une requête de *j*, il lui retourne immédiatement le message *free(k)*. Remarquons que dans ce cas *j* est informé du nombre de ressources que *i* est entrain d'utiliser. Cette information lui parvenait à la réception de la réponse à sa requête précédente où la requête de *i* était plus prioritaire que celle de *j*.

Le protocole suppose que chaque processus peut communiquer directement avec tout autre processus. Les canaux sont supposés FiFo et fiables. Le nombre de messages engendré par SC est situé entre *2(N-1)* et *3(N-1)*.

2.2. Solution de R. Baldoni

Dans [9], R. Baldoni a introduit le concept d'ensembles d'arbitres pour permettre le partage de *k* ressources identiques en supposant que chaque processus peut en demander *h*<=*k*. L'ensemble des processus est structuré en ensemble d'ensembles d'arbitres dont chaque *k* ensembles d'arbitres est en intersection non vide. A chaque processus est associé un ensemble d'arbitres. Ce processus est amené à récolter une permission de chacun des membres de cet ensemble lorsqu'il désire accéder à sa SC. Chaque nœud entretien une connaissance sur le nombre de ressources disponibles selon son point de vue, initialement ce nombre est à *k*. Lorsqu'il accorde sa permission à *h* ressources, il diminue ce nombre de *h*.

La sûreté du service est assurée étant donnée, d'une part, que chaque *k* ensembles d'arbitres sont en intersection non vide et d'autre part, chaque processus n'accorde sa permission que si le nombre de ressources disponibles (selon son point de vue) est suffisant pour la requête en cours. Cet algorithme engendre une complexité de *O(q)* où *q* est la taille des ensembles d'arbitres. L'auteur montre que tous les ensembles d'arbitres ont une taille inférieure à $O(n^{k/k+1})$ si tous les ensembles d'arbitres ont la même taille et chaque processus apparaît dans le même nombre d'ensembles d'arbitres. La méthode introduite pour construire les ensembles d'arbitres souffre de la grande taille avec l'augmentation du nombre *k*.

2.3. Solution de Y. Manabe et *al*

Dans [85], Y. Manabe et *al* ont proposé un schéma général (ou protocole) basé quorums construits selon la notion de *k*- arbitre (qui formalise le concept d'ensemble d'arbitres) afin de permettre l'accès à *k* ressources au maximum concurremment (sachant que chaque processus peut en demander *h* ressources/ *h*<=*k*). Le protocole présenté dans [85] est une extension de celui de Maekawa [80] en se basant sur l'idée définie par M. Raynal et *al* dans [111]. Dans [85], deux types de *k*- arbitres ont également été invertigués, à savoir les *k*- *arbitres symétriques* et *non dominés*. Le premier type possède des propriétés intéressantes permettant la disponibilité des quorums et l'équitabilité en terme d'égalité de nombre de messages

échangés pour chaque requête. Le second, dont la définition est équivalente à celle donnée au Chapitre IV, permet une bonne protection contre les pannes et les partitionnements.

Concept de k – arbitre

Un ensemble de quorums *C* est un *k*- arbitre sous *U* si et seulement si les propriétés suivantes sont vérifiées :

- Intersection : $\forall k+1$ *quorums* $Q_1, Q_2, \ldots, Q_{k+1} \in C$, $\bigcap_{i=1}^{k+1} Q_i \neq \emptyset$
- Minimalité : $\forall Q_i, Q_j$ éléments de C, $Q_i \not\subseteq Q_j$.

La propriété d'intersection des *k*- arbitres est une condition nécessaire et suffisante pour assurer la propriété de sûreté dans le protocole de *h* parmi *k* exclusion mutuelle de Y. Manabe et *al* [85]. En effet, les processus en intersection entre *(k+1)* quorums permettent d'éviter les conflits entres les différents processus demandeurs de ressources.

Concept de k– arbitre symétrique

Cette notion permet de créer en plus une certaine distribution ''équitable de charge en processus'' quant à la complexité en messages.
Un *k*- arbitre *C* sous *U* est symétrique si et seulement si les propriétés suivantes sont vérifiées :

- Effort équitable : $\forall p_i \in U, /\{j / i \in Q_j/ = \beta$. Autrement dit, tous les processus sont contenus dans un même nombre de quorumsβ.
- Taille égale : $\forall Q_i, /Q_i/ = \gamma$. Autrement dit, tous les quorums sont de même tailleγ.

L'algorithme

Dans [85], quand un processus *i* désire utiliser des ressources, il sélectionne un quorum *Q* et lui envoie un message de requête pour *h* ressources à chacun des membres de ce quorum y compris soit même, et attend la réception de réponses de chacun des membres de ce quorum auquel cas il accède à sa SC. A la sortie de la SC, un message de libération comprenant les *h* ressources est envoyé à chacun des membres de *Q*. Quand un processus *i* reçoit une requête de *j*, il lui retourne un message de permission si la condition suivante est satisfaite: $\sum_{r \in U} c_i[r] + h <= k$ où $c_i[r]$ est le nombre de ressources utilisée actuellement par le processus *r* du point de vue de *i* (*r* est le processus auquel *i* a déjà envoyé une permission et duquel il n'a pas encore reçu de réponse) ; de plus $c_i[j]$ est affecté de *h*. Dans le cas où cette condition n'est pas vérifiée, la requête de *j* est insérée dans une file d'attente. A la réception d'un message de libération de ressources de *j*, *i* affecte $c_i[j]$de *0* et de plus, il envoie un message de permission à chacun des processus dans sa file pour lesquels la condition (déjà décrite) est vérifiée.

Dans ce protocole, chaque requête nécessite *3/Q/* messages pour être servie. Bien entendu, les actions décrites précédemment ne suffissent pas pour éviter la famine et l'interblocage. Le problème de famine est résolu en créant un ordre total sur les requête à l'aide de l'estampillage de Lamport [76]. L'interblocage peut se produire à cause de l'imprévisibilité des délais de communication où des processus peuvent recevoir des requêtes dans un ordre arbitraire. Ce qui les mène à envoyer dans cet ordre des permissions aux différents processus, ce qui engendre l'interblocage du système puisque aucun message ne peut avoir la totalité des permissions requises. Le protocole ne prévoit pas de mécanismes pour éviter cette situation. Des mécanismes peuvent être introduits [9, 33, 84, 112] mais avec conséquence l'augmentation de la complexité de messages par SC (*5/Q/* dans le pire des cas [3, 84]).

2.4. Solution de H. Kakugawa et M. Yamashita

Dans [69], H. Kakugawa et M. Yamashita ont introduit le concept de *coteries locales* pour résoudre le problème de la *h* parmi *k* exclusion mutuelle. Un algorithme simple de construction d'une coterie locale est alors proposé. Leur protocole de résolution du problème de la *h* parmi *k* exclusion mutuelle permet aux processus d'accéder à des ressources nommées, contrairement aux autres protocoles définis pour ce problème. Ce protocole est très utile lorsque chaque processus est intéressé par telles ressources plutôt que telles autres, quoique toutes ces ressources soient banalisées. Supposons par exemple le cas d'un bâtiment où des imprimantes identiques sont disposées chacune au niveau d'un étage. Dans ce cas et afin de limiter les déplacements des utilisateurs, chacun qui est situé au niveau de l'étage *i* est amené a utiliser, par exemple, l'imprimante de l'étage *(i-1)* ou *i* ou *(i+1)*. Un autre problème classique dans ce contexte est celui connu par *''the drinking philosophers''* défini dans [33] et étudié dans [13, 40]. Les auteurs ont discuté la possibilité d'extension de leur solution pour permettre à un processus de donner une liste de ressources à utiliser selon un ordre de priorité donné.

Concept de coterie locale

Soient :

$U=\{u_1, u_2, ..., u_n\}$, l'ensemble de tous les processus du système

et $R=\{r_1, r_2, ...r_m\}$, l'ensemble des ressources partagées par ces processus.

L'ensemble des ressources accessibles par un processus $u \in U$ est désigné par $\alpha(u)$ (ensemble supposé non vide), ce qui définit une fonction : $\alpha : U \rightarrow 2^R$. Le nombre de ressources utilisé par *u* à tout instant est compris entre *0* et $/\alpha(u)/$. L'ensemble des ressources utilisé par *u* à un moment donné est noté par $\rho(u)$. Le triplet (U, R, α) est appelé *la structure de partage*.

Un ensemble $\{Q_u / u \in U\}$ est appelée une coterie locale si les conditions suivantes sont vérifiées:

- Non-vide: $\forall u \in U, [Q_u \neq \emptyset]$.
- Propriété d'intersection: $\forall u, v \in U, \alpha(u) \cap \alpha(v) \neq \emptyset \Rightarrow \forall q \in Q_u, \forall r \in Q_v [q \cap r \neq \emptyset]$.
- Propriété de minimalité : $\forall u \in U, \forall q,r \in Q_u [q \text{ non } \subseteq r]$.

Notons que si $/R/=1$ *et* $\alpha(u)=R \ \forall u \in U$, une coterie locale devient une coterie. Remarquons aussi que si des processus *u* et *v* ne partagent pas des ressources, des quorums de *u* et *v* en intersection vide peuvent être construits et donc l'allocation de ressources pour *u* et *v* se fait de manière indépendante.

Dans la solution de [69], les processus maintiennent une base de données distribuée qui consiste en paires de *processus et ressource qu'ils utilisent*. Un processus *u* qui désire utiliser *h* ressources $\in \alpha(u)$ envoie une requête qui demande si les *k* ressources sont disponibles pour *u*. Les auteurs utilisent les coteries locales pour implémenter cette base de données. Partant du principe qu'un processus *v* est (partiellement) responsable des ressources *r* accessibles par un processus *w*, s'il appartient à Q_w, si cette ressource est en cours d'utilisation par *w*; cela signifie que celui-ci a obtenu la permission de chaque processus *v* dans un quorum *q* de Q_w. Par conséquent, *v* a mémorisé dans sa base de données locale le fait que *w* utilise *r*.

L'algorithme

Un processus désirant accéder à h ressources sélectionne un quorum arbitraire $q \in Q_u$ et envoie une demande à chacun de ses éléments. Chaque processus recevant cette requête, retourne à son émetteur les noms de toutes les ressources disponibles selon sa base locale. Le processus u, en recevant toutes les réponses attendues, sélectionne h ressources parmi toutes les ressources communes des listes reçues et envoie un message de verrouillage accompagné des h noms des ressources à chaque élément de q pour leur permettre de mettre à jour l'état de ces ressources. A la libération de ressources, un message de libération est envoyé à ces mêmes processus afin qu'ils libèrent ces ressources. Des mécanismes ont été introduits (, telle que la datation des requêtes [76],) pour éviter l'interblocage, notamment si un processus n'obtient pas toutes les ressources demandées.

Chaque processus u entretient deux ensembles; S_u, qui est constitué de l'ensemble des processus w tel que au moins un quorum de w contient u; R_u, qui est l'ensemble des ressources accessibles par chaque processus dans S_u.

Un processus u désirant accéder à h $(<=/\alpha(u)/)$ ressources, il sélectionne arbitrairement un quorum $q \in Q_u$ et envoie une simple requête (*Query*) (ne contenant pas le nombre de ressources demandées) estampillée à chacun de ses membres. Ce processus accède aux ressources demandées s'il reçoit un message de réponse (*Response*) de chacun des processus de q et l'ensemble des ressources demandées est inclus dans l'intersection des ressources libres communes à tous les éléments de q. Avant d'accéder à ces ressources, u envoie un message estampillé de verrouillage de ces ressources (*Lock*) à chaque élément de q. A la libération de ressources, u envoie un message estampillé de déverrouillage de ces ressources (*Unlock*) à tous les éléments de q.

Un processus v recevant un message *Query* de u, lui retourne immédiatement un message *Response* contenant l'état de chaque ressource selon son point de vue, s'il n'attend aucun message de verrouillage. Dans le cas où il attend un message de verrouillage de w, si la requête de w est plus ancienne que celle u, la requête de u est enfilée localement; autrement, dans le but de retirer la droit de verrouillage à w, v envoie à w un message de préemption *Preempt* (en enfilant la requête de u localement) si un message dans ce sens n'est pas en cours, et attend la réception soit d'un message de retour (*Return*) ou un message *Lock* de la part de w.

La réception d'un message *Return* issu de w au niveau d'un processus v lui permet d'envoyer au processus en tête de sa file locale (i.e. ayant la requête la plus prioritaire) un message *Response*. La réception d'un message *Lock* permet à un processus de mettre à jour sa base de données et d'envoyer un message *Response* au processus en tête de file locale, s'il y a lieu. La réception d'un message *Unlock* par v issue de w permet à v de mettre à jour sa base de données et d'envoyer un message *Response* au processus duquel il attend un message *Lock* (pour lui permettre de garder une nouvelle version sur l'état des ressources), s'il y a lieu; sinon il l'envoie le message *Response* au premier de sa file locale, si elle n'est pas vide. La réception d'un message *Preempt* par w issue de v est sans effet si w a déjà envoyé un message *Unlock*; autrement, un message *Return* est retourné à v et par conséquent la copie de réponse reçue dernièrement de v est à supprimer et donc, w attend un nouveau message *Response* de v.

Le protocole de Kakugawa et Yamashita engendre dans le meilleur cas $/q/$ messages où $/q/$ est la taille du plus petit quorum. C'est le cas où seulement les messages *Query*, *Response*, *Lock* et *Unlock* sont utilisés pour chaque processus. Dans le pire des cas, $(7+ /\alpha(u)/)*/q/$ où $q \in Q_u$ pour un processus u; c'est le cas où les autres types de messages sont nécessaires.

2.5. Solution de Y. Manabe et N. Tajima

Dans [86], Y. Manabe et N. Tajima ont introduit le concept de *(h, k)- arbitres* pour résoudre le problème de la *h* parmi *k* exclusion mutuelle. Il définissent deux *(h, k)*- arbitres : un *(h, k)*- arbitre uniforme et un *(h, k)*- arbitre *(k+1)* cube. La taille des quorums dans un *(h, k)*- arbitre ne dépasse pas celle des quorums dans un *k*- arbitre. Par conséquent, les *(h, k)*- arbitres sont plus efficaces que les *k*- arbitres. Les auteurs, dans [86], proposent une solution pour le problème de la *h* parmi *k* exclusion mutuelle basée quorum en utilisant le schéma des *(h, k)*-arbitres. Contrairement aux *k*- arbitres où un processus sélectionne toujours le même quorum, un *(h, k)*- arbitre est un ensemble de quorums pour chaque *h (1<= h <=k), {$Q_{h,k}$ / 1<= h <=k}* , où $Q_{h,k}$ est un ensemble de quorums. Un processus utilise un quorum dans $Q_{h,k}$ lorsqu'il souhaite utiliser *h* unités de ressources.

Concept de (h, k)- arbitre

Un *(h, k)*- arbitre est l'ensemble {$Q_{h,k}$ / 1<=h<=k} où $Q_{h,k}$ est un ensemble de quorums. La propriété d'intersection diffère de celle des *k*- arbitres, elle sera définie ci-dessous. La propriété de minimalité est la même que celle de la *1*- coterie.

Un certain nombre de définitions s'impose:

- Un *ensemble* est constitué d'éléments non répétés, tandis qu'un *panier* peut avoir des éléments répétés. Par exemple *{1, 1, 2, 3}* est un panier à 4 éléments.
- Un *schéma de requêtes* de *h* parmi *k* exclusion mutuelle r_k est un panier d'entiers positifs ne dépassant pas *k*. Un schéma de requêtes r_k est *conflictuel* si et seulement si $\sum_{h \in rk} h >= k+1$.
- Un schéma de requête conflictuel r_k est *critique* si et seulement si $\forall h \in r_k$, $rk- \{h\}$ est non conflictuel. Par exemple, *{4, 3, 3}* et *{1, 2, 2, 1}* sont deux schémas de requêtes conflictuels dont seul le second est critique.
- Pour un schéma de requêtes r_k, un panier $\{\exists q \in Q_{h,k} / h \in r_k\}$ est appelé une *affectation de quorum* pour r_k. Soit $QA(r_k)$ un ensemble de toutes les affectations de quorums pour r_k. Une affectation de quorum exprime le cas où un processus sélectionne un quorum et envoie une requête. Etant donné que $/q_{h,k} />=1$, on peut trouver plusieurs affectations de quorums.

La propriété d'intersection suivante assure la propriété de sûreté:

$\forall qa \in QA(cr_k), \bigcap_{q \in qa} q \neq \phi$ pour un schéma de requête conflictuel et critique cr_k.

Il est montré facilement dans [86] que pour chaque schéma de requêtes conflictuel r_k il existe un schéma de requêtes conflictuel et critique cr_k de telle sorte que $cr_k \subseteq r_k$. En se basant sur ce résultat, il est montré aussi qu'il est impossible que les requêtes de r_k utilisent en même moment une ressource partagée. Par conséquent, la propriété de sûreté est satisfaite en considérant tout schéma de requêtes conflictuel bien sur en ne considérant que les affectations de quorums vérifiant la propriété d'intersection ci-dessus.

L'algorithme

Un processus désirant utiliser *h* ressources, sélectionne un quorum $q_{h,k} \in Q_{h,k}$ et exécute la même procédure que celle utilisée pour les *k*- arbitres [85]. Un processus désirant accéder à *h* ressources sélectionne un quorum $q \in Q_{h,k}$ et envoie une requête estampillée selon les horloges de Lamport [76] à chacun de ses éléments. Lorsque ce processus reçoit une réponse

''*Ok*'' de chacun des éléments de *q*, il accède à ces *h* ressources. A la fin de l'utilisation de ces ressources, un message de libération ''*Release''* est envoyé à chaque élément de *q*.

Chaque processus dispose à tout instant de x_p permissions, initialement ce nombre est *k*, correspondant au nombre total de ressources. Chaque processus maintient une file de requêtes reçues organisée en fonction des estampilles des requêtes. Chaque requête est de la forme *(h, p, c)* où *h* est le nombre d'exemplaires de ressources demandées, *p* est le processus concerné et *c* l'estampille de la requête au sens de Lamport. De plus, à chaque requête résidant dans une file est affecté un attribut : ''*wait''*, ''*Ok''* ou *''cancel''*. Quand un processus *p* reçoit une requête *(h', p', c')*, il l'insère dans sa file locale selon sa priorité et lui affecte l'état *wait*. Si le nombre total de ressources demandées des requêtes prioritaires dans la file n'est pas supérieur à *k-h'* et $x_p >= h'$, *p* envoie un message ''*Ok''* à *p'*, change le statut de la requête à ''*Ok''*, et exécute $x_p := x_p - h'$.

L'insertion d'une requête dans une file peut engendrer la conséquence suivante: Il peut y avoir une requête *(h'', p'', c'')* dans la file avec le statut ''*Ok*'' et le nombre d'unités requises par les requêtes de hautes priorités est supérieur à *k-h''*. Plusieurs requêtes peuvent satisfaire cette condition. Afin d'éviter l'interblocage, le ''*Ok*'' de ces requêtes doit être avorté ; donc, *p* envoie un message ''*cancel*'' à *p''* et change le statut de ces requêtes à ''cancel''. Quand *p''* reçoit un message ''*cancel*'' de *p*, il lui retourne un message ''*cancelled*'' s'il n'a pas encore reçu tous les messages ''*Ok*'' attendus. Il se met alors en attente d'un autre message ''*Ok*'' de *p*. La réception du message ''*cancelled*'' au niveau de *p* lui permet de changer l'état de la requête *(h'', p'', c'')* à ''*wait*'', d'exécuter $x_p := x_p + h''$ et d'essayer d'envoyer le message ''*Ok*'' aux processus propriétaires des requêtes plus prioritaires que celle de *p''* en se basant sur la condition déjà indiquée pour l'envoi de message ''*Ok*''.

Le protocole de Y. Manabe et N. Tajima engendre dans le meilleur cas $/q_{h,k}/$ messages par requête, où $/q_{h,k}/$ est la taille du plus grand quorum dans $Q_{h,k}$. C'est le cas où la requête n'engendre pas de conflit avec les autres requêtes. Il s'agit donc d'envoyer un message de demande à chaque membre du quorum sélectionné, de recevoir une réponse de chacun de ces membres et de leur envoyer une libération à la fin de l'utilisation des ressources. Dans le pire des cas, le protocole génère $(3h+3)\ /q_{h,k}/$. C'est le cas où des conflits se produisent et des requêtes moins prioritaires doivent être avortées.

2.6. Solution de J. R. Jiang

Dans [56], J. R. Jiang a développé un protocole de *h* parmi *k* exclusion mutuelle qui se base sur le concept de *k- coterie de cohortes* (voir Chapitre IV). Cette solution est une amélioration de celle définie par le même auteur dans [58] qui utilise le concept de *k*- coterie. Dans son ancienne solution, un processus qui désire accéder à *h* ressources sélectionne aléatoirement *h* quorums disjoints deux à deux et envoie une requête à chacun de leurs membres. Ce processus accède à sa SC s'il reçoit une permission de chacun des membres auxquels une requête est adressée par ce processus. Quand un processus *j* reçoit une requête, il retourne une autorisation s'il ne l'a pas déjà attribué à un autre processus. La sûreté est assurée puisque d'une part, un processus ne donne sa permission qu'au plus un processus à la fois et d'autre part, il existe au plus *k* quorums disjoints deux à deux.

Cette solution est tolérante aux fautes, étant donné qu'un processus qui n'arrive pas à récolter suffisamment de permissions après un délai de garde (i.e. dans le cas de panne de nœuds), il sélectionne *h* autres quorums disjoints. Dans ce contexte, cette solution est meilleure que celles qui utilisent les arbitres. En effet, ces dernières ne sont pas tolérantes aux fautes, étant donné qu'un processus fait recours à la sélection de quorum une seule fois. Si un

nœud tombe en panne, ce processus ne peut pas récolter les permissions nécessaires à l'entrée en SC.

Cependant, deux problèmes apparaissent dans la solution de [58]: Premièrement, la solution ne donne pas le critère de choix des *h* quorums et deuxièmement, il est difficile de fixer la valeur du délai d'attente.

La solution développée dans [56] permet de remédier à ces problèmes: Elle n'utilise pas de délai d'attente ni fait recours à la re-sélection de *h* quorums. En échange, cette solution engendre un nombre de messages constant dans le meilleur cas et fait partie des meilleurs protocoles utilisant les *k*- coteries en terme de haute disponibilité.

2.7. Autres Solutions de *h* parmi *k* exclusion mutuelle

Les *k*- arbitres ont le bénéfice d'une grande tolérance aux fautes et un faible coût de messages de communication. Cependant, ils requièrent une intersection non vide entre les $(k+1)$ quorums dans chaque *k*- arbitre. Par conséquent, la construction des *k*- arbitres est difficile. Pour cela, des méthodes de construction de telles structures s'avèrent nécessaires. Les opérations de combinaisons des *k*- arbitres sont alors utilisées, à savoir l'opération de *jointure* et de *composition*. L'opération de jointure des *k*- arbitres permet la construction de nouveaux *k*- arbitres à partir des *k*- arbitres connus. L'opération de composition permet de construire des *k*- arbitres à partir des coteries, des *k*- coteries, cohorte- coterie et/ou coterie locale.

Opération de jointure de k- arbitres.

Initialement, l'opération de jointure est proposée par M. Neilsen et M. Mizuno [99]. Elle produit une nouvelle coterie plus large en réalisant la jointure de deux coteries connues. Y-C. Kuo, dans [75], étend l'opération de jointure à celle des *k*- arbitres pour produire un *k*- arbitre plus large et qui peuvent disposer (sous certaines conditions) d'une propriété de disponibilité et donc de tolérance aux fautes meilleure que celles des *k*- arbitres originaux.

Soient C_1 et C_2, deux *k*- arbitres sous respectivement U_1 et U_2, ou $U_1 \cap U_2 = \emptyset$.

Soient $x \in Q$, où $Q \in C_1$ et $U_3 = (U_1 - \{x\}) \cup U_2$. L'opération de jointure de deux *k*- arbitres notée $\oplus_x$ est définie par :

$C_1 \oplus_x C_2 = \{ CT_x(Q_1, Q_2) / Q_1 \in C_1, Q_2 \in C_2 \}$ où

$CT_x(Q_1, Q_2) = \{ (Q_1 - \{x\} \cup Q_2 \text{ si } x \in Q_1 \text{ ou } Q_1 \text{ sinon} \}$.

Autrement dit, $C_1 \oplus_x C_2$ est obtenue en remplaçant chaque occurrence de x dans les quorums de C_1 par les nœuds de tout quorum dans C_2, si $x \notin Q_1$, Q_1 est pris tel qu'il est. Par exemple, considérons les deux *2*- arbitres suivants :

$C_1 = \{\{1, 2, 3\}, \{1, 2, a\}, \{1, 3, a\}, \{2, 3, a\}\}$ sous $U_1 = \{1, 2, 3, a\}$ et
$C_2 = \{\{4, 5, 6\}, \{4, 5, 7\}, \{4, 6, 7\}, \{5, 6, 7\}\}$ sous $U_2 = \{4, 5, 6, 7\}$.
$C_1 \oplus_a C_2 = \{ \{1, 2 , 3\}, \{1, 2, 4, 5, 6\}, \{1, 2, 4, 5, 7\}, \{1, 2, 4, 6, 7\}, \{1, 2, 5, 6, 7\}, \{1, 3, 4, 5, 6\}, \{1, 3, 4, 5, 7\}, \{1, 3, 4, 6, 7\}, \{1, 3, 5, 6, 7\}, \{2, 3, 4, 5, 6\}, \{2, 3, 4, 5, 7\}, \{2, 3, 4, 6, 7\}, \{2, 3, 5, 6, 7\}\}$.

Opération de composition de k- arbitres

L'opération de composition définie par Y-C. Kuo dans [74] permet de construire des *k*- arbitres à bases de coteries ou de *k*- coteries. Par cette méthode, le problème de construction de *k*- arbitres est réduit au problème de construction de coteries ou de *k*- coteries.

Soient C_1 et C_2 deux ensembles de quorums sous respectivement U_1 et U_2 où $U_1 \cap U_2 = \emptyset$.
L'opération de composition de *k*- arbitre notée *o* est définie par :

$C_1 \, o \, C_2 = \{\cup \, Q_i^{(2)} / \, \forall Q^{(1)} \in C_1 \, et \, \forall \, i \in Q^{(1)}, \, Q_i^{(2)} \in C_2\}$.

Autrement dit, $C_1 \, o \, C_2$ est essentiellement identique à C_1, sauf que chaque nœud dans $C_1 \, o \, C_2$ est un nœud logique, qui est remplacé par les nœuds d'un quorum particulier dans C_2 [74].
Par exemple, considérons les deux ensembles de quorums :

$C_1 = \{\{1, 2\},\{1, 3\}, \{2, 3\}\}$ sous $U_1 = \{1, 2, 3\}$ et
$C_2 = \{\{a, b, c\}, \{a, b, d\},\{a, b, e\}, \{b, c, d\}, \{c, d, e\}\}$ sous $U_2 = \{a, b, c, d, e\}$.
$C_1 \, o \, C_2 = \{ \{a, b, c\} \cup \{a, b, d\}, \{a, b, c\} \cup \{a, b, e\}, \{a, b, d\} \cup \{a, b, e\}\}$
$= \{\{a, b, c, d\}, \{a, b, c, e\}, \{a, b, e, d\}\}$

Soit $C_3 = PROPER \, (C_1 \, o \, C_2) \, où \, PROPER(S)$ est une fonction qui élimine les éléments dans *S* tel qu'aucun élément dans le sous ensemble est *propre* de l'autre. Avec cette définition, Y-C. Kuo [74] montre que Si C_1 est une k_1- coterie sous U_1 et C_2 est un k_2- arbitre sous U_2, alors C_3 est un k_3- arbitre sous U_2, où $k_3 = 2k_2 - k_1 + 1$ et $k_1 <= 2k_2$. Il en découle aussi que si C_1 est une coterie sous U_1 et C_2 une coterie sous U_2, alors C_3 est un *2*- arbitre sous U_2. De cette manière, il est alors possible de construire un *k*- arbitre des coteries connues puis, récursivement, composer les *k*- arbitres avec d'autres *k*- coteries connues pour former d'autres *k*- arbitres avec des valeurs de *k* encore plus grands.

Approche	Outils &mécanismes	Autres	Protocoles	Objectif	Messages	Observations
Permission	Individuelle	Diffusion totale Information d'état	Raynal [111], 1991	Généralise Ricart-Agrawala [117]	*entre 2(N-1)* et *3(N-1)*	-Les processus communiquent directement -Estampillage de Lamport
	Arbitre	Ensembles d'arbitres	Baldoni [9], 1994		*O(q)* où *q taille d'un ensemble d'arbitres*	-Les processus communiques directement -Estampillage de Lamport
		k- arbitres	Manabe et *al*[85], 1998	Se base sur des idées dans [111] et étend Maekawa [80]	*3/Q/* pour réaliser une SC, où *Q* est un quorum.	- Définit : *k*- arbitres symétriques et non dominés - Estampillage de Lamport
		Coteries locales	Kakugawa- Yamashita [69], 1996	Permet d'accéder à des ressources nommées.	*/q/* messages où *q* est le quorum le plus petit. Dans le pire des cas, *(7+ /α(u)/)*/q/* où $q \in Q_u$ le quorum pour un processus *u et α(u),* l'ensemble des ressources accessibles par *u.*	-Les processus communiquent directement -Estampillage de Lamport
		(h, k)- arbitres	Manabe-Tajima [86], 2004	Pour chaque *h* ressources demandées, un quorum $q_{h,k}$ *est choisi.*	*Entre* $/q_{h,k}/$ *et (3h+3)* $/q_{h,k}/$ *où* $q_{h,k}$ *est le plus grand quorum appartenant à* $Q_{h,k}$. $Q_{h,k}$ *est un ensemble de quorums*	Définit : *(h, k)*- arbitre uniforme et *(h, k)*- arbitre *(k+1)* cube. - Taille d'un quorum est au plus celle d'un *k*- arbitre.
		k- coterie de cohortes	Jiang [56], 2004.	Améliore la solution du même auteur dans [58].		- *h* quorums sont choisis pour une requête. - Pas de re-sélection de quorum en cas de pannes de nœuds. - Résistance aux pannes de nœuds.
		k- arbitre	Kuo [74], 2001.	Construire de nouveau k- arbitres à partir des coteries et/ou des *k*- coteries, cohorte- coterie, coterie locale connus.		- Composition de k- arbitres. - Jointure de k- arbitres.

Table 5 : *Classement des solutions classiques de h parmi k EM distribuée.*

3. La *h* parmi *k* EM dans les réseaux mobiles ad hoc

A l'heure actuelle, deux solutions de *h* parmi *k* exclusion mutuelle ont été élaborées pour les réseaux mobiles ad hoc, en l'occurrence celle de J-R. Jiang [55] et celle C-Z. Yang [152]. Ces solutions seront décrites dans la suite de ce chapitre.

3.1. Solution de J-R. Jiang

Dans [55], J-R. Jiang a proposé un protocole de *h* parmi *k* exclusion mutuelle orienté jeton pour les réseaux mobiles ad hoc. Ce protocole est une adaptation du protocole d'exclusion mutuelle *RL* [150] (voir Chapitre II Section 3.3) pour la *h* parmi *k* exclusion mutuelle; ce qui lui permet d'hériter de ses principaux mécanismes. Le protocole de J-R. Jiang [55] utilise une structure de *DAG* de pointeurs orientés jeton mappées sur les liens physiques. Chaque nœud du réseau entretient seulement des informations liées à son voisinage immédiat. Chaque nœud détient une file d'attente des identités des nœuds voisins desquels des requêtes pour le jeton ont été reçues. Le protocole maintiens *un seul* jeton, initialement détenu par le nœud racine du *DAG*. Le nœud de plus basse élévation (i.e. nœud puits ou racine) est constamment le détenteur du jeton. Le protocole assure que tous les nœuds qui ne détiennent pas le jeton possèdent constamment un chemin au nœud détenteur du jeton. Autrement dit, chaque nœud non détenteur du jeton doit assurer continuellement l'existence d'au moins un voisin avec une élévation moindre puisque sa requête doit être envoyée à travers ce nœud (qui forme son lien sortant vers le détenteur du jeton). Dans le cas où ce nœud se trouve sans ce genre de voisin, il invoque la technique d'inversion de liens afin de lui assurer l'existence de ce noeud. Chaque nœud choisi dynamiquement son voisin de plus petite élévation comme nœud prochain préféré sur le chemin du détenteur du jeton pour lui envoyer sa requête. Toutes les requêtes, qui atteignent un nœud détenteur du jeton, sont traitées d'une manière symétrique et servies continuellement. La structure de *DAG* est réorganisée quand il est nécessaire (i.e. lors de déplacement de jeton ou lors de ruptures de liens physiques) et les requêtes bloquées sont re-routées. Une requête qui arrive au niveau d'un nœud en SC est immédiatement servie si ce noeud détient des ressources libres, ce qui permet d'augmenter la concurrence d'accès en SC. La différence essentielle avec le protocole *RL* [150] est liée à la gestion de l'allocation de plusieurs ressources à un même noeud. De ce fait, le protocole doit assurer qu'au plus *k* ressources peuvent être utilisées concurremment. Le jeton transporte une variable qui indique à tout moment le nombre de ressources libres. Ce protocole nécessite un message supplémentaire permettant le transport du nombre de ressources libérées à destination du nœud détenteur du jeton. Le nœud qui reçoit le jeton et qui est destinataire peut accéder à sa SC si le nombre de ressources est suffisant pour satisfaire sa requête. Autrement, il est amené à attendre la réception d'un message de libération dans l'espoir de récupérer les ressources manquantes. Une fois un nœud demandeur trouve le nombre requis de ressources, avant d'accéder à sa SC, il examine de faire passer le jeton avec le nombre restant de ressources (même nul) à son prochain selon le contenu de sa file d'attente. Durant la SC, si le nœud reçoit une requête et est toujours détenteur du jeton, le fait passer à ce nœud demandeur. A la sortie de la SC, un message de libération transportant le nombre de ressources libérées doit être envoyé sur le chemin du jeton.

Discussion

Le protocole de J-R. Jiang s'inspire grandement de l'algorithme RL [150]. Il hérite de ses nombreuses caractéristiques. Par conséquent, les même avantages et inconvénients discutés dans le Chapitre II, Section 3.3. Néanmoins, on peut ajouter aussi le problème de centralisation du service de récolte de requêtes et de distribution de ressources au niveau du seul nœud détenteur du jeton.

3.2. Solution de C-Z. Yang

Dans [152], C-Z. Yang a développé une solution de de *h* parmi *k* exclusion mutuelle orienté jeton pour les réseaux mobiles ad hoc. Cette solution s'inspire des algorithmes d'exclusion mutuelle de R. Badoni et *al* [12] et J. Walter et al [150]. Il emploie un ensemble dynamique d'anneaux dynamiques sur lesquels au maximum *k* jetons (i.e. ressources) circulent et un DAG dirigé vers un nœud spécial appelé l'arbitre de jeton. Ce nœud est le seul habilité à dispatcher les jetons sur les différents anneaux. Chaque anneau couvre un ensemble de nœuds. Les requêtes sont adressées à l'arbitre de jeton. Les jetons libérés suivent les chemins définis par la structure de *DAG* vers l'arbitre de jetons.

Cette solution fonctionne en tours. Durant chaque tour, un nouvel arbitre de jeton est sélectionné. Un nouveau tour est enclenché lorsque l'arbitre de jeton courant ne peut pas satisfaire la prochaine requête. Le propriétaire de cette requête devient le nouvel arbitre de jeton.

La solution de C-Z. Yang est flexible pour différentes politiques de navigation des jetons. Elle est ses premiers stades de développement (d'après l'auteur de ce travail) étant donné qu'il ne considère que la panne de liens comme caractéristique des réseaux mobiles ad hoc.

Discussion

Le protocole développé dans [152] présente un certain nombre de mécanismes intéressants. Premièrement, il utilise la structure de *DAG* de l'algorithme *RL* [150] pour désigner l'arbitre de jeton. Celle-ci lui permet en partie d'hériter de ses principales caractéristiques, notamment en terme d'optimisation des messages de demandes de ressources étant donné qu'ils sont dirigés correctement vers leur destination. Néanmoins, le maintient de cette structure est coûteuse en messages. L'utilisation de structures d'anneaux logiques dynamiques nécessite le recours à la couche de routage, ce qui d'une part, suppose un routage proactif et d'autres parts, les performances du protocole dépendent de celle de cette couche.
Etant donné que le protocole de [152] est une ''ébauche'' de solution à la *h-k* EM, il ne tolère pas la perte de jetons, la panne (ou la déconnexion de nœuds) et le partitionnement.

4. Conclusion

Dans ce chapitre, une étude sur les solutions de *h* parmi *k* EM connue dans la littérature des environnements distribués statiques et mobiles ad hoc a été réalisée. Ces solutions se basent en général, sur le concept de quorums mais diffèrent sur la manière de les construire. Elles profitent des propriétés des quorums pour la disponibilité et la résistance aux pannes mais leurs performances dépendent de la taille des quorums construits. Il est délicat de penser à utiliser ce concept dans les réseaux mobiles ad hoc en regardant sa dynamicité, étant donné qu'il est difficile de contacter tous les membres d'un quorum. Les deux seules solutions connues des réseaux mobiles ad hoc sont dues à J-R. Jiang [55] et à C-Z. Yang [152]. Ces deux solutions proposent des mécanismes intéressants et adéquats aux réseaux mobiles ad hoc

mais ne tiennent pas compte de certaines caractéristiques de ces réseaux telles que la pannes de nœuds et le partitionnement et présente un problème de scalabilité.

Les deux prochains chapitres sont dédiés à la présentation de nouveaux protocoles de *h-k* EM [16] pour les réseaux mobiles ad hoc qui proposent de mécanismes pour la majorité des caractéristiques des de ces réseaux.

Chapitre VII

Un protocole de *h* parmi *k* exclusion mutuelle basé liens physiques pour les réseaux mobile ad hoc

1. Introduction

L'allocation de ressources est un problème fondamental dans les systèmes distribués. L'exclusion mutuelle est le point d'entrée dans ce large domaine. Dans ce cas, il s'agit de gérer une ressource qui ne peut être utilisée que par un seul processus à la fois. Quand la ressource existe en *k* exemplaires, il s'agit du problème de la *k*- exclusion mutuelle dans lequel chaque processus ne peut utiliser qu'une seule ressource à la fois; donc, au maximum k processus exécutent en même moment leurs SC. Cependant, un processus peut vouloir demander *h* (*h* <=*k*) exemplaires de ressources. Il s'agit donc, du problème de la *h* parmi *k* exclusion mutuelle (*h*-*k*-EM).

Dans le but de résoudre le problème de *h* parmi *k* EM, plusieurs solutions ont été proposées dans la littérature des systèmes distribués. Le premier protocole est du à M. Raynal [111]; il requière à un nœud de récolter *k* permissions afin d'utiliser les *h* ressources demandées (ou par abus de langage, d'accéder à sa SC). Toutes les autres solutions utilisent le concept de structures de quorums, qui sont des collections d'ensembles de nœuds. Dans [9], le concept d'ensembles d'arbitres est utilisé. Dans [85], les *k*- *ar*bitres sont utilisés. Dans [69], le concept de coterie local est utilisé. Dans [58], le concept de *k*- coterie est utilisé. Dans [86], le concept de *h*-*k*- arbitres qui généralise les *k*- arbitres est utilisé et dans [57], le concept de cohortes est utilisé. Les solutions qui utilisent les *k*- arbitres et les *h*-*k*- arbitres sont analysés dans [58]. A cause de la difficulté de construction de quorums, des opérations de composition sur les *k*-arbitres ont été définies dans [74, 75]. Toutes ces solutions ont le même principe [80]: Pour réaliser une SC, un nœud doit recevoir une autorisation de tous les membres du quorum sélectionné; la différence entre ces différentes solutions réside dans la manière de construire ces quorums. Toutes ces solutions engendre un coût de communication bas relativement à la solution de [111] et sont souvent tolérantes aux pannes comme celle dans [57].

Le concept de quorum n'est pas pratique dans le cas de réseaux ad hoc puisque la topologie de communication peut être très dynamique. De ce fait, il est très délicat d'entretenir une structure stable de quorums car il est difficile de contacter tous ses membres. Deux travaux seulement sont connus dans la littérature des réseaux mobiles ad hoc. Ils sont dus respectivement à J-R. Jiang [55] et C-Z. Yang [152].

La solution de J-R. Jiang [55] est orientée jeton et est une adaptation de la solution d'exclusion mutuelle développé dans [150] (voir Chapitre II) lui permettant d'hériter ses principales caractéristiques. Elle utilise un seul jeton et une structure de DAG mappée sur la topologie réelle du réseau, qui maintient plusieurs chemins vers le détenteur du jeton. Cette solution est convenable au réseaux mobiles ad hoc et permet à chaque nœud d'entretenir uniquement des informations sur son voisinage immédiat. La principale différence avec la solution de [150] est liée à la gestion de plusieurs exemplaires de ressources. Le jeton transporte des informations sur le nombre de ressources libérées. Cette solution favorise l'accès concurrent aux ressources. Les messages des ressources libérées suivent le chemin du jeton.

La solution de C-Z. Yang [152] s'inspire en partie des algorithmes d'exclusion mutuelle de Baldoni et *al* [12] et Walter et *al* [150]. Sa solution se base sur l'utilisation de deux types de structures : Un ensemble d'anneaux logiques dynamiques sur lesquels au maximum k jetons (i.e. ressources) circulent et un DAG de pointeurs dirigés vers un nœud spécial nommé l'arbitre de jeton. Cet arbitre de jeton est le seul qui est autorisé à dispatcher les jetons à travers les différents anneaux. Chaque anneau couvre un groupe de nœuds demandeurs. Les requêtes sont adressées à l'arbitre de jeton. Les jetons libérés suivent la structure de DAG pour atteindre l'arbitre de jeton. La solution de C-Z. Yang fonctionne en tours. Durant chaque tour, un nouvel arbitre de jeton est sélectionné. Un nouveau tour est initié lorsque l'arbitre de jeton courant ne peut pas satisfaire une prochaine requête. Le nœud propriétaire de cette requête devient le nouvel arbitre de jeton. Ce mécanisme permet d'éviter l'interblocage. La solution de C-Z. Yang est flexible pour différentes politiques de circulation des jetons. Cette solution est dans son premier stade étant donné (comme a indiqué son auteur) qu'elle ne considère que la rupture de liens.

Ces deux protocoles ne tiennent pas compte de certaines caractéristiques des réseaux mobiles ad hoc, telle que la panne de nœuds et le partitionnement et leurs conséquences comme la perte du jeton et de ressources. Dans ce chapitre, nous proposons un nouveau protocole distribué orienté jeton pour le problème de la h parmi k EM dans les réseaux mobiles ad hoc [16, 14, 15]. Ce protocole ne fait pas recours à la couche de routage et n'utilise pas de structure logique de communication. Une requête est envoyée sur les routes d'une de noeuds pour lesquels une requête est présente localement avec un rayon dynamique (typiquement 20%) calculé pour atteindre une proportion de ceux-la. Durant la circulation du jeton, les requêtes sont satisfaites en se basant en même temps sur leurs ages, leurs distances par rapport au jeton et le nombre de ressources demandées. Les ressources libérées suivent le chemin du jeton. Par ailleurs, le protocole proposé prend en charge de nombreuses caractéristiques des réseaux mobiles ad hoc telles que : Le mouvement des nœuds (i.e. les cassures et formations de liens), le partitionnement et la fusion, les pannes de nœuds et leurs conséquences, i.e. la perte de jeton et de ressources. Le protocole favorise la scalabilité du réseau et considère un nombre indéterminé de nœuds. De plus, il gère un jeton avec k ressources au niveau de chaque partition créée et les régénère dans le cas de perte. La duplication de jeton et de ressources est évitée dans le cas de fusions de partitions.

Ce chapitre est organisé comme suit : Après un bref aperçu sur quelques travaux antérieurs dans la Section 1, la Section 2 présente les idées de base du protocole et les structures de

données et de communication utilisées. Ensuite dans la Section 3, une description du fonctionnement du protocole est donnée. La Section 4 finalise la description du fonctionnement du protocole en montrant son comportement vis à vis du problème de partitionnement. Un simple exemple d'exécution suivi d'une discussion sur certains aspects cachés du protocole sont donnés dans la Section 5. La Section 6 traite la correction du protocole. La Section 7 expose les différents résultats de simulation obtenus ainsi que leurs interprétations. Finalement, la Section 8 conclue le chapitre.

2. Eléments généraux du protocole

2.1. Modèle du réseau

Nous supposons un réseau composé d'un nombre variable de nœuds mobiles, chaque nœud possède un identificateur unique. Les mouvements des nœuds du réseau sont libres (i.e. mobilité aléatoire). Les nœuds utilisent un support de communication sans fil, les liens de communication sont bidirectionnels, FIFO et fiables. Un nœud peut à tout moment tomber en panne ou quitter le réseau. Les mouvements des nœuds peuvent engendrer des créations et/ ou des pertes de liens, le partitionnement et la fusion. La couche de liaison permet à tout moment aux nœuds de connaître leurs voisins. Par ailleurs, nous supposons que la durée d'une SC est bornée. Dans la suite de ce chapitre et afin de décrire notre protocole, nous nous plaçons au niveau du nœud désigné par l'indice *i*.

2.2. Idées de base du protocole

Le protocole décrit dans ce chapitre reprend certains principes du protocole développé dans [20]. La lecture de ce chapitre ne suppose pas la connaissance préalable du protocole développé dans [20]. Il utilise l'approche à jeton en combinant la circulation et la demande de ce dernier. Le jeton est identifié de manière unique dans chaque partition et transporte entre autres une file de requêtes récoltées lors de son parcourt et le nombre de ressources disponibles. Afin d'assurer la sûreté du service, le détenteur du jeton est le seul à pouvoir distribuer les ressources aux processus demandeurs. Le protocole se base sur la connaissance locale, celle du voisinage immédiat et ne prend en considération que les liens physiques pour acheminer de l'information. Ce qui lui permet de favoriser la *scalabilité* du réseau et de considérer un nombre *indéterminé* de nœuds. D'autre part, le protocole fait recours à un certain nombre de mécanismes que nous décrivons par la suite. Dans la suite de ce chapitre, nous utilisons par abus de langage l'accès à la SC au lieu de l'accès aux ressources.

2.2.1. Rayon d'envoi

Puisque la détention du jeton conditionne l'accès à la SC, la diffusion d'une requête est l'un des moyens permettant de l'acquérir. Afin de minimiser l'impact de la diffusion, nous utilisons le concept du *rayon d'envoi*. Ce rayon est calculé de manière dynamique à partir des requêtes déjà reçues par *i* (dans une file locale notée Q_i). En supposant que les requêtes reçues localement sont triées dans un tableau temporaire T_i selon l'ordre croissant de leurs longueurs de routes, le rayon est calculé comme suit *:* $R_i = / (T_i (pos).route) /$ où $pos = Round (P*/T_i /)$, où *P* est une constante représentant la proportion de nœuds à atteindre et */x/* désigne la longueur de *x*. Les nœuds figurant dans le tableau T_i et délimités par ce rayon vont recevoir la requête au même titre que ceux se trouvant sur leurs chemins. Dans la Figure 36, admettons que le nœud *1* désire émettre une requête, il doit donc calculer son rayon d'envoi R_1. Pour se faire, supposons que sa file Q_1 contient les demandes des nœuds *2, 4, 11* et *12* avec leurs routes

respectives *2* ; *5*, *4* ; *6* , *9*, *11*; *6*, *10*, *12*. La requête sera envoyée sur ces chemins pour atteindre tous ces nœuds (i.e. *2, 4, 11* et *12*) si $R_1 = 3$, ou *50 %* de ces nœuds si $R_1 = 2$. Cette méthode de calcul du rayon favorise la création d'une cascade de chemins vers le détenteur du jeton. Puisque le jeton récolte les requêtes rencontrées sur son chemin, tous ces processus seront éventuellement satisfaits, en regardant la mobilité et les pannes de nœuds.

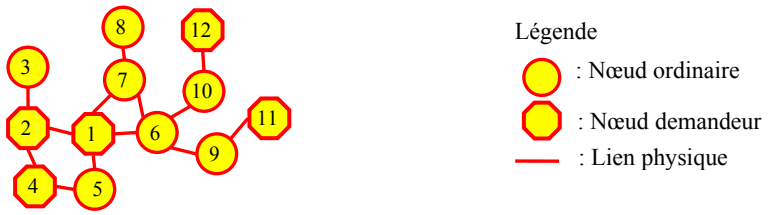


Figure 36: *Exemple de calcul du rayon d'envoi.*

2.2.2. Relances de requêtes

Un nœud demandeur de ressources peut ne pas recevoir le jeton et ceci pour plusieurs raisons (dues à la mobilité): Changement fréquent du voisinage de ce nœud, changements fréquents des liens dans le réseau, pannes de nœuds, perte de jeton, etc… Dans le but d'éviter la famine à ce nœud, la notion de relance de requête est utilisée: Un nœud demandeur de ressources non satisfait au bout d'un délai défini par un temporisateur (i.e. *RequestTimer*) est amené à relancer sa requête. Cette relance est considérée comme une nouvelle tentative de la même requête; autrement dit, elle est générée avec la même estampille afin de lui assurer son ancienneté dans le système. Le nombre de relances ne doit pas être illimité. En effet, au-delà d'un certain seuil (i.e. *Req_Threshold*,), un mécanisme lié au soupçon de perte du jeton doit être enclenché (voir Section 4.1).

2.2.3. Sélection du nœud privilégié

Chaque nœud recevant une demande aura connaissance du nombre de ressources demandées par le nœud originaire de cette demande ainsi que de la route parcourue et l'inscrit dans sa file d'attente. Si cette file est organisée suivant une politique qui privilégie les requêtes les plus proches, des cas de famine peuvent en découler. En effet, dans le cas d'un groupe de nœuds proches les uns des autres qui demandent l'entrée en SC fréquemment, le jeton ne cessera pas de circuler entre eux, ce qui prive les demandeurs les plus éloignés. Afin d'éviter ce problème, l'ancienneté des requêtes est aussi considérée [76]. D'autre part, afin d'éviter la sous utilisation de ressources, le nombre de ressources demandées est aussi un autre critère. D'où, la priorité d'une requête d'un nœud *j*, au niveau du nœud privilégié *i* est donnée par $Pr_j = (c1*NS_j + c2*\ nb\_hops_j + c3*Nb\text{-}req\text{-}resources_j, j)$, où *c1*, *c2* et *c3* sont des constantes dont les valeurs définissent l'importance de tel ou tel paramètre, nb_hops_j est le nombre de sauts séparant *i* de *j*, $Nb\text{-}req\text{-}resources_j$ est le nombre de ressources demandées et *j* permet de créer un ordre total entre les noeuds. Comme la distance qui sépare deux nœuds et le nombre de ressources sont bornés et les estampilles des nœuds servis augmenteront rapidement, la priorité tournera forcément en faveur des nœuds plus éloignés. Notons que dès lors qu'un nœud est sélectionné comme prochain nœud privilégié, le jeton lui est acheminé *même* si le nombre de ressources n'est pas suffisant pour satisfaire sa requête. Dans ce cas, le nœud destinataire garde le jeton localement jusqu'à ce qu'il reçoive assez de ressources pour servir sa requête.

D'autre part, pour assurer la propriété de sûreté, un nœud demandeur *n'accède à sa SC que s'il reçoive le jeton*. Aussi, quand un nœud décide d'accéder à sa SC, avant d'y accéder il doit

envoyer le jeton au prochain nœud privilégié. Ce mécanisme assure l'accès concurrent à la SC.

2.2.4. Verrouillage de ressources

Quand un nœud est sélectionné comme prochain à être servi, le jeton lui est véhiculé sur le chemin enregistré avec sa requête. Afin d'éviter au jeton de faire éventuellement plusieurs sauts avant d'atteindre son destinataire, et par conséquence améliorer le temps d'attente d'un nœud demandeur, des nœuds intermédiaires demandeurs de ressources et se trouvant sur la route du jeton vers le destinataire peuvent être servis, puisque le jeton leur est parvenu, mais sans influencer la trajectoire initiale du jeton ; d'où l'utilisation de la notion de *verrouillage de ressources*. Quand un nœud est sélectionné comme prochain destinataire, le nombre de ressources demandées par ce nœud lui est déduit du nombre total libre de ressources transportées par le jeton. Les nœuds intermédiaires ne peuvent donc utiliser que l'excédent.

2.2.5. Recherche de route du jeton

Pendant la circulation du jeton vers une destination donnée, la route peut être perdue. Ainsi, un état est associé au jeton: *ONRoute*, s'il suit normalement une route prédéfinie; *NORoute*, dans le cas de perte de route et un nouveau destinataire n'est pas encore trouvé. Un nœud *i* demandeur de ressources qui reçoit le jeton avec un état *NORoute* est permis de servir sa requête puisqu'il est sans destination fixe.

Quand un noeud *i* est amené à véhiculer le jeton à une destination via la route que son message transporte, deux cas sont à distinguer: Le prochain nœud sur la route ou la destination est un voisin, le jeton lui est simplement transmis. Par contre, si la route est brisée, le nœud *i* tente de la recalculer d'une manière locale en cherchant la présence des nœuds suivants le saut brisé dans son voisinage. Si cette opération échoue, le jeton change de destinataire qui sera choisie selon la méthode de *sélection du nœud privilégié.* En cas d'échec, il est envoyé à l'un des voisins (avec l'état *NORoute*) pour la recherche d'un autre destinataire. Ce dernier comportement évite la stagnation du jeton au moment où des requêtes restent non satisfaites. La conséquence en est éventuellement la non satisfaction de certains nœuds demandeurs. Ce problème peut être surmonté par le mécanisme de relance de demandes de SC après un seuil prédéfini.

2.2.6. Recherche et régénération du jeton

Le soupçon de perte de jeton est déclenché quand un nœud *i* ne le reçoit pas après un nombre limité de tentatives de demandes. Au-delà, si ce nœud n'est pas en état de recherche de ressources (voir Section 2.2.9) (le cas contraire signifie que le jeton est présent et donc le soupçon est avorté), une recherche dans toute la partition courante s'impose. Pour des raisons d'optimisation, le message diffusé (i.e. *IsThereToken ()*) est à quadruple objectif: Demande de jeton, recherche de jeton, proposition de création de jeton en cas de perte et demande d'annulation de ressources (voir Section 4.1). Ainsi, tous les nœuds informés de cette recherche en cours se *bloquent* temporairement du point de vue génération de demandes de SC, véhicule du jeton et véhicule de ressources, s'il y a lieu. Après un délai fixe, i.e. *SuspicionTimer*, initialisé au niveau de chaque noeud informé par cette recherche, la régénération du jeton est effectuée. Sachant que plusieurs nœuds peuvent être en même temps en situation de recherche du jeton, en cas de perte de jeton (ou sa non présence dans la partition courante), un conflit se produit lors de sa création. De ce fait, on utilise un mécanisme de sélection d'extremum connu dans les protocoles d'élection de nœud afin de choisir le nœud qui va créer le jeton: Par exemple, le nœud ayant l'identité la plus petite doit

le créer. Notons que seulement les nœuds qui cherchent effectivement le jeton participent activement (peuvent être candidats) dans le processus d'élection.

2.2.7. Désignation du jeton

A cause des problèmes de partitionnement, plusieurs jetons peuvent être créés, un pour chaque partition créée. Au moment de la fusion de partitions, il est impératif de ne garder qu'un seul jeton dans la partition ainsi produite. Pour faciliter la destruction d'un jeton, il suffit de l'identifier. L'identité retenue pour un jeton *est celle du nœud qui l'a créé*. Cependant un nœud, ayant créé un jeton dans une partition donnée, peut avoir la chance d'en créer un autre s'il se retrouve dans une autre partition. Afin d'éviter l'ambiguïté de désignation des jetons, à chaque nouveau jeton est associé *un numéro de séquence local au processus qui l'a créé*.

2.2.8. Récupération de ressources libérées

Chaque nœud qui libère des ressources est amené à les véhiculer vers le détenteur du jeton puisque les ressources libres accompagnent le jeton. Il est donc intéressant de leur faire *suivre les traces du jeton afin de localiser son détenteur* en évitant la circulation en boucle et en optimisant le chemin quand il est possible. Dans ce but, chaque nœud garde trace du nœud auquel il a envoyé le jeton la dernière fois. Pour des raisons de changements de liens, cette information à elle seule ne suffit pas pour diriger les ressources vers leurs destinations. Dans l'exemple de la Figure 37, si le chemin du jeton est *4*, *3*, *6*, *5*, *2*, *7*, et si une rupture de liens se produit entre le nœud *5* et *2*, les ressources issues, par exemple, du nœud *4* vont suivre par défaut le chemin *4*, *3*, *6*, *5*, *3*; une boucle est alors formée ! De ce fait, on introduit la notion de *numéro d'ordre*. Le jeton balise son chemin à l'aide d'un *séquenceur circulant* qui lui est affecté de telle sorte que chaque nœud visité par le jeton garde la *dernière* valeur du séquenceur tirée localement. Lorsqu'un nœud *i* libère des ressources, il affecte au message de libération de ces ressources la valeur locale du séquenceur comme numéro d'ordre (ou *clé*). Celui-ci va servir à identifier l'exploration de ce message de manière unique étant donné la multiplicité de messages de libérations qui explorent en même temps le réseau à la recherche du bon chemin vers leurs destinations (voir Section 3.4). Ce bon chemin est caractérisé par la croissance des numéros d'ordres des nœuds traversés par un message de libération de ressources. Ainsi, grâce à ces deux informations, les ressources libérées suivent dans les meilleurs cas un chemin linéaire vers le détenteur du jeton (c'est le cas où chaque nœud prochain sur le chemin du jeton est toujours voisin de son prédécesseur sur ce même chemin) ; autrement, un chemin arborescent. Parfois, à cause des changements fréquents de liens, ces deux outils ne suffissent pas de faire parvenir les ressources à leur destination. Autrement dit, si tous les chemins sont explorés sans succès, les ressources sont gardées localement (soit par le noeud originaire de la libération soit par un autre atteint par ces ressources et se trouvant dans cette situation à cause d'une rupture de lien) avec un état d'inaptitude (i.e. *unable*) en attendant la création d'un nouveau lien qui permettra de les acheminer vers la bonne destination ou le passage d'un autre message de libération de ressources.

Figure 37 : *Acheminement en boucle de ressources.*

2.2.9. Recherche et régénération de ressources

La perte de ressources peut être soupçonnée par un nœud détenteur de jeton et demandeur de SC (i.e. état *wait*) lorsqu'il ne reçoit pas les ressources manquantes (on admet naturellement que la demande d'un processus ne dépasse pas *k* ressources) après un délai fixé (i.e. *RessourcesTimer*). Les raisons sont, soit la perte de ressources suite à la panne d'un ou plusieurs nœuds soit leurs déplacements vers d'autres partitions soit la perte du chemin de retour pour les messages de libérations de ressources suite à de changements fréquents de liens de communications. Pour remédier à tous ces problèmes, il est intéressant d'informer tous les nœuds de la partition de ce fait et de régénérer les ressources manquantes. Pour éviter les conséquences d'une fausse détection, les nœuds informés sont invités à détruire les ressources éventuelles en leurs possessions même s'ils sont en SC. Un délai de stabilisation (i.e. *RecoverTimer*) est nécessaire pour tous les processus de la partition. Il permet, d'une part, pour des raisons de sûreté du service, au détenteur de jeton de temporiser les nœuds en SC d'en finir avant de générer les ressources. Il permet aussi à tous les noeuds de neutraliser toute recherche éventuelle de jeton durant cette période de stabilisation.

2.3. Structures

2.3.1. Structures de bases locales à un nœud

Chaque nœud *i* est doté d'un certain nombre de variables et de structures définissant son contexte. Ces variables et structures seront données comme suit:

- $State_i$: Indique l'état de *i*, par rapport à la SC, qui peut être *idle*, *req*, *inCS,* ou *wait*. Initialement, $State_i=idle$.
- NS_i : Estampille du nœud *i*, elle permet de dater ses requêtes. Initialement, $NS_i=0$.
- Req_res_i : Indique le nombre de ressources requises si $State_i=req$.
- $attempts_i$: Indique le nombre de relances de la requête en cours.
- N_i: Indique l'ensemble de tous les voisins ayant un contact direct avec *i*. Initiallement, N_i contient tous les voisins de *i*.
- Q_i: La file locale à *i* structure comme $(j, NS_j, Req\_Res_j, Route_j, attempts_j)$, où *j* est un noeud demandeur de SC, NS_j est l'estampille de la requête de *j*, Req_Res_j est le nombre de ressources requises, $Route_j$ est la route suivie par la requête entre *j* et *i* et $attempts_j$ est le nombre de tentatives de la requête de *j*.
- $Holder_i$: Une variable booléenne indiquant si *i* détient ou non le seul jeton de la partition en cours. Initialement, $Holder_0= true$ et $Holder_i=false$, si $i<>0$.
- $Last\_Holder_i$: Contient l'identificateur du noeud détenteur du jeton avant *i*. Initialement, $Last\_Holder_i =i$.
- $Token\_ID_i$ (respectivement $Token\_ID\_NS_i$) : Indique l'identité (respectivement l'estampille) du jeton. Initialement $Token\_ID_i =0$ (respectivement $Token\_ID\_NS_i=0$).

- *Search_Token$_i$* : Une variable booléenne indiquant si *i* cherche ou non le jeton. Initialement, *Search_Token$_i$ = false.*
- *TCreatedNS$_i$* : Indique un nombre séquentiel local à *i* utilisé lors de la création d'un jeton. Initialement *TCreatedNS$_i$=0.*
- *Block_state$_i$*: Si *Search_Token$_i$ =true,* cette variable indique l'action à effectuer par *i* au moment de la mise de *Search_Token$_i$* à *false* (i.e. après déblocage). Par exemple, elle peut prendre comme valeurs: *ToAsk_For_CS, ToSend_Token, ToSend_Release…*
- *Next$_i$*: L'identificateur d'un noeud voisin pour lequel *i* a envoyé en dernier le jeton. Initialement *Next$_i$=i.*
- *Order_ Num$_i$*: Le numéro d'ordre de *i* sur le chemin du jeton. Initialement *Order_ Num$_i$=0.*
- *Rel_List$_i$*: Une trace de numéros d'ordre de messages de libération de ressources et d'identificateurs de noeuds desquels *i* a reçu ces messages la première fois.
- *Rel_state$_i$*: Elle prend *unable* si des ressources bloquées sont gardées au niveau de *i* et *idle* sinon. Initialement *Rel_state$_i$= idle.*
- *Rel_res$_i$* : Indique le nombre de ressources bloquées si *Rel_state$_i$=unable*.
- *Rel_order$_i$* : Indique le numéro d'ordre des ressources bloqués si *Rel_state$_i$= unable*.
- *Search_Res$_i$*: Une variable booléenne indiquant si *i* est entrain de chercher des ressources ou non. Initialement *Search_Res$_i$=false.*

2.3.2. Types de communications entre noeuds

Le protocole utilise différents types de communications pour réaliser le service de *h* parmi *k* EM.

- Une demande d'entrée en SC est exprimée par le message *Request (i, NS$_i$, Req_res$_i$, attempts$_i$, Route_p$_i$, Route_f$_i$, aware$_i$)*. Ce message est accompagné respectivement de l'identité du nœud source, de l'estampille du message, du nombre de ressources demandées, du nombre de tentatives de la requête, du rayon d'envoi prévu, du nombre de tentatives de cette requête, de la route planifiée à suivre dans le meilleur cas, de la route suivie qui se construit au fur et à mesure et d'une connaissance sur les nœuds visités par ce message (permettant de diminuer la redondance de ce type de message).
- Le privilège d'entrée en SC est matérialisé par le message *Token (Tid, TidNS, TLast_Holder, TRoute, TState, TOrder, TAvail_res, TLock_Res, TKnownNS, TReq_Nodes, TReq_Res)*. Ce message est accompagné respectivement de son identification, de l'identification de son ancien détenteur, de la route à suivre, de son état (i.e. suit un chemin prévu ou a perdu son chemin), de son numéro d'ordre actuel, du nombre de ressources disponibles, du nombre de ressources verrouillées pour le destinataire, ainsi que des informations sur les requêtes connues de ce jeton.
- Les ressources libérées sont envoyées au nœud détenteur de jeton à l'aide du message *Release (Tid, TidNS, order$_i$, Last_order$_i$, Free_res$_i$, Visit_Neigh$_i$)*. Ce message transporte respectivement l'identification du jeton actuel, la clé du message, le numéro d'ordre du nœud émetteur, le nombre de ressources libérées et une connaissance des nœuds visités par ce message.
- La panne de nœuds, le partitionnement et les changements fréquents de liens entre nœuds peuvent engendrer une perte ou un isolement temporaire du jeton de la partition considérée. Le soupçon de perte du jeton dans la partition en cours apparaît dès lors qu'un nœud demande plusieurs fois le jeton mais sans le recevoir. Le message *IsThereToken (NS$_i$, Req-res$_i$, Route_f$_i$, attempts$_i$, j, TCreatedNS$_i$, aware$_j$)* permet donc la recherche d'un jeton. Ce message transporte respectivement des informations sur la requête qui est à

l'origine de cette recherche, une proposition d'identification du jeton à créer en cas de perte et une connaissance des nœuds visités par ce message.

- Dans le cas où le jeton en cours n'est pas perdu, son détenteur utilise le message *ThereIsToken (aware$_i$)* comme confirmation de son existence. Ce message transporte uniquement une connaissance des nœuds visités par ce message.
- Le soupçon de perte de ressources dans une partition apparaît lorsqu'un nœud demandeur détenant le jeton a attendu longtemps sans recevoir de ressources libérées lui manquants, le message *IsThereResources (Tid$_i$, TidNS$_i$, aware$_i$)* permet la recherches de ces ressources. Ce message transporte uniquement l'identification du jeton courant et une connaissance sur les nœuds visités par ce message.
- Les fusions de partitions sont aussi une autre conséquence des mouvements libres des nœuds. Des problèmes de duplication de jetons peuvent donc survenir étant donné que chaque partition est dotée de son propre jeton. Comme la fusion se produit à l'occasion de créations de liens entre nœuds, l'échange d'informations entre les nœuds concernés devient vital. Pour cela, le message *LinkInfo (Tid, TidNS, Search_Token$_j$, j, NS$_j$, Req_res$_j$, route_j$_i$, attempts$_j$, TCreatedNS$_j$, Search_res$_j$, Rel_order$_j$, Last_order$_j$, Rel_res$_j$, aware$_j$)* est utilisé. Au niveau d'un nœud émetteur, ce message transporte respectivement l'identification du jeton de sa partition, et éventuellement les arguments d'un message *IsThereToken ()* ou d'un message *IsThereResources ()* ou d'un message *Release ()* tous reçus ou émets par ce nœud.
- Enfin, dans le cas de détection de fusion, le message *DeleteToken (Tid, TidNS, aware$_i$)* est utilisé pour supprimer l'un des deux jetons et toutes les ressources de sa partition. Afin d'informer les nœuds dans la partition ciblée de l'identification du seul jeton retenu, l'identification de celui-ci accompagne ce message et aussi une connaissance sur les nœuds visités par ce message permettant de minimiser le nombre de messages à envoyer.

3. Description du fonctionnement du protocole

Chaque nœud du réseau exécute un algorithme symétrique. Son texte est formé d'un ensemble de primitives liées aux différents événements le constituant. Au niveau de chaque nœud, les primitives s'exécutent de manière atomique. Cependant, des événements qui se produisent durant l'exécution de la SC peuvent être traités.

3.1. Acheminement d'une requête

Quand un nœud *i* désire accéder à sa SC (*State$_i$=Req*), dans le cas de détention du jeton (*Holder$_i$=true*), il y accède directement si le nombre de ressources est suffisant pour satisfaire sa requête (*TAvail_res>=Req_res$_i$*). Autrement, il passe dans un état local spécial (*State$_i$:=wait*) d'attente de libération de ressources. Un délai de garde *ResourcesTimer* est associé à cette attente, au bout duquel le soupçon de perte de ressources est engagé (voir Section 2.2.9). Si le nœud *i* ne dispose pas du jeton, il envoie une requête *Request (),* estampillée selon les horloges de L. Lamport [76] sur les routes d'une proportion de nœuds (, dont une requête est présente localement,) définie par un rayon calculé localement (voir l'exemple dans la Section 2.2.1). Tous les nœuds intermédiaires recevant cette requête se chargent de la véhiculer selon le rayon indiqué. Dans le cas où la file de requêtes Q_i est vide ou toutes les requêtes sont enregistrées avec des routes vides, c'est aux nœuds intermédiaires (en commençant par ses voisins) récepteurs de la requête d'en définir un selon les états de leurs files. Un délai de garde (i.e. *RequestTimer*) est associé à chaque requête émise. Au bout de ce délai, si le jeton ne parvient pas à ce nœud demandeur, la requête est relancée mais avec

la *même* estampille afin de lui assurer son ancienneté. Dans le cas échéant, ce processus est exécuté un nombre de fois limité (i.e. *Req_Threshold*) avant de soupçonner la perte du jeton (voir Section 4.1).

Quand un noeud *i* est amené à envoyer une requête mais il est bloqué (i.e. $Search\_Token_i = true$), l'envoie de la requête est reporté au moment du déblocage ($Block\_state_i$ est alors affecté de *ToAsk_For_CS*).

3.2. Conditions d'accès à une SC

Un nœud *i* demandeur de SC y accède dans les cas suivants:

- il détient le jeton à l'état de repos et le nombre de ressources du jeton est suffisant pour satisfaire sa requête ($(State_i = req)$ and $(Holder_i = true)$ and $(TAvail\_res >= Req\_res_i)$),
- il détient le jeton et attend les ressources (i.e. $State_i = wait$), mais vient de recevoir un message de libération de suffisamment de ressources,
- il reçoit le jeton $(Holder_i = true)$ et de plus, soit il est destinataire et les ressources sont suffisantes (i.e. $(Last(TRoute) = i)$ and $(TAvail\_res >= Req\_res_i)$), où *Last* est une fonction qui retourne le dernier élément de la route donnée en paramètre), soit les ressources non verrouillées du jeton sont suffisantes (i.e. $(TAvail\_res - TLock\_Res) >= Req\_res_i$), soit le jeton est sans destination fixe et les ressources transportées par le jeton sont suffisantes (i.e. $(TState = NORoute)$ and $(TAvail\_res >= Req\_res_i)$). Dans ce cas, les ressources éventuellement bloquées au niveau de *i* sont ajoutées à celles du jeton avant de s'en servir.

Dans ces cas ci-dessus (excepté le cas où *i* est intermédiaire), si le nombre de ressources libres n'est pas suffisant, le nœud *i* se met (ou reste selon le cas) à l'état d'attente de ressources (i.e. *wait*).

Notons que dès lors que l'accès à la SC est décidé, si le nœud *i* n'est pas bloqué à cause d'une recherche en cours du jeton ($Search\_Token_i = false$), avant d'exécuter sa SC le jeton accompagné des ressources restantes est passé au prochain nœud demandeur, s'il en existe. Le prochain nœud peut être choisi selon la méthode de sélection décrite à la Section 2.2.3 ou celui adressé par la route inscrite sur le jeton si *i* n'est pas destinataire. Dans le cas où sa file de requêtes est vide, ce jeton est gardé localement à l'état de repos. Dans le cas où le nœud est bloqué, l'envoi du jeton est reporté au moment du déblocage (i.e. $Block\_state_i$ est alors affecté de *ToSend_Token*).

3.3. Déplacement du jeton

Un nœud *i* détenant le jeton et n'étant pas dans une situation de blocage (voir Section 4.1), se charge de le véhiculer, avec l'état *ONRoute*:

- Cas a : sur la route d'un nouveau destinataire choisi selon la politique de sélection du nœud privilégié en lui verrouillant le nombre de ressources nécessaires (si le nombre de ressources n'est pas suffisant, toutes ces ressources sont verrouillées), si l'un des cas suivants se présente :

- le nœud *i* est détenteur du jeton (et il est soit en SC soit à l'état de repos) et vient de recevoir une nouvelle requête,
- le nœud *i* est destinataire du jeton et a décidé d'accéder à sa SC (voir Section 3.2) et sa file d'attente contient au moins une requête,
- le nœud *i* était bloqué *et* était entrain d'attendre l'envoi du jeton à un prochain nœud demandeur (i.e. $(Search\_Token_i = true)$ and $(Block\_state_i = ToSend\_Token)$), mais vient de

se débloquer suite à l'expiration du temporisateur *StabilisationTimer* (sera vu dans la Section 4.1).

- le jeton vient d'être reçu avec un état *NORoute*, mais le nœud *i* n'est pas demandeur de SC et sa file locale n'est pas vide.

- Cas b: sur la route d'un destinataire prédéfini mais après une éventuelle opération de récupération de cette route (dans le cas de sa perte suite au déplacement de nœuds prévus sur celle-ci) dans le cas suivant:

- le jeton vient d'être reçu avec une route prédéfinie et le noeud *i* n'est pas le destinataire.

Dans le cas (b), si la récupération de route est impossible, un nouveau destinataire s'il y a lieu est choisi; sinon, le jeton sera affecté de l'état *NORoute* et sera passé à un voisin qui est sélectionné afin de rechercher un nouveau destinataire en évitant les boucles. Cette dernière opération est aussi effectuée dans le cas (a), si la file n'est pas vide et les chemins vers tous les nœuds de celle-ci sont non récupérables.

En dehors des cas cités ci-dessus, le jeton est gardé localement. Il est à noter que pour tous les cas indiqués ci-dessus, si le nœud *i* est bloqué à cause d'une recherche en cours du jeton, l'envoi du jeton est reporté au moment de son déblocage ($Block\_state_i$ est alors affecté de *ToSend_Token*) (voir Section 4.1).

3.4. Acheminement de ressources

Rappelons que l'acheminement de ressources vers le détenteur du jeton se fait en suivant les traces du jeton (voir Section 2.2.8). Dans ce cadre, un nœud *i* en possession de ressources libres et qui ne détient pas le jeton (autrement, les ressources sont ajoutées à celles du jeton avec conséquence éventuelle l'entrée en SC si ce nœud est en attente de ressources (voir Section 3.2),) est amené à transmettre un message de libération de ressources vers un des nœuds voisins. Celui-ci peut être le noeud auquel il a envoyé le jeton la dernière fois (i.e. $Next_i$) mais dans le cas où le lien est rompu avec celui-ci, le choix porte sur un autre voisin sélectionné selon un critère qui permet d'éviter la circulation en boucle du message (voir Section 2.2.8).

Pour un nœud *i*, deux cas d'acheminement existent:

- Soit lorsqu'il sort de sa SC. Dans ce cas, le message aura comme *clé* le numéro d'ordre du nœud *i* sur le chemin du jeton. Si des ressources sont bloquées localement, pour des raisons d'optimisation, le message aura à véhiculer le cumul des ressources libérées et celles bloquées mais avec une clé qui prend le maximum entre la clé des ressources bloquées et celle des ressources libérées.
- Soit lorsqu'il reçoit un message de libération de ressources. Dans ce cas, la clé à envoyer avec le message est celle du message reçu si aucune ressource n'est bloqué localement ; est le maximum de la clé des ressources bloquées et celle du message reçu, autrement.

Dans ces deux cas, le nœud *i* envoie le message à son prochain sur le chemin du jeton (cheminement normal), s'il existe *(*i.e. $Next_i \in N_i$*)* et si le numéro d'ordre du nœud précédent (ou celui de *i* si le numéro d'ordre des ressources est celui de *i*) sur le chemin du jeton est inférieur ou égal au numéro d'ordre de *i*. Dans le cas contraire, un voisin est choisi selon la méthode d'exploration en séquentiel d'un réseau avec transport d'information de contrôle contenant l'ensemble des voisins visités [114, 115, 116].

Si le nœud *i* est dans l'état de recherche de ressources (i.e. $Search\_Res_i = true$), il n'aura pas à acheminer les ressources libérées lors de la sortie de la SC puisque celles-ci ont été *annulées* lors de la réception du message de recherche de ressources pendant l'exécution de sa

SC (voir Section 2.2.9). Si le nœud *i* est bloqué à cause d'une recherche en cours du jeton, l'envoi de ressources est reporté au moment du déblocage ($Block\_state_i$ est alors affecté de *ToSend_Release*).

La Figure 38 donne un exemple d'acheminement de ressources. Dans la Figure 38(a), on suppose que partant du nœud *1*, le jeton est véhiculé sur le chemin *2, 4, 7, 3, 6, 5*. Si le numéro d'ordre de *1* est *10*, les nœuds traversés par le jeton ont respectivement comme numéros d'ordre *11*, *12*, *13*, *14*, *15*, *16*. Dans la Figure 38(b) où des changements de liens ont eu lieu, admettons que le nœud *1* aura à véhiculer des ressources libérées localement. Le message correspondant aura comme numéro d'ordre *10* et comme chemins possibles: *4*, *2*, *7*, *2*, *6*, *5* ou *4*, *2*, *6*, *5* ou *4*, *3*, *6*, *5* ou *3, 6, 5*. Pour le premier chemin (le plus long), à défaut du prochain nœud sur le chemin du jeton (qui est *2*), le nœud *1* choisit *4* (choix aléatoire) et lui envoie le message de ressources (afin de chercher le bon chemin du jeton), qui, également pour la même raison, choisi *2*, qui choisi *7*. Le nœud *7* n'ayant pas d'autres voisins retourne le message à *2*. Celui-ci choisi l'autre voisin (non exploré) *6*, ainsi le message retrouve son chemin vers *5* (noeud détenteur du jeton). Pour le dernier chemin (le plus court), le nœud *1* choisit le nœud *3* au lieu de *4*; ainsi, le message retrouve le chemin du jeton. Dans la Figure 38(b), si on suppose maintenant que le chemin du jeton partant de *7* est *2*, *6*, *3*, *4*, *3*, *6*, *5* et si le numéro d'ordre du nœud *7* est *10*, les nœuds traversés par le jeton ont respectivement comme numéros d'ordre *11*, *12*, *13*, *14*, *15*, *16*, *17*. Le message de ressources issue du nœud *7* avec un numéro d'ordre *10* sera envoyé à *2*, qui le passe à *6* qui l'envoie à sa destination, i.e. le nœud *5* (qui est le dernier nœud auquel le jeton est envoyé par *6*), c'est un chemin optimal! Pour supporter le cheminement arborescent (i.e. sans boucle), on fait recours à une structure de trace distribuée qui permet de se souvenir des nœuds desquels chaque message de libération (connu par son numéro d'ordre) est reçu la première fois.

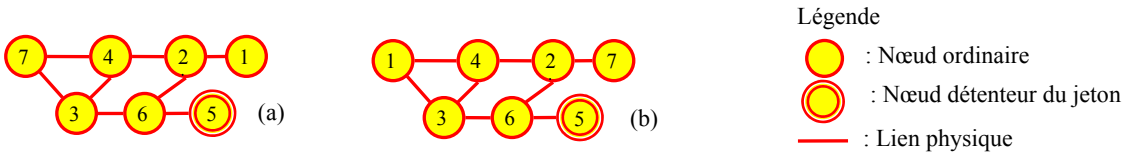


Figure 38: *Exemple d'acheminement de ressources.*

3.5. Mise à jour des files d'attente

La mise à jour des deux files entretenues (i.e. la file d'attente Q_i locale à chaque nœud *i* et celle transportée par le jeton) se fait lors de la réception du jeton, d'une nouvelle requête ou d'un message *IsThereToken ()* ou à la sortie d'une SC. Globalement, les mises à jour de ces files permettent de purger les requêtes déjà satisfaites ou anciennes et de se rendre compte de celles qui sont nouvelles ou présentant de meilleures routes.

A cet effet, le jeton est amené à transporter, entre autres, des informations sur les requêtes satisfaites. Cette solution engendre l'augmentation de la taille du jeton. Une solution d'échange pourrait être de supprimer chaque requête selon un critère prédéfini (par exemple, quand elle atteint un certain délai de séjour dans la file). Cette dernière solution est viable puisque des mécanismes (tels que les relances, la satisfaction des nœuds intermédiaires, …) permettent de remédier à ses effets secondaires éventuels.

3.6. Régénération de ressources

Cette opération est réalisée à deux occasions différentes:

- Dès qu'un nœud demandeur de ressources et détenteur de jeton (i.e. $State_i$= *wait*) ne reçoit pas, après un délai fixé (i.e. *ResourceTimer*), les ressources manquantes issues des libérations effectuées par d'autres nœuds. En ce moment, ce nœud (qui est d'ailleurs le seul détenteur du jeton de la partition) est amené à diffuser un message *IsThereResources ()* dans toute sa partition (et met $Search\_Res_i$ à *true*). Ce message a pour rôle de demander aux nœuds détenteurs de ressources bloquées localement (i.e. Rel_State_i=*unable*) et à ceux en cours d'utilisation de ressources ($State_i$ =*inCS*), de les supprimer. Ensuite, et après un délai de stabilisation fixé (i.e. *RecoverTimer*), ce nœud *privilégié* régénère les *k* nouvelles ressources de sa partition. De ce fait, la distribution de ressources reprend son cours normal.
- Au moment de la régénération du jeton (qui sera vue dans la Section 4.1): L'opération de régénération de ressources est également nécessaire. En effet, le jeton perdu ou déplacé vers une autre partition peut prendre (de même que certains nœuds) avec lui certaines ressources. L'opération de régénération du jeton *s'accompagne* alors de celle des ressources. C'est ainsi, qu'un nœud qui crée le jeton est amené à régénérer les *k* ressources. Afin d'assurer que seulement *k* ressources existent dans la partition en cours, deux recommandations sont nécessaires : Premièrement, chaque nœud détenteur des ''anciennes'' ressources doit les supprimer même s'il est en SC. Deuxièmement, la distribution de ressources régénérées se fait après l'écoulement d'un certain délai (i.e. *SuspecionTimer*). Ce délai permet de temporiser les nœuds en cours d'utilisation de ressources pour en finir avec (ce qui assure la sûreté).

4. Le partitionnement et la fusion

Chaque partition est connue par une identité (i.e. *Tid,* celle du jeton qui circule en elle,) afin de permettre la détection de fusions de partitions et donc de ne garder qu'un seul jeton. L'estampille du jeton (*i.e. TidNS*) est aussi utilisée afin de distinguer les partitions dont les jetons ont été créés par le même nœud. Ces informations sont connues de tous les nœuds de la partition.

4.1. Le partitionnement, la perte de jeton et de ressources

Dans le cas où le nombre de tentatives d'un nœud *i* demandeur de jeton atteint sa limite (i.e. $attempts_i$= *Req_Threshold*), s'il n'est pas en état de recherche de ressources, i.e. $Search\_Res_i$= *false* (auquel cas le jeton est présent), il commence à soupçonner la non présence de ce dernier (i.e. partitionnement ou perte du jeton), et change la manière de demander la SC. En effet, aucun rayon d'envoi n'est calculé, sa demande va être diffusée sur tout le réseau; de plus, cette dernière n'est plus véhiculée par un message *Request ()*, mais par un message *IsThereToken ()*. Le nœud *i* se *bloque* ensuite (i.e. met $Search\_Token_i$ à *true*, ce qui le mène à ne plus générer de demandes) en attente de la réception d'un message de retour *ThereIsToken ()* pendant une durée de temps limitée (i.e. *SuspicionTimer),* au bout de laquelle le traitement du partitionnement et la régénération de ressources sont appliqués. Si le message *IsThereToken ()* rencontre le jeton au niveau d'un nœud donné, celui-ci est amené à diffuser dans tout le réseau le message *ThereIsToken ()* mais après un délai de stabilisation *StabilisationTimer* pour éviter le blocage de certains nœuds, après leur déblocage, par des messages *IsThereToken ()* non *éteints*. Si avant la fin du délai *SuspicionTimer*, le message *ThereIsToken ()* est reçu, le nœud *i* se *débloque* (met $Search\_Token_i$ à *false* et exécute l'action éventuelle en attente enregistrée dans $Block\_state_i$).et continue la propagation de ce message. Par contre, si aucune réponse *ThereIsToken ()* ne lui est parvenue avant l'expiration de ce

délai, alors puisque sa demande sous-entend aussi une proposition de création d'un nouveau jeton, le nœud *i*, s'il est seul à avoir demandé la création d'un jeton, le crée, régénère les *k* ressources et le protocole redémarre pour cette partition. En cas de propositions simultanées de création de jetons, seule celle du nœud, par exemple, de plus petit identificateur aboutie (cette méthode est similaire aux mécanismes d'*élection* selon le critère d'extremum [123, 97, 144]), et le jeton créé par ce nœud extremum aura comme identité celle de ce nœud et comme estampille le numéro de séquence proposé par ce noeud. Autrement dit, *le seul nœud qui trouve l'identité suggérée au jeton égale à la sienne a à sa charge la création du jeton et la régénération des k ressources.* Les autres prennent alors acte du nouveau jeton et annulent les ressources éventuellement en leurs possessions.

Quand un nœud *j* reçoit un message *IsThereToken ()*, il le traite comme une requête normale (i.e. il essaye de l'insérer dans sa file) et si *j* est à l'état de recherche de ressources, cette recherche du jeton est stoppée puisque la recherche de ressources émane du nœud détenteur du jeton! Dans le cas contraire, s'il n'est pas détenteur du jeton, il considère ce message comme une exploration pour l'élection à la création d'un jeton. Ceci le mène à participer à cette élection en mettant à jour les variables appropriées, et continue éventuellement la propagation de ce message et se bloque (, en déclenchant son propre temporisateur *SuspicionTimer,*) s'il ne l'était pas. Dans le cas où *j* serait détenteur du jeton (i.e. pas de partitionnement), il ne traitera que le premier message reçu. Ce traitement se résume à *une attente* (i.e. *StabilisationTimer*) dans laquelle *j* s'assure que les différentes diffusions simultanées de ce type de message se soient arrêtées (dans cette période d'attente, *j* ne fait pas passer le jeton au prochain). Ensuite, il diffuse le message *ThereIsToken ()* dans tout le réseau.

4.2. La fusion

La fusion est détectée à l'occasion de créations de liens. A chaque création de lien, les deux nœuds le formant s'échangent les identifications des jetons qui leurs sont associés. En cas de duplication de jetons, un seul restera valide et l'autre sera supprimé.

Dès qu'un nœud *i* détecte un nouveau voisin, il lui transmet un message *LinkInfo ()* qui contient des informations sur son état de la manière suivante: Si *i* n'est pas bloqué, seules les informations (identité et estampille) sur le jeton utilisé par *i* et les ressources en sa possession si celles-ci sont bloquées à son niveau sont transmises. Si *i* est bloqué, le message *LinkInfo ()* inclus aussi les informations du message *IsThereToken ()* traversant ou généré par *i,* ou les informations du message *IsThereResources ()* si par contre *i* est en état de recherche de ressources. Ce mécanisme permet de remédier aux cas de déconnexions momentanées: Un des deux nœuds peut être détenteur du jeton ou de ressources bloquées! On peut constater que pour un lien créé, deux messages sont échangés (par cause de symétrie), et donc dans un souci d'optimisation du nombre de messages *LinkInfo ()*, on impose la règle suivante: Si *i* se trouve dans l'une au moins des situations : il est bloqué, il recherche les ressources ou il dispose de ressources bloquées, alors il transmet systématiquement le message *LinkInfo ()*, mais si au contraire il est dans un état normal, un seul nœud (choisit de manière déterministe,) prend à sa charge la transmission de ce message.

Quand un nœud *j* reçoit un message *LinkInfo ()* de la part de *i*, s'il trouve que le jeton de *i* est le même que celui de *j* (i.e. le lien créé est alors interne à la partition) et selon les informations reçues, *j* exécute un traitement équivalent à celui exécuté lors de la réception d'un message *IsThereToken ()* ou *IsThereResources ()* ou *Release ()*. Par contre, si les jetons sont différents alors il y a eu une fusion et un jeton doit donc être supprimé. Le choix porte, par exemple, sur le jeton le plus récent dans le cas de jetons créés par le même nœud ou sinon, sur celui avec l'identité la plus grande). Si le jeton à supprimer est celui de la partition de *i*, *j*

lui envoie un message *LinkInfo ()* normal s'il ne l'a pas fait auparavant. Dans le cas contraire, *j* diffuse le message *DeleteToken ()* dans sa partition après avoir fait le même travail que chaque noeud qui reçoit ce message.

Quand un nœud *r* reçoit un message *DeleteToken ()*, si le jeton de la partition locale est moins prioritaire, alors il met à jour ses informations locales par celles reçues, réinitialise son contexte (vide sa file, annule ses ressources éventuelles, se débloque s'il était bloqué, désactive ses temporisateurs éventuels, etc…) et continue la diffusion de ce message. De plus si *r* est détenteur du jeton, il le supprime même s'il est en SC, et cela ne pose pas de problème car à la sortie de la SC, il se trouvera sans jeton. Dans tous les cas, si *r* est demandeur de ressources ou à l'état *wait* (qu'il change à *req*), il arme le temporisateur *RequestTimer* pour relancer sa requête plus tard.

5. Exemple d'exécution et discussion

5.1. Exemple d'exécution

La Figure 39 montre un scénario du fonctionnement du service de *h- k-* EM d'une manière qui permet de faciliter la présentation. Notons que les actions qui n'engendrent pas d'effets ne seront pas citées. Initialement (Figure 39(a)), le nœud *4* détient le jeton à l'état de repos avec *12* ressources libres. Les nœuds *2, 7* et *1* demandent respectivement *2, 2* et *5* ressources. Ils diffusent leurs requêtes dans tout le réseau. Admettons que les estampilles de toutes les requêtes sont identiques. Comme toutes ces requêtes atteignent le nœud *privilégié 4* avec la même estampille, c'est le nombre de sauts augmenté du nombre de ressources demandées qui va déterminer le prochain nœud privilégié, il s'agit du nœud *2*. Le jeton lui est envoyé accompagner des ressources libres (i.e. *12*). Arrivant au nœud *2*, celui-ci se sert et puisque les requêtes des nœuds *7* et *1* sont toutes les deux reçues au niveau du nœud *2*, le prochain noeud privilégié est alors *7*. Le nœud *2* lui envoie le jeton sur la route *1*, *3*, *7* en lui verrouillant les ressources demandées et accède à sa SC. Arrivant au nœud *1*, le jeton lui permet de se servir car le nombre de ressources non verrouillées (i.e. *TAvail_res - TLock_Res= 8*) suffissent pour satisfaire sa requête et le fait suivre sur le chemin prévu. A l'arrivée du jeton au niveau du nœud *7*, celui-ci se sert et accède en SC en gardant le jeton à l'état de repos (puisque sa file est vide). Ensuite, le nœud *1* libère les ressources préalablement acquises, celles-ci suivent les traces du jeton sur le chemin 3, 7.

Dans la Figure 39(b), le nœud *7* sort de sa SC et ajoute les ressources à celles du jeton détenu localement; ensuite, le nœud *4* exprime une demande sur *11* ressources avec, par exemple, un rayon $R_4 = 2$. Sa requête est envoyée sur les chemins des noeuds *7*, *1* et *2* puisqu'il a déjà reçu leurs requêtes. Le jeton lui est envoyé par le nœud *7* sur le chemin *3*, *4*. En recevant le jeton, il réalise que le nombre de ressources libres du jeton (i.e. *Tavail_res=10*) ne suffit pas sa requête, il retient alors le jeton et passe à l'état d'attente de ressources manquantes (i.e. $State_4 = wait$). Par la suite, le nœud *2* libère ses *2* ressources, grâce au chemin tracé par le jeton et les numéros d'ordre des nœuds sur ce chemin qui sont restés monotones croissants, le message de libération de ressources suit le chemin *1*, *3*, *4* au lieu du long chemin *1*, *3*, *7*, *3*, *4* (grâce à $Next_3$ sur le chemin du jeton qui devient *4*). L'arrivée des ressources au niveau du nœud *4* lui permet d'accéder à sa SC en gardant le jeton avec *1* ressource libre

Dans la Figure 39(c), où des changements de liens ont eux lieu, les nœuds *7* et *3* demandent respectivement *5* et *6* ressources (avec $R_7 = R_3 = 2$). En supposant que la requête du nœud *7* arrive en premier au niveau du nœud *4*, celui-ci lui envoie le jeton sur le chemin *3*, *7*. Arrivant

au niveau du nœud 3, celui-ci ne peut pas se servir puisque le nombre excédent de ressources du jeton reçu (i.e. *TAvail_res - TLock_Res*) est nul. Il se contente alors de faire suivre le jeton sur son chemin prévu. Lorsque le jeton arrive au niveau du nœud *7*, celui-ci le garde localement en passant à l'état *wait*.

Dans la Figure 39(d), où d'autres changements de liens ont eux lieu, le nœud *4* sort de sa SC mais perd son prochain sur le chemin du jeton (qui est le nœud *3*), il choisit alors un voisin quelconque (, soit 5,) et lui envoie les ressources. Le nœud *5* à défaut d'autre voisin retourne les ressources au nœud *4*. Celui-ci choisit l'autre voisin non exploré (i.e. le nœud *1*), qui choisit le nœud *7*, ainsi le détenteur du jeton est retrouvé. Ces ressources permettent à ce dernier d'accéder à sa SC en envoyant le jeton au nœud *3* qui se sert et garde le jeton avec *1* ressource libre. Entre-temps, le nœud *1* demande *1* ressource (avec $R_1 = 2$) ; le jeton est alors reçu par ce dernier sur le chemin *7*, *1* ce qui lui permet d'accéder à sa SC.

Dans la Figure 39(e) où un partitionnement a eu lieu, le nœud *5* exprime une requête sur *8* ressources (avec $R_2=2$). Le jeton lui est envoyé mais à défaut de ressources, il passe à l'état d'attente de ressources en déclenchant le temporisateur *ResourcesTimer*. Le nœud *7* sort de sa SC, son message de ressources libérées explore la partition, par exemple, avec le chemin *3*, *6*, *3*, *2*, *3*, *7*. Les ressources sont ainsi gardées par le nœud *7* à l'état *unable*. Ensuite, les nœuds *6* et *2* demandent chacun 4 ressources.

Dans la Figure 39(f), le nœud *7* rompe son lien avec le nœud *3* et forme un nouveau lien avec le nœud *5*. En s'échangeant les informations à travers les messages *LinkInfo ()*, le nœud *5* reçoit les *5* ressources bloquées au niveau du nœud *7*. Le nombre total de ressources acquises (qui est *5*= (*0*+*5*)) ne suffit pas la requête du nœud *5*, il reste toujours au même état en relançant son temporisateur. Supposons maintenant que le temporisateur du nœud *5* expire, il diffuse alors le message *IsThereResources ()*. Le nœud *1* qui est en SC annule ses ressources. Le nœud *1*, en sortant de sa SC, se trouve sans ressources, il n'aura donc pas à envoyer le message de libération de ressources! Après le délai *RecoverTimer*, le noeud *5* régénère toutes les ressources et accède à sa SC.

Parallèlement à ces opérations, les nœuds *6* et *2* redemandent plusieurs fois le jeton sans l'avoir. Ils procèdent alors à la recherche du jeton en diffusant le message *IsThereToken ()*. Après l'expiration des temporisateurs *SuspicionTimer*, seul le nœud *2* est amené à créer le jeton et à régénérer les *k* ressources en accédant à sa SC. Les autres nœuds prennent acte du nouveau jeton. Parallèlement et à l'expiration de son propre temporisateur *SuspicionTimer*, le nœud *3* est invité à annuler ses ressources. Par la suite, le nœud *6* régénère sa requête après l'expiration du délai *RequetTimer* qu'il a déclenché lors de la prise en compte du nouveau jeton. Le jeton lui sera alors envoyé, ce qui lui permet d'accéder lui aussi à sa SC (Figure 39(g)).

Dans la Figure 39(g), la fusion des deux partitions a eu lieu à travers le lien qui s'est formé entre les nœuds *7* et *3*. Seul le nœud *3* envoie un message *LinkInfo ()* à *7* (puisque les deux nœuds sont à l'état normal). Supposons que le jeton le mois prioritaire est celui de la partition du nœud *7*, donc un message *DeleteToken ()* est diffusé par le nœud *7* dans sa partition. Ce message permet au nœud *5* de supprimer son jeton avec ses *4* ressources libres et d'annuler toutes les ressources en sa possession qui sont au nombre de *8*.

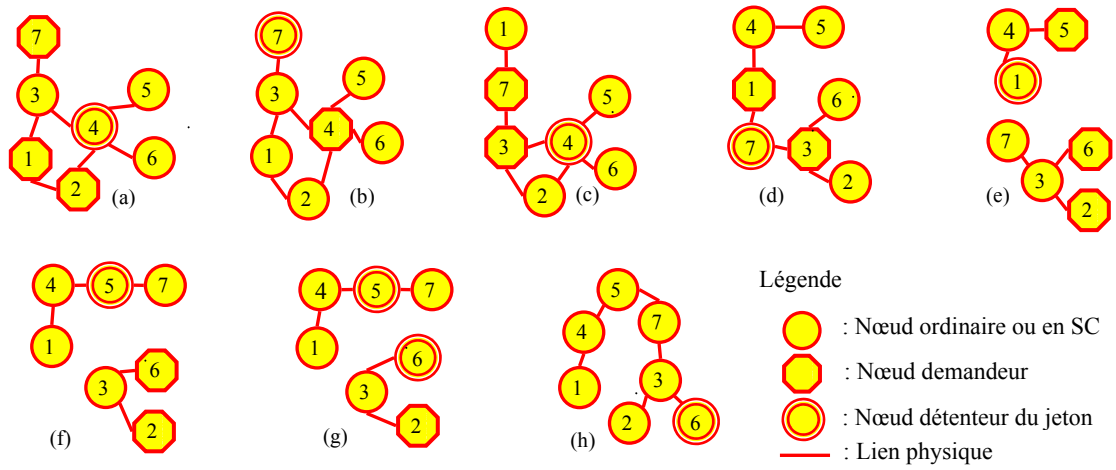


Figure 39: *Un exemple d'exécution du protocole de h* parmi *k exclusion mutuelle.*

5.2. Discussion

Pourquoi le message LinkInfo () peut transporter dans certains cas, les informations d'un des messages IsThereToken () ou IsThereResources () ?

A partir du principe que les messages du type *IsThereToken ()* ou *IsThereResources ()* doivent être propagés dans tout le réseau, un nœud peut s'isoler momentanément et donc peut ne pas recevoir ce message. Dés lors qu'il intègrera le réseau, il sera informé en véhiculant dans le message *LinkInfo ()* les informations correspondant à ces deux types de messages.

Pourquoi le message LinkInfo () peut transporter dans certains cas, les informations d'un message Release ()?

Etant donné que les ressources libérées doivent être envoyées (acheminées) au détenteur du jeton pour de futures réutilisations et puisque certains nœuds peuvent détenir localement des ressources bloquées, la découverte d'un nouveau voisin leur donne une opportunité pour acheminer ces ressources vers leur destination.

Pourquoi régénérer les ressources au lieu d'attendre la réception de ressources bloquées?

Lors des déconnexions fréquentes, les routes vers les destinations peuvent se briser aussi fréquemment, ce qui rend l'opération de récupération des ressources non perdues délicate, coûteuse en temps et presque impossible dans certaines situations. D'autre part, le manque de ressources peut signifier soit leurs pertes, soit leurs blocage quelque part dans une région de la partition courante, soit les deux et tout cela de manière non contrôlée. Le fait qu'il faille avoir à tout moment k ressources au plus dans une partition donnée, une solution conduit à régénérer les k ressources au niveau du seul nœud détenteur du jeton. Par conséquent, l'annulation des ressources bloquées au niveau de certains nœuds, des ressources libérées après les sorties de SC et celles en cours d'acheminement devient impérative. Pour des raisons de sûreté, la régénération des ressources se fait après un délai d'attente (i.e. *RecoverTimer)* afin de permettre aux nœuds qui utilisent actuellement des ressources de quitter leurs SC.

Pourquoi déclencher le temporisateur RecoverTimer au niveau de chaque nœud ?

Dans le but de prévenir les cas d'isolements temporaires et de maintenir au plus k ressources dans une partition donnée, un nœud qui reçoit le message *IsThereResources ()* se met en état de recherche de ressources (i.e. $SearchingResources_i := vrai$) pendant une durée de temps

limitée par le temporisateur *RecoverTimer*. Ce temps d'attente est nécessaire pour informer tous les nœuds de la partition de cette recherche y compris ceux isolés temporairement. Un nœud isolé est alors informé par le message *LinkInfo ()* reçu lors de la réintégration du réseau, ce qui va lui permettre d'annuler les ressources éventuellement bloquées localement !

Quelle est la taille de chaque message de contrôle?

Chaque type de message de contrôle du service de *h*-*k*-EM transporte des informations utiles au fonctionnement. La taille d'un message dépend des informations transportées. Certains messages transportent l'ensemble des identités des nœuds visités par celui-ci et éventuellement la route planifiée à suivre ou la route suivie. Dans le cas échéant, la taille de chaque information est *N*, i.e. le nombre courant de nœuds du réseau. D'autres informations peuvent aussi être transportées, mais la plupart des cas elles sont minimes, à l'exception du message *Token ()*. Dans ce message, les paramètres *TKnownNS, TReq_Nodes and TReq_Res* peuvent être chacun de taille *N* dans le cas échéant.

6. Etude de correction.

La majorité des outils produits par le service de *h*-*k* EM ont pour but la satisfaction des deux propriétés, en l'occurrence la sûreté et la vivacité. En premier, on s'intéressera à la sûreté. Cette propriété, rappelons le, implique que chaque ressource ne peut être utilisée que par un processus à la fois et au maximum *k* ressources sont accessibles à un instant donné. Rappelons qu'un jeton est géré au niveau de chaque partition. De plus, le seul nœud qui le détient a le droit de satisfaire les requêtes. Si le protocole assure qu'à tout moment un seul jeton avec au maximum (étant donné que certains nœuds peuvent tomber en panne) *k* ressources réside au niveau de chaque partition, la propriété de sûreté est alors vérifiée. Evidemment, dans l'absence de la création de jeton et du fusionnement de partitions, cette propriété est assurée étant donné qu'un seul jeton circule dans la partition courante. Nous allons étudier la sûreté dans trois situations : A la création d'un nouveau jeton après perte, à la suppression d'un ou plus de jeton(s) durant le fusionnement de partitions et à la régénération de ressources après perte(s).

Pour la première situation (voir aussi la Section 4.1), supposons qu'un ou plusieurs nœuds sont en cours de recherche du jeton. Tous les noeuds de la partition courante seront informés au même titre que ceux en déconnexion temporaire (par l'utilisation des temporisateurs qui attendent leurs reconnexions et des messages d'informations *LinkInfo ()*). Si un nœud détient le jeton (ou le reçoit pendant qu'il est dans l'état de recherche du jeton), après un délai *StabilisationTimer*, ce noeud informe tous les autres de la non perte du jeton et ainsi la recherche en cours s'arrête. (Notons que la recherche du jeton est aussi arrêtée si une recherche de ressources est en cours. Notons aussi que la valeur du temporisateur *StabilisationTimer* est calculée en sorte qu'elle permette à tous les nœuds d'être informés avant l'expiration des temporisateurs *SuspicionTimer*). Autrement, à l'expiration de son temporisateur, un nœud doit créer le nouveau jeton avec une nouvelle identification. Ce *seul* nœud a été choisi par un processus d'élection qui a accompagné la recherche du jeton. Comme résultat de cette élection, tous les nœuds sont informés de l'identité du nouveau jeton. Par conséquent, un seul jeton existe dans la partition ce qui vérifie la sûreté.

Pour la seconde situation (voir aussi la Section 4.2), supposons deux partitions chacune avec son propre jeton identifié de manière unique. Pour faciliter la présentation, considérons que chaque partition est colorée différemment (en *bleu* ou en *rouge*) et cette couleur correspond à ce jeton. Admettons maintenant qu'un lien est établi entre les nœuds *i* et *j* appartenant chacun à l'une de ces deux partitions. Quelque soit la politique qui gère la priorité

des jetons, un jeton devient moins prioritaire par rapport à l'autre, par exemple le *bleu*. Par conséquent, le nœud *j* va diffuser dans sa partition le message *DeleteToken ()*. Ainsi, à la fin de cette dessimination, tous les nœuds vont être de couleur *bleu*. Un délai peu s'écouler entre le moment du fusionnement et la suppression effective du jeton en dépendant de la vitesse de propagation du message de suppression du jeton. Des mécanismes ont été définis pour remédier aux effets éventuels de cette période transitoire (voir Section 7.3)

Pour la troisième situation, nous devons considérer deux cas différents : Le premier cas est quand un nœud demandeur de ressources et détenteur de jeton est en cours de recherche des ressources manquantes (voir aussi les Sections 2.2.9 et 5.2). Ce nœud informe tous les autres de sa partition (similairement à la recherche du jeton). Ce message de recherche incite alors les nœuds qui, en ce moment, utilisent ou seulement détiennent des ressources à les supprimer. La régénération de ressources se fera après un délai de stabilisation (i.e. *RecoverTimer*) afin de s'assurer que tous les nœuds soient sortis de leurs SC et aient détruits leurs anciennes ressources. A cet effet, la distribution de ressources reprend son court normal. Un cas spécial peut se produire: Quand un message en transit de libération des anciennes ressources est reçu par un nœud déjà informé sur la recherche de ressources. Comme ce nœud est dans l'état spécial de recherche de ressources (i.e. $Search\_Res_i = true$), ce message sera automatiquement ignoré. Tous ces mécanismes assurent la propriété de sûreté. Le second cas est le moment de la régénération du jeton (voir Section 4.1). L'opération de régénération du jeton est alors *accompagnée* par celle des ressources. Ainsi, un nœud qui crée le jeton est amené à régénérer *k* ressources. Afin d'assurer l'existence de seulement *k* ressources dans la partition en cours, deux recommandations sont prise en compte. Premièrement; chaque nœud détenant d'anciennes ressources est prié de les détruire même s'il est toujours en SC. Deuxièmement; la distribution des nouvelles ressources est reprise après un délai de temps fixe (i.e. *SuspicionTimer*). Ce délai permet aux nœuds en cours d'utilisation de ressources d'en finir. La propriété de sûreté est alors assurée.

Le protocole ne permet pas l'interblocage étant donné que chaque noeud demandeur de ressources et détenteur de jeton, si le nombre de ressources du jeton n'est pas suffisant pour satisfaire sa requête, garde le jeton jusqu'à la réception des ressources manquantes. Autrement dit, l'allocation en parties de ressources n'est pas permise.

Considérons maintenant la propriété de vivacité qui implique que chaque nœud demandeur de ressources arrive à les acquérir. Plusieurs mécanismes ont été appliqués pour favoriser cette propriété. Le premier mécanisme est l'utilisation du rayon d'envoi et le transport de requêtes par le jeton (voir Section 2.2.1). Le rayon d'envoi permet de créer une cascade de chemins vers le nœud détenteur du jeton. Ce qui implique que ce nœud est indirectement au courant des requêtes lointaines. Mais, ce mécanisme est insuffisant pour satisfaire toutes les requêtes, d'où la collecte de requêtes rencontrées par le jeton lors de son parcourt. Par conséquent, tous ces nœuds seront éventuellement satisfaits en dépendant de la mobilité et des pannes. Le second mécanisme est la relance de requêtes (voir Section 2.2.2). En effet, un nœud qui ne reçoit pas le jeton durant une période fixe doit relancer sa requête. Afin de garder son ancienneté, cette requête est relancée avec la même estampille. Dans le cas échéant, ce processus est répété un certain nombre de fois (i.e. *Req_Threshold*) avant de suspecter la perte du jeton. Le troisième mécanisme est évidemment la recherche et la régénération du jeton (voir Section 4.1). Quand la perte du jeton est détectée, celui-ci sera régénéré et par suite les nœuds demandeurs régénèrent leurs requêtes après un délai fixe. Le quatrième mécanisme est la régénération de ressources (voir Section 2.2.9). Quand un nœud demandeur et détenteur de jeton ne reçoit pas les ressources manquantes après un seuil fixe, il informe tous les nœuds de sa partition et régénère les ressources manquantes. Ce nœud va alors nécessairement exécuter sa SC. Deux autres mécanismes sont aussi utilisés, ils permettent d'éviter la famine de nœuds.

Le premier est la politique de satisfaction de requêtes (voir Section 2.2.3) qui utilise l'ancienneté des requêtes. Au court du temps, les anciennes requêtes deviennent nécessairement plus prioritaires et par suite, servies en premier. Le second mécanisme est le verrouillage de ressources (voir Section 2.2.4). Quand un nœud est sélectionné comme prochain privilégié, les ressources requises par ce nœud sont déduites du nombre total transporté par le jeton et réservée à ce nœud. Les nœuds intermédiaires ne peuvent utiliser que le surplus, s'il y a lieu.

Pratiquement, si on se fie aux résultats de simulations (qui seront détaillés dans la Section 7.2), nous constatons que la majorité des requêtes d'admission en SC sont satisfaites au maximum après *0.5* redemande. D'autre part, le taux de satisfaction est proche de *1*. Selon les traces d'exécutions, les nœuds non satisfaits sont ceux qui expriment leurs requêtes vers la fin d'exécution de la simulation. Ces requêtes tardives doivent évidemment attendre les libérations de ressources qui n'auront pas lieu avant la fin de la simulation.

7. Simulation

L'évaluation des performances du protocole est effectuée à l'aide de l'outil NS2 (*Network Simulator 2*) [136] version 2.26 sous le système d'exploitation LINUX MANDRAKE 8.2. La codification du protocole est effectuée à l'aide du Langage objet Otcl.

7.1. Environnement de simulation et paramètres

L'environnement dont les différentes simulations ont été effectuées, peut être décrit à travers un certain nombre de paramètres. Les paramètres constants sont: la surface de mouvement est de *1000 × 500 mètres*; le nombre de nœuds dans le réseau est de *25*; La durée de la SC est de *0.1 seconde*; le rayon de communication entre nœuds mobiles est de *250 mètres*; le temps de simulation est de *100 secondes*. Le nombre de ressources du réseau est *10*. Les paramètres variables sont :

- La mobilité : Elle est donnée par le modèle aléatoire se basant sur le mouvement relatif des nœuds (voir Chapitre I). A partir de la formule de la mobilité, nous avons : mobilité faible : *0* < Mob <= *3* ; mobilité moyenne : *3* < Mob <= *8* ; mobilité élevée : *8* < Mob. Pour la simulation, nous avons utilisé quatre valeurs de mobilité : *0*, *1.5*, *5*, *10*. Avec les différents scénarios de mouvements réalisés lors de la simulation, il ressort que la connectivité moyenne varie entre *5* et *7* voisins par nœud.
- La charge : Elle est définie par une loi de poisson de paramètre λ dont les valeurs prises pour la simulation sont : *0.07, 0.10, 0.125, 0.167, 0.25, 0.5, 1* correspondant respectivement aux intervalles ($1/\lambda$) *15s*, *10s*, *8s*, *6s*, *4s*, *2s*, *1s* durant lesquels un nœud, après satisfaction, génère une nouvelle demande d'entrée en SC.

7.2. Résultats et interprétations

Avant d'entamer l'étude des performances du protocole, une série de tests préliminaires a permis de dégager des valeurs aux temporisateurs permettant un fonctionnement correct du protocole. Les valeurs sont : *RequestTimer=12* s ; *SuspicionTimer=20 s* : Temps nécessaire pour véhiculer le message *IsThereToken ()* à tous les nœuds ajouté au temps de réception du message *ThereIsToken ()* pour tous ces nœuds. Si la valeur du timer < *20*, alors il y aura une régénération du jeton, ce qui n'est pas tolérable. *ResourceTimer=6 s*, c'est le temps maximal durant lequel un nœud se met en attente du message *Release ()*, afin d'atteindre un court délai

de synchronisation et afin d'éviter la diffusion du message *IsThereToken ()* par les autres nœuds demandeurs; *StabilisationTimer* (resp. *RecoverTimer*) = *6 s*, c'est le temps nécessaire pour que le message *IsThereToken ()* (resp. *IsThereResources ()*) atteigne tous les nœuds du réseau. Si la valeur du temps est inférieur à *4* secondes, certains nœuds du réseau ne reçoivent pas le message *IsThereToken ()* (resp. *IsThereResource ()*).

L'ensemble des mesures réalisées se scindent en deux catégories : les mesures *''préalables''* et les mesures *''effectives''*. Les mesures préalables ont pour objectifs de quantifier certains comportements au sein du protocole afin d'interpréter efficacement les résultats liées aux mesures effectives. Les mesures effectives servent bien entendu à évaluer les performances du protocole. Toutes les mesures ont été réalisées en *absence* de partitionnement. Il est aussi à noter que toutes les mesures (à l'exception de celles de la Figure 40) utilisent la même légende pour la mobilité que celle de la Figure 41. Ceci nous a mené à la décrire une seule fois pour des raisons de simplification.

Mesures préalables

La proportion P

Elle représente la proportion de nœuds, dont une requête est présente dans la file d'attente d'un nœud voulant entrer en SC, qui doit être atteint par la requête de ce dernier. Ce paramètre est déterminé à l'aide de mesures effectuées sur le nombre de messages par SC *(nb_MSG/SC)* pour différentes charges et mobilités et ceci en variant à chaque fois la proportion *P* (10%, 25%,..100%). Le choix de *nb_MSG/SC* est motivé par le fait qu'en général les mesures effectives sont *interdépendantes*. La Figure 40 montre que *nb_MSG/SC* est sensible à la variation du pourcentage *P*. Les valeurs *25%* et *50%* de la proportion *P* engendrent un nombre minimum de messages pour la mobilité nulle (Figure 40(a)). Dans la Figure 40(b), toutes les proportions (excepté 100%) donnent un nombre de messages bas. Les valeurs *10%* et *25%* de la proportion *P* engendrent un nombre minimum de messages pour la mobilité moyenne (Figure 40(c)); Les valeurs *25%*, *50%* et *100%* de la proportion *P* engendrent un nombre minimum de messages pour la mobilité forte (Figure 40(d)). Il ressort que la valeur 25% de la proportion *P* est la plus adéquate d'après les graphes de la Figure 40. Cette valeur est donc fixée pour toutes les autres mesures de la simulation.

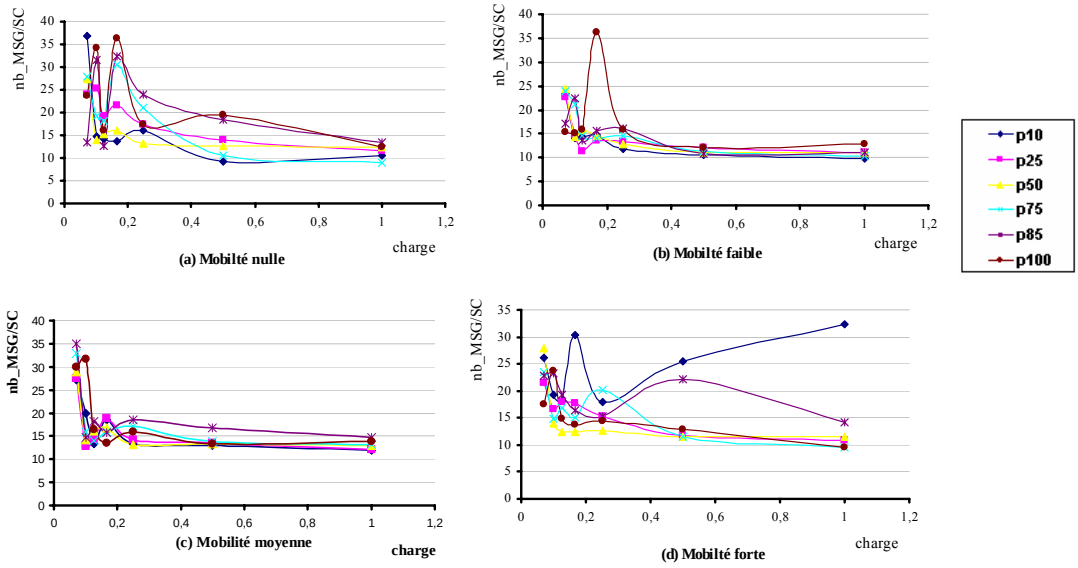


Figure 40 : *Graphes liés à la proportion P.*

Le taux de diffusion (nb_diffusions/demande)

Cette métrique est liée aux cas de diffusions de demandes quand la file de requêtes est vide. Elle est calculée comme suit : *nombre de diffusions / nombre de demandes*.

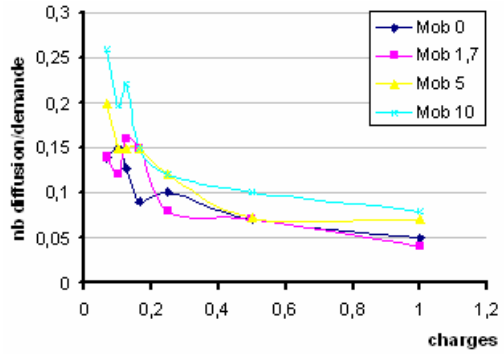


Figure 41 : *Taux de diffusion*.

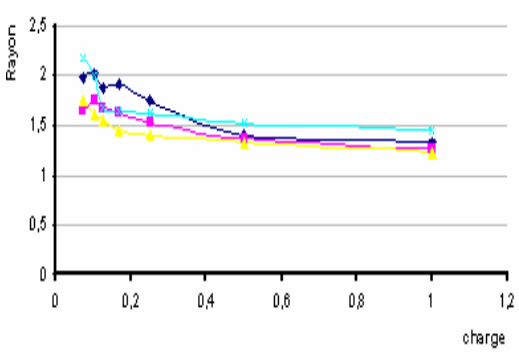


Figure 42 : *Rayon de d'envoi.*

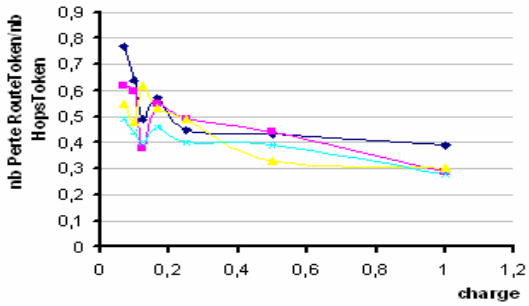
La Figure 41 montre que le taux de diffusion diminue avec l'augmentation de la charge de demande et converge vers la valeur 5% pour les moyennes et fortes charges. Lorsque la charge est faible, les files d'attente des nœuds demandeurs sont quasiment vides ce qui les mènent à diffuser leurs requêtes. Par contre, plus la charge est grande plus les files contiennent des requêtes et donc moins il y a la diffusion de requêtes.

Le rayon de d'envoi

Cette métrique indique le nombre de sauts moyen que doit parcourir une requête; son calcul est décrit dans la section 2.2.1. Dans la Figure 42, le rayon moyen d'envoi de demandes diminue avec l'augmentation de la charge et converge vers 1.2 sauts pour les moyennes et fortes charges. Ceci s'explique par le fait que plus la charge de demandes est importante, plus le nombre de requêtes présent dans la file est important ; d'où des demandes de nœuds proches en terme de nombre de sauts ce qui engendre une diminution du rayon. Remarquons que les résultats de la Figure 42 sont cohérents avec ceux de la Figure 41.

Le taux de perte de route du jeton (nb_NORouteHops/nb_tokenHops)

Cette métrique indique le nombre de fois où le jeton a perdu la route vers sa destination (i.e. le nombre d'envois avec l'état *NORoute).* Il est calculé comme suit : *nombre de pas du jeton avec l'état NORoute/ nombre total de pas du jeton.* D'après la Figure 43, le taux de perte de routes du jeton tend globalement vers la décroissance avec l'augmentation de la charge et ceci pour toutes les mobilités. D'autre part, les courbes pour les différentes mobilités sont proches les une des autres. Tous ces résultats sont dus au fait que plus la charge augmente (i.e. la majorité des nœuds deviennent demandeurs), plus le jeton parcoure de faibles distances afin d'atteindre les prochains destinataires. Néanmoins, le taux de perte de routes du jeton converge vers *30%*. Cette convergence peut s'expliquer d'une part, par le fait que la connectivité est relativement constante et d'autre part, par les ruptures de liens qui accompagnent le mouvement des nœuds.



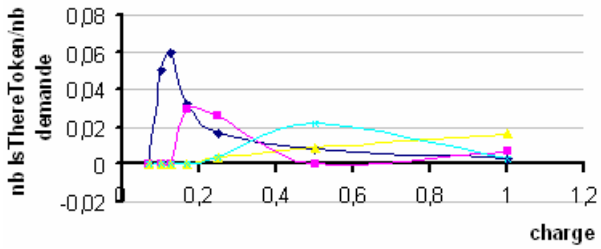


Figure 43: *Taux de perte de route du jeton.* **Figure 44:** *Le taux de recherche du jeton.*

Le taux de recherche du jeton (nb_IsThereToken/nb_demandes)

Cette métrique indique le nombre de recherches du jeton par demande. Elle est calculée par: *nombre de fois que le message IsThereToken est généré/ nombre de demandes.* D'après la Figure 44, le taux de recherche du jeton est globalement assez faible (*6*% au maximum) et converge vers une valeur très faible quelque soit la mobilité. Ce faible taux s'explique par le fait que tous les nœuds font partie d'une même partition ; donc, en général un nœud qui s'isole temporairement fini par rejoindre le réseau et véhiculer le jeton éventuellement bloqué localement.

Le taux de recherche de ressources (nb_IsThereResources/nb_demandes)

Cette métrique indique le nombre de recherches de ressources par demande. Elle est calculée par: *nombre de fois que le message IsThereResources est généré/ nombre de demandes.* D'après la Figure 45, Le taux de recherche de ressources est très faible et nul pour les faibles mobilités. Ce faible taux s'explique par le fait que tous les nœuds font partie d'une même partition ; donc, en général les nœuds qui s'isolent temporairement finissent par rejoindre le réseau et véhiculer les ressources éventuellement bloquées localement.

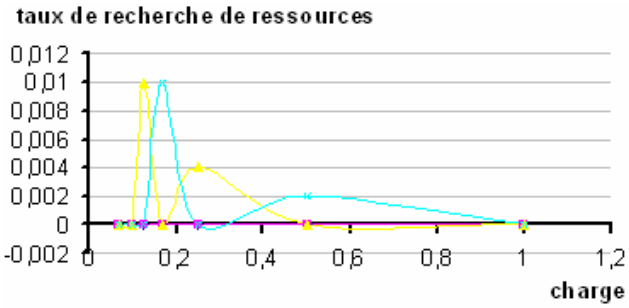


Figure 45 : *Taux de recherche de ressources.*

Mesures effectives

Le nombre de messages par SC (nb_MSG /SC)

Cette métrique représente le nombre moyen de messages nécessaires pour la réalisation d'une SC. Elle est calculée par : *((nombre de messages de demandes d'entrée en SC + le nombre de messages de redemandes + le nombre de sauts effectués par le jeton + nombre de messages LinkInfo () + le nombre de sauts effectués par le message release)) / nombre d'entrées en SC.* Comme le montre les graphes de la Figure 46, les graphes ont une même allure, i.e. le *nb_MSG/SC* est décroissant et donc ''inversement proportionnel '' à la charge. On note également, une certaine stabilisation au niveau des moyennes et fortes charges (*en moyenne 11 messages par SC*). Par ailleurs, nous pouvons remarquer que pour les moyennes et fortes charges, l'impact de la variation de la mobilité sur le *nb_MSG/SC* est insignifiant,

contrairement à ce qui se passe pour les faibles charges où les tracés présentent des pentes différentes. La réduction du nombre de messages est conjointement liée au rayon d'envoi de demandes. Ceci dit, plus la charge de demandes est importante, plus le rayon d'envoi de demandes diminue, d'où des demandes de nœuds proches en termes de nombre de sauts très probable. Par conséquent, le jeton effectue le minimum de sauts pour satisfaire les nœuds demandeurs. D'autre part, les messages *Release ()* suivent le chemin du jeton et donc effectuent de même un nombre restreint de sauts. Ces résultats sont de plus, confirmées par toutes les mesures préalables.

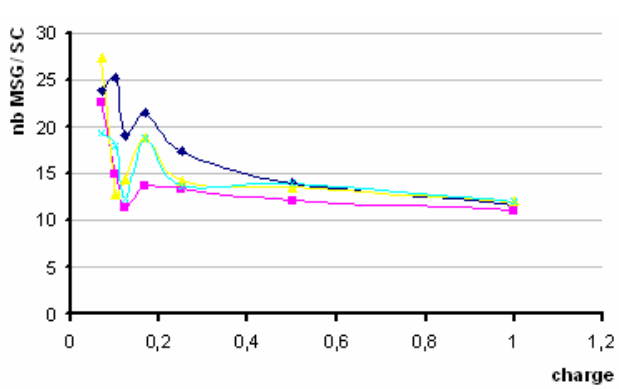


Figure 46: *Nombre de messages par SC.*

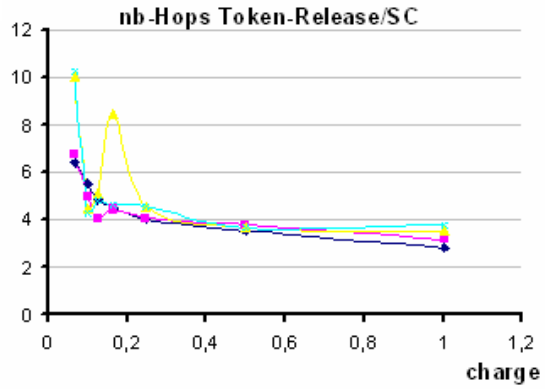


Figure 47: *Délai de synchronisation.*

Le délai de synchronisation (Délai_Syn)

Il indique le nombre moyen de sauts effectués par le jeton et forcément par les messages *Release ()* entre deux SC. Il est calculé par: (*nombre de sauts effectués par le jeton + le nombre de sauts effectués par le message release) / nombre d'entrées en SC*. D'après la Figure 47, les allures des graphes tendent vers une décroissance commune du délai de synchronisation en convergeant vers une valeur moyenne de *2.5* sauts. Un rapprochement des graphes est observé au niveau des moyennes et fortes charges. En contrepartie, l'influence de la mobilité est observée au niveau des faibles charges. De par sa nature, le protocole cherche à satisfaire les nœuds les plus proches (en nombre de sauts). Aussi, chaque nœud demandeur se trouvant sur la route du jeton sera potentiellement satisfait et donc, pour les fortes charges, le jeton et par conséquence les messages *Release ()* circulent la plupart du temps entre des nœuds demandeurs (qui sont proches les uns des autres), ce qui explique le court délai de synchronisation quelle que soit la mobilité. En ce qui concerne les faibles charges (de *0.07* à *0.25*), le jeton circule davantage entre des nœuds non demandeurs, afin d'atteindre ceux qui veulent accéder à la SC. Ceci dit, plus la mobilité est élevée, plus le risque de cassures de liens et donc de ruptures de routes (voir Figure 43) sont plus importantes, et donc dans ces conditions, le jeton va passer beaucoup plus de temps à chercher les demandeurs de SC avant de les atteindre; de même, les messages *Release ()* qui suivent les traces du jeton vont parcourir de longs chemins; tous ces faits expliquent l'augmentation du délai de synchronisation.

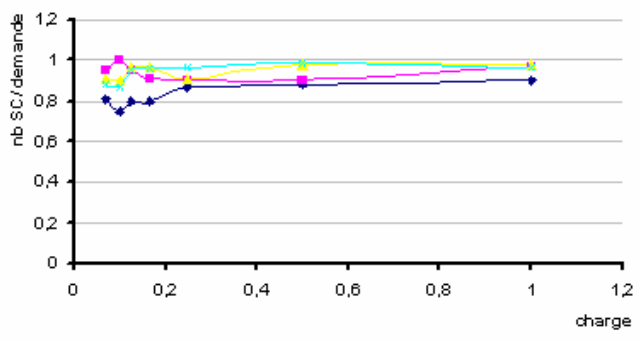


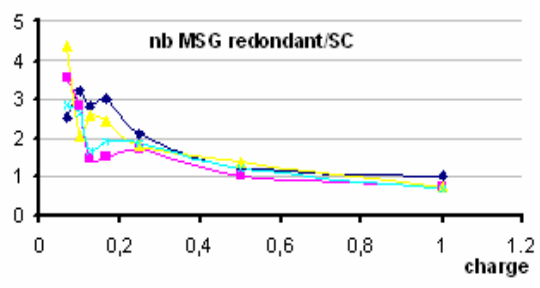
Figure 48 : *Taux de satisfaction.*

Le taux de satisfaction (nb_SC / demande)

Cette métrique indique le taux de satisfaction moyen des demandes d'entrées en SC. Elle est calculée de la manière suivante: *nombre d'entrées en SC/ nombre de requêtes,* les redemandes ne sont naturellement pas considérées. Pour cette métrique, dans la Figure 48 on remarque un rapprochement entre les différents graphes avec une allure presque horizontale et convergente vers un taux de satisfaction de *1*. Cependant, l'allure des graphes correspondant aux mobilités faible et nulle (*0* et *1.7*) présente une légère différence comparativement aux deux autres graphes, et cela pour l'intervalle de charge allant de *0.07* à *0.16*. Particulièrement dans le cas de mobilité nulle, le taux de satisfaction est légèrement inférieur par rapport aux autres mobilités. Finalement, globalement l'influence de la mobilité est presque insignifiante pour les moyennes et fortes charges. Il est à noter que d'après les traces d'exécution, la quasi-totalité des nœuds demandeurs sont satisfaits. En effet, les valeurs inférieures à *1* du taux de satisfaction sont dues principalement aux demandes tardives ; c'est-à-dire les demandes des nœuds qui viennent à la fin d'exécution de la simulation, ils doivent attendre les libérations de ressources. Ceci se confirme par les résultats de mesures concernant le taux de recherche du jeton et de ressources qui sont pratiquement nul. Autrement dit, la grande majorité des demandes ne font pas recourt à la recherche du jeton (resp. ressources) donc elles seront satisfaites.

Le nombre de messages redondants (nb_MsgRedondants / SC)

Cette métrique indique le nombre de messages redondants. Elle est calculée par : *(nombre de messages Request () reçus - nombre de messages Request () traités) / nombre d'entrées en SC.* L'allure des graphes de la Figure 49 présente une décroissance commune du nombre de messages redondants (avec une tendance vers une certaine stabilité (*0.75* saut)) avec l'accroissement de la charge de demandes. Le rapprochement des graphes entre eux montre une certaine indépendance vis-à-vis de la mobilité. Sachant que le taux de redondance est lié directement au taux de diffusion, les graphes de la Figure 41 sont cohérents avec les résultats obtenus dans la Figure 49. En effet, lorsque la charge de demandes est faible, les nœuds ont tendance à diffuser leurs requêtes; le nombre élevé de messages redondants est induit d'une part, par l'insuffisance d'informations de contrôle transportées (permettant d'éviter de visiter les nœuds déjà visités) et d'autres part, par le rayon d'envoi des demandes (voir Figure 42) qui est élevé pour les faibles charges. Quant aux moyennes et fortes charges, le taux de redondance s'affaiblit puisque le taux de diffusion et le rayon de d'envoi diminuent pour ces charges. En conclusion, il ressort que le taux de redondance est principalement dépendant du rayon de d'envoi.



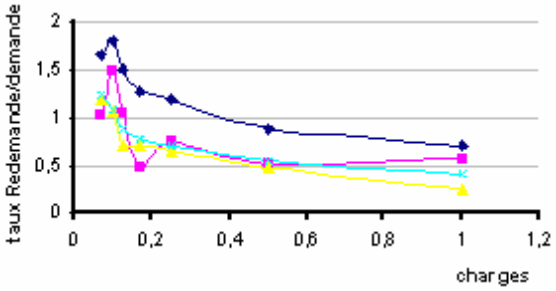


Figure 49*: Nombre de messages redondants/ SC.* **Figure 50 :** *Taux de redemandes par SC.*

Le taux de redemandes (nb_redemandes / SC)

Cette métrique indique le taux de redemandes d'entrées en SC lié au nombre total d'entrées en SC. Elle est calculée par : *nombre de redemandes / nombre de demandes* ; naturellement, nous ne considérons pas les redemandes dans le calcul du nombre de requêtes. D'après la Figure 49, nous remarquons un rapprochement des allures des graphes correspondant aux différentes mobilités. Cependant, une tendance vers la valeur *0.5* à partir des moyennes charges de demandes est enregistrée et cela pour toutes les mobilités. Ce taux montre que la majorité des demandes d'entrées en SC sont satisfaites après *0.5* redemande. Cela revient essentiellement au temps que met un nœud détenteur du jeton en attente des messages *Release ()* pour satisfaire sa demande. En effet, et d'après les traces, certains noeuds demandeurs gardent le jeton bloqué chez eux, alors que le temps d'attente des autres nœuds demandeurs s'écoule et par conséquent ces derniers redemandent l'entrée en SC. Les résultats de la Figure 50 confirment aussi que la quasi-totalité des demandes sont satisfaites (Figure 48) mais peut être après des redemandes.

7.3. Résultats se rapportant au partitionnement

La principale mesure effectuée concerne le nombre de SC exécutées après fusionnement mais à l'aide du jeton qui va être supprimé et le nombre de messages *Release ()* qui circulent dans la partition contenant le jeton à supprimer après fusionnement. Après chaque fusionnement de deux ou plusieurs partitions, un seul jeton reste valide et les autres sont supprimés dans des délais assez ''brefs''. De plus, *k* ressources au plus circulent dans le réseau, les autres ressources seront supprimées dès la réception des messages *DeleteToken ()*. Parmi les influences possibles, nous pouvons citer: Le nombre de nœuds dans la partition concernée, la vitesse de propagation du message *DeleteToken (),* l'endroit où se trouve le jeton et les ressources à supprimer lors de la fusion.

La Table 6 montre le nombre de SC exécutées après fusion (à l'aide du jeton qui va être supprimé). Cette table montre que le nombre de SC exécutées après fusionnement est nul dans la majorité des cas et prend la valeur *1* pour maximum. De plus, le nombre de messages *Release ()* circulant dans la partition qui contient le jeton à supprimer est assez faible. Donc, il s'ensuit que ces nombres ne dépendent pas réellement du nombre de nœuds dans la partition mais de l'endroit du jeton et des ressources à supprimer et aussi de la vitesse de propagation du message *DeleteToken ()* par rapport au temps d'exécution d'une SC.

Afin de remédier au problème de duplication temporaire de jeton (i.e de non sûreté), une solution simple peut être adoptée: Avant la fusion (admettons deux partitions), un jeton existe dans chacune des deux partitions concernées. Quand un lien se forme avec conséquence la fusion de ces deux partitions, on peut faire si comme si ''la fusion n'est pas encore réalisée'' jusqu'à la destruction effective du jeton et la sortie du dernier nœud de sa SC dans la partition

ciblée. Il s'agit d'interdire la communication entre ces deux partitions, en évitant de véhiculer les messages à travers ce lien pendant une durée limitée par un temporisateur.

D'autres tests ont montré que chaque partition ne possédant pas de jeton, aboutit à la création de ce dernier après consensus. Cette création de jeton survient après des délais limités (directement proportionnels au seuil de propagation). Aussi, chaque partition possédant le jeton et qui soupçonne un manque de ressources, aboutit à la régénération de ces derniers après un temps fini (délimité par *ResourceTimer* et *RecoverTimer*).

Nombre de nœuds dans la partition qui contient le jeton à supprimer.	4	7	11	15	19
Nombre de SC exécutées au moment du fusionnement (la fusion a eu lieu au cours de ces SC)	0	0	1	0	1
Nombre de SC exécutées après la fusion (avant que le message *DeleteToken ()* n'atteigne le porteur du jeton)	0	1	0	0	0
Nombre de messages *Release ()* reçus avant la réception de *DeleteToken ()*	0	0	0	1	2

Table 6: *Mesures liées à la fusion de partitions.*

8. Conclusion

Dans ce chapitre, nous avons présenté une solution au problème de *h-k-* exclusion mutuelle dans les réseaux mobiles ad hoc [16, 14, 15]. Le protocole présenté se base sur les liens physiques, n'utilise aucune structure logique. Les requêtes sont acheminées suivant un rayon dynamique d'envoi déterminé selon la connaissance locale afin de couvrir une proportion de nœuds demandeurs. Le jeton est géré selon une méthode qui combine la circulation et la demande. De plus, la politique de satisfaction des nœuds favorise en même temps les nœuds les plus proches, les requêtes les plus anciennes et les demandeurs de moins de ressources (ce qui évite la sous utilisation de ressources). La mobilité des nœuds fait partie intégrante de la solution proposée grâce aux différents mécanismes présentés dans ce qui précédait. Aussi, le protocole prend en charge les pannes de nœuds, le partitionnement, le fusionnement et par conséquent, tous les problèmes qui en résultent (perte de jeton, perte de ressources, duplications de jeton et de ressources).

A partir des différents tests ainsi que de l'ensemble des métriques effectuées, nous pouvons constater que le protocole donne des résultats satisfaisants dans sa globalité. Ces résultats sont en accord avec ceux obtenus dans le protocole d'exclusion mutuelle cadre [20]. Les complexités des différentes métriques sont à peu près semblables, mais avec des valeurs ajoutées induites par les messages liés à la gestion de ressources. Le nombre de messages nécessaires à une entrée en SC est assez faible surtout pour les moyennes et fortes charges. Le protocole se montre bien adapté à la mobilité, car cette dernière n'influe pas beaucoup sur ses performances globales. Le nombre de messages émis par SC est assez faible surtout pour les fortes charges car le rayon tend vers la valeur *1,2*. Le délai de synchronisation dépend de la mobilité mais se stabilise au fur et à mesure que la charge du réseau augmente en convergeant vers *2.5*. Le taux de satisfaction tend vers la valeur *1* pour les moyennes et fortes charges, les traces d'exécution montrent que les requêtes non satisfaites sont celles qui viennent juste avant la fin de la simulation; ils doivent attendre la libérations de ressources. Aussi, le nombre de messages redondants chute avec l'augmentation de la charge. Le taux de redemandes dans sa globalité est relativement faible lorsque la charge augmente puisqu'une demande d'entrée en SC est satisfaite après *0,5* redemande.

Chapitre VIII

Un protocole de *h* parmi *k* exclusion mutuelle basé structures logiques pour les réseaux mobiles ad hoc

1. Introduction

Dans le chapitre 7, nous avons proposé un nouveau protocole distribué orienté jeton pour le problème de la *h* parmi *k* EM dans les réseaux mobiles ad hoc qui ne fait pas recours à l'utilisation de structure logique de communication. Dans le présent chapitre, nous proposons un autre protocole de *h* parmi *k* exclusion mutuelle orienté jeton qui permet de tenir compte d'un certain nombre de caractéristiques fondamentales des réseaux mobiles ad hoc. Ce protocole ne fait pas recours à la couche de routage mais se base sur l'utilisation de deux structures logiques. Ces structures logiques consistent en deux DAG (Direct Acyclique Graph) à *niveaux*, la structure de DAG *Racine* et la structure de DAG *Clé* (ou *Distribution).* Ces structures permettent de s'adapter aux changements de la topologie du réseau et essayent 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 de ces deux appartient à l'une ou l'autre structure de DAG. 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 la structure de DAG 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 libérées sont envoyées au détenteur du jeton clé à travers les chemins définis par la structure de DAG. Les nœuds autres que racine ou clé se chargent de transmettre les messages dans la structure du DAG correspondant. Il est clair que cette méthode de répartition permet le partage de charge.

La mobilité des nœuds fait partie intégrante de la solution proposée grâce au mécanisme d'inversement de liens utilisé en cas d'une défaillance de liens ou de déplacement de jeton (afin de garantir la continuité et la sûreté du service), car le nœud puits doit être constamment dans le niveau le plus bas. De plus, les chemins vers la racine doivent rester décroissants (du

plus haut niveau jusqu'à la racine) et chaque nœud dans le réseau doit garder un chemin vers chaque sommet de l'une ou l'autre des deux structures de DAG.

Dans la solution de J-R. Jiang [55], le jeton utilisé circule constamment tant qu'il y a des ressources disponibles et les requêtes suivent ses traces à travers la structure de DAG. A chaque réception du jeton, même temporaire, son nouveau détenteur doit informer tous ses voisins de sa nouvelle élévation, dont la conséquence est éventuellement l'inversion de liens pour assurer le maintient de la structure et les réacheminements de demandes et de libérations de ressources. En outre, le seul nœud détenteur de jeton, (notamment quand il n'a pas suffisamment de ressources pour satisfaire sa demande pendante), peut être la cible de toutes les requêtes. Les conséquences immédiates sont la concentration de charge au niveau d'un même nœud et une charge supplémentaire en terme de messages et d'énergie pour les unités mobiles. En comparaison avec la solution de J-R. Jiang [55], la gestion des deux structures de DAG que notre protocole entretient est similaire à celle du DAG, mais elle s'articule autour de la notion de niveau au lieu de celle de l'élévation utilisée dans la solution de Jiang. L'apport de notre solution consiste en l'utilisation des deux jetons qui permet d'une part, l'équilibre de charges à travers le partage de services entre les détenteurs de ces deux jetons et d'autres part, la minimisation du nombre de messages liés à l'inversion de liens à travers la minimisation des déplacements de jetons tout en favorisant la concurrence d'accès aux ressources.

Ce chapitre est organisé comme suit : Après une description sommaire de notre protocole dans la Section 1, la Section 2 présente le fonctionnement du protocole à travers les mécanismes et outils utilisés. La Section 3 mis en évidence certains aspects cachés du protocole. La Section 4 présente les différents résultats de simulation obtenus ainsi que leurs interprétations et finie par une comparaison avec la solution du Chapitre 7. Finalement, la Section 5 conclue le chapitre.

2. Fonctionnement du protocole

2.1. Modèle du réseau

Nous supposons un réseau composé de *n* nœuds mobiles. Chaque nœud possède un identificateur unique. Les mouvements des nœuds du réseau sont libres (i.e. mobilité aléatoire). Les nœuds utilisent un support de communication sans fil, les liens de communication sont bidirectionnels, FIFO et fiables. Les mouvements des nœuds peuvent engendrer des créations et/ ou des pertes de liens, mais sans engendrer de partitionnement. La couche de liaison permet à tout moment aux nœuds de connaître leurs voisins. Par ailleurs, nous supposons que la durée d'une SC est bornée. Dans la suite de ce chapitre et afin de décrire notre protocole, nous nous plaçons au niveau du nœud désigné par l'indice *i*.

2.2. Les différents mécanismes et outils

2.2.1. Structure de DAG à niveaux

Pour son fonctionnement, le protocole fait appel à la structure de DAG à *niveaux* pour organiser les noeuds. Cette organisation est bâtie systématiquement sur les liens physiques du réseau sous-jacent. Chaque niveau est désigné par une valeur entière qui peut être négative. Le niveau le plus bas, initialement le niveau *0* contient un seul noeud, le nœud *puits*. Le niveau suivant *1* comporte un ensemble de nœuds qui ont chacun un lien physique avec le nœud racine. Le niveau *2* est composé des nœuds qui ont au moins un voisin dans le niveau *1*

et aucun voisin dans le niveau *0*. De manière générale, le niveau *i (>=2)* contient les nœuds qui ont au moins un voisin au niveau *(i-1)* et aucun voisin dans les niveaux inférieurs à *(i-1)*.

Tous les niveaux de la structure logique sont initialement adjacents en terme de valeur. Au cours du fonctionnement du protocole, certains niveaux peuvent devenir non adjacents mais, restent dans l'ordre décroissant du plus grand niveau jusqu'au plus petit.

Afin de définir formellement les niveaux associés aux structures de DAG, 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 voisins entre eux, $L= \{(x, y) / x \leftrightarrow y\}$. La structure de DAG est définie par l'ensemble des $(r+1)$ niveaux définis comme suit :

$Lev_0 = \{x_0\}$, ou x_0 est le nœud puits,
$Lev_1 = \{x / (x, x_0) \in L\}$
$Lev_2 = \{x / \exists y \in Niv_1, (x, y) \in L \ \& \ (x, x_0) \notin L\}$
-
$Lev_r = \{x / \exists y \in Lev_{r-1}, (x,y) \in L \ \& \ \forall m<r-1, \neg \exists z \in Lev_m / (x,z) \in L\}$; r et m sont des entiers.

Dans la structure de DAG de la Figure 51, $Lev_0 = \{0\}$, $Lev_1 = \{1, 2, 3\}$ et $Lev_2 = \{4, 5, 6\}$.

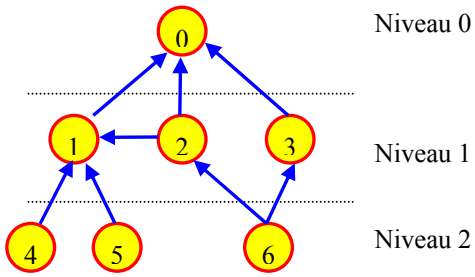


Figure 51: *Structure logique de DAG par niveaux.*

Dans le protocole, il est nécessaire d'avoir au moins un chemin logique de chaque nœud vers le nœud puits et inversement. De ce fait, chaque nœud *i* entretient trois ensembles:

- l'ensemble des *nœuds sortants* comprenant les nœuds voisins de niveau immédiatement inférieur. Du point de vue de *i*, chaque élément de cet ensemble peu servir comme prochain sur le chemin du nœud puits pour véhiculer sa requête. Par exemple, dans la Figure 51, pour le nœud *6,* cet ensemble est *{2, 3}*.
- l'ensemble des *nœuds entrants* comprenant les nœuds voisins de niveau immédiatement supérieur. Le nœud *i* est susceptible de recevoir une requête uniquement des éléments de cet ensemble. Par exemple, dans la Figure 51, pour le nœud *1*, cet ensemble est {4, 5}.
- l'ensemble des *nœuds de même niveau* comprenant les nœuds voisins de même niveau. Cet ensemble n'intervient que dans certaines situation liées au maintient de la structure.

2.2.2. Types de jetons et de DAG

Dans le but d'équilibrer la charge du service de *h-k* EM, on introduit deux types de jetons, le jeton *clé* (ou *distributeur*) et le jeton *racine* (ou *récolteur*). Le nœud détenteur du jeton clé (ou *nœud distributeur*) se charge de la distribution de ressources et de la récupération de ressources libérées. Le nœud détenteur du jeton racine (*nœud racine*) se charge de récolter les requêtes. Afin d'assurer au moins un chemin depuis chaque nœud du réseau vers chacun des nœuds détenteurs de ces jetons, on introduit deux structures de DAG, le *DAG clé* (ou *distribution*) et le *DAG racine*. Le jeton clé circule sur la structure de DAG clé et le jeton racine sur la structure de DAG racine. Le nœud détenteur de jeton doit constamment être le

nœud puits de la structure de DAG correspondante et donc accessible par tous les autres nœuds du réseau.

Dans chaque structure de DAG, à travers les ensembles définis dans la Section 2.2.1, chaque nœud hormis le nœud puits maintient un ou plusieurs lien(s) logiques sortant(s) correspondant(s) au lien(s) physique(s), dirigé(s) vers le(s) nœud(s) du niveau inférieur. Aussi, chaque nœud entretient des liens entrants pour chacune des deux structures de DAG. Ces liens permettent, selon la structure, de véhiculer à destination les messages liés au service d'allocation de ressources. Les messages de demandes de ressources suivent les liens dirigés vers le nœud récolteur de la structure de DAG racine. Les messages de ressources allouées suivent les chemins inverses mais sur la structure de DAG distribution et les messages de libérations de ressources suivent les liens dirigés vers le nœud distributeur de la structure de DAG distribution.

En outre, pour un nœud *i*, l'ensemble des nœuds voisins du même niveau reste symbolique tant que le réseau est dans un état stationnaire et ils ne sont pas considérés dans les différentes étapes du fonctionnement du protocole. Néanmoins, ils peuvent être considérés lors de la réorganisation de la structure de DAG pour devenir des liens entrants ou sortants. De ce fait, ces liens sont qualifiés de bidirectionnels.

2.2.3. La file d'attente et chemins de requêtes

Afin de satisfaire les requêtes, celles-ci doivent parvenir comme première étape au niveau du nœud récolteur (qui va par la suite les communiquer au nœud distributeur, mécanisme qui sera défini plus loin). Pour cela, les requêtes qui arrivent au niveau de chaque nœud sont celles des nœuds voisins appartenant à un niveau supérieur; elles doivent être acheminées jusqu'à ce qu'elles arrivent au nœud récolteur. Pour véhiculer les ressources aux propriétaires de ces requêtes, les chemins du retour vers ceux-là doivent être sauvegardés. A cet effet, chaque nœud *i* entretient une file d'attente Q_i, dans laquelle il sauvegarde les requêtes dans l'ordre de leurs arrivées. La file Q_i est constituée d'un ensemble d'éléments (N_j , r_j) où N_j est l'identité du nœud propriétaire d'une requête et r_j le nombre de ressources correspondantes [110]. Les éléments (N_j, r_j) sont ordonnés suivant une politique *fifo*. Cette politique garantie qu'aucune requête n'est satisfaite avant que toutes les requêtes qui la précèdent ne le soient.

Dans la Figure 52, qui reprend les mêmes structures que ceux de la Figure 51, le nœud *6* véhicule une requête sur deux ressources au nœud *0* via le nœud *3*. Celle-ci est enregistrée chez les nœuds *6* et *3* au nom du nœud *6* et au niveau du nœud *0* au nom du nœud *3*, le nœud qui l'a transmis au nœud *0*. Ensuite, le nœud *4* envoie une requête sur trois ressources au nœud *0* via le nœud *1*, puis le nœud *5* envoie une requête sur *4* ressources toujours au nœud *0* via le nœud *1*. Les files d'attentes respectives aux niveaux des nœuds concernés sont mises à jour comme décrit sur la Figure 52.

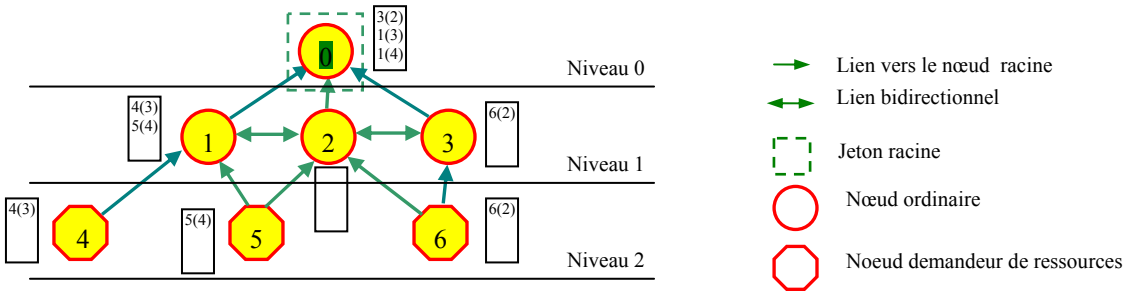


Figure 52: *Files d'attentes et chemins de requêtes.*

De cette manière, il est aisé de faire remonter les ressources vers le nœud demandeur ayant droit. Par exemple, dans la Figure 52, du point de vue du nœud *0*, le premier destinataire de *2* ressources est le nœud *3*. Arrivant à ce nœud, il constate que le destinataire (le nœud en tête de sa file) de ces ressources est le nœud *6*. A la réception de ressources, le nœud 6 s'identifie comme destinataire de celles-ci.

Chaque nœud dans le réseau maintient des chemins multiples vers chacun des nœuds récolteur et distributeur. Pour véhiculer une requête sur un chemin qui mène vers le nœud récolteur, il est nécessaire de choisir à chaque pas franchi par celle-ci le prochain nœud. Le choix du prochain nœud sortant se fait à partir de *l'ensemble des nœuds sortants* entretenu et est par défaut *arbitraire*. Ce noeud choisi est considéré comme prochain nœud sur le chemin du jeton. Son choix peut être fait selon un critère donné, par exemple son nombre de voisins; cela suppose des messages supplémentaires d'informations. Dans la Figure 52, la requête du nœud *6* par exemple, possède deux chemins possibles pour atteindre le nœud récolteur *0* qui sont soit *6*, *3*, *0* soit *6*, *2*, *0*.

2.2.4. Politique de circulation des jetons

Rappelons que la récolte de requêtes et la distribution de ressources se font à deux endroits qui sont en général distincts. La question qui se pose est comment communiquer les requêtes récoltées par le nœud récolteur au nœud distributeur de ressources?

Initialement, les deux structures de DAG sont confondues; ce qui signifie que le nœud puits qui détient le jeton racine détient aussi le jeton clé. De manière générale et au cours du fonctionnement du protocole, le nœud puits garde les deux jetons jusqu'à ce qu'il sorte de sa SC (sauf dans le cas initial où ce nœud puits n'est pas en SC auquel cas, il envoie les deux jetons dès la réception d'une première requête). Lorsqu'il sort de sa SC, il réagit selon les cas suivants:

- Si sa file est vide, il garde les deux jetons à l'état de repos;
- Si sa file contient une seule demande, alors il envoie au nœud correspondant à celle-ci les deux jetons;
- 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, la structure de DAG enracinée du réseau change et les demandes futures de ressources seront alors envoyées au nouveau nœud racine à travers cette nouvelle structure de DAG. Par contre, le jeton clé est gardé localement afin de satisfaire les demandes restantes, ce qui maintient la structure de DAG distribution inchangée. Dès qu'il reste uniquement la demande du dernier nœud de sa file, celle-ci n'est pas satisfaite mais le jeton clé est envoyé à son propriétaire. Ce qui va lui permettre d'exécuter sa SC. 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 éventuellement déjà récoltées (, donc 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 de suffisamment 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, considérons un nœud

qui 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 au moins k_1+k_2 ressources sont disponibles, le nœud distributeur lui envoie k_1+k_2 ressources à la fois.

La Figure 53, qui reprend l'état de la Figure 52, montre les deux structures de DAG qui sont confondues et dont le nœud puits est *0*. Celui-ci détient les deux jetons.

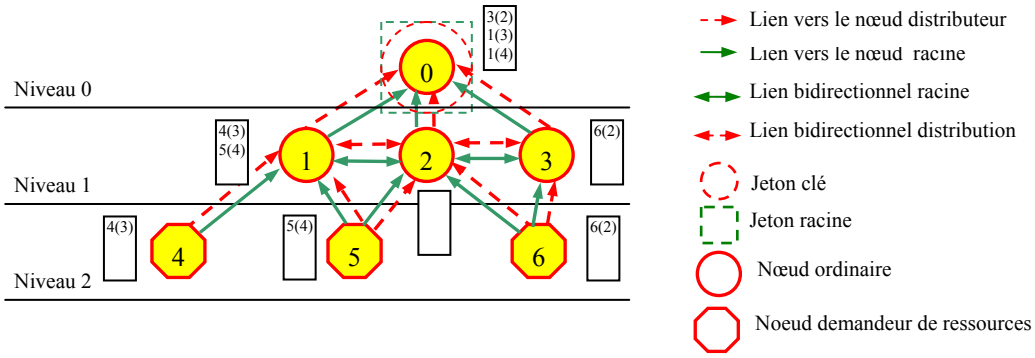


Figure 53: *Exemple de structures de DAG confondues.*

2.2.5. Inversement de liens

La structure des deux DAG change lors des déplacements de jetons et de changements de liens physiques entre noeuds, étant donné que les nouveaux détenteurs de jetons doivent constamment rester comme noeuds puits. Pour préserver alors la structure des deux DAG, on fait recours à la technique d'*inversement partielle de liens (Partial Reversal technique)* qui est déjà utilisée dans [150]. On distingue quatre cas d'inversions de liens liés au mouvement des jetons.

- *Lorsque le nœud puits envoie les deux jetons*, il doit inverser ses liens (correspondants aux deux structures de DAG) vers le nœud récepteur des deux jetons. Comme le nœud récepteur doit avoir un niveau moins élevé que celui de l'émetteur, il diminue alors son niveau relativement à celui de l'émetteur et inverse aussi ses liens en informant certains de ses voisins pour leur permettre de s'adapter à cette nouvelle situation. Chaque nœud intermédiaire recevant ces deux jetons fait de même jusqu'au nouveau destinataire des deux jetons.

La figure 54(a) montre la topologie logique initiale dans laquelle le nœud *0* possède les deux jetons à l'état de repos. Quand il reçoit la demande de *6 (il s'agit de la première qui parvient chez le nœud 0)*, le nœud *0* lui envoie les deux jetons et inverse ses liens avec le nœud 3, le nœud par lequel ces deux jetons passent. En recevant ces jetons, celui-ci diminue son niveau à *-1*, met à jour ses liens logiques, informe ses voisins de son nouveau niveau et passe les deux jetons au nœud *6*. Lorsque le nœud *6* reçoit les deux jetons, il diminue son niveau à *-2* et fait de même que son prédécesseur. La Figure 54(b) montre l'état des deux structures après réorganisation dont le nœud *6* devient le nœud racine.

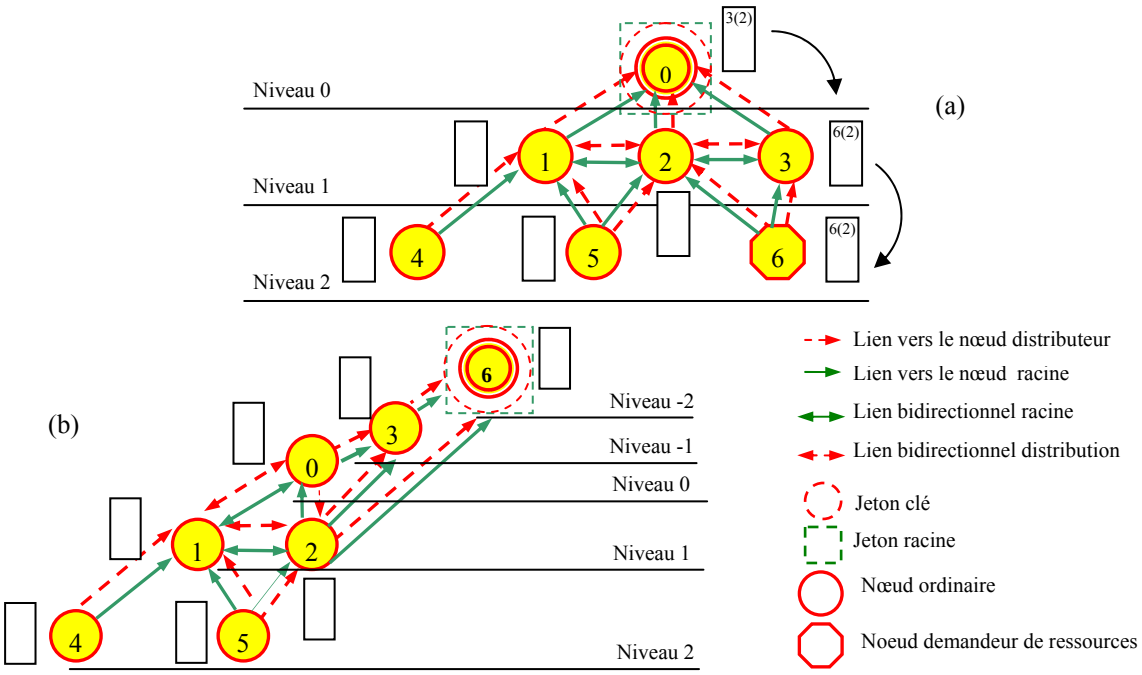


Figure 54 : *Circulation des deux jetons et réorganisation des DAG.*

- *Lorsque le nœud envoie le jeton racine* à une nouvelle destination, la structure de DAG racine est réorganisée afin que le nouvel nœud racine soit au niveau le plus bas et tous les liens deviennent orienter vers lui. Ce qui va permettre l'acheminement de prochaines requêtes vers le nouveau nœud racine. Le mécanisme d'inversion de liens est similaire au premier.

Considérons le système de la Figure 53 et supposons que le nœud *0* est en SC. Lorsque ce nœud sort de sa SC, Il trouve que sa file contient *3* requêtes en attente. Tout d'abord, il envoie le jeton racine vers le nœud en queue de sa file i.e., le nœud *1*, inverse son lien de la structure racine vers ce nœud, informe ses autres voisins et commence à satisfaire les deux premières requêtes de sa file d'attente en leur envoyant des mesages de ressources. Lorsque le nœud *1* reçoit ce jeton, il trouve qu'il n'est pas destinataire, il l'envoie donc au nœud en queue de sa file i.e. le nœud *5*. Le nœud *1* exécute la même procédure d'inversion de liens que celle exécutée par le nœud émetteur. Le nœud *5*, destinataire du jeton racine, lorsqu'il le reçoit, met à jour son lien pour devenir le nouveau nœud récolteur, informe ses voisins et commence à attendre les requêtes qu'il va satisfaire (en commençant par la sienne) lorsque le jeton distributeur lui parviendra. Remarquons que ce nœud est celui dont la dernière requête est parvenue chez le nœud *0*; ce qui signifie que le protocole sert les requêtes dans leur *ordre d'arrivée*, ce qui évite la famine. La Figure 55 donne le résultat de réorganisation de la structure de DAG racine. Dans cette Figure, pour des raisons de clarté du schéma, nous avons omis les arcs de la structure de DAG clé qui reste inchangée. Rappelons que le nœud *0* est toujours nœud distributeur. Si le nombre de ressources dont il dispose est suffisant, le nœud *0*, parallèlement aux actions entreprises par les autres nœuds liées au mouvement du jeton racine, commence à satisfaire les requêtes dans sa file (voir Figure 53) en suivant leurs chemins inverses. Il envoie alors *2* ressources au nœud *3* (le premier nœud demandeur du point de vue du nœud *0*). Celui-ci trouve que le nœud demandeur, de son point de vue, est le nœud *6*, il lui passe ces deux ressources. Le nœud destinataire de ressources est alors atteint. Le même processus est appliqué pour la prochaine requête.

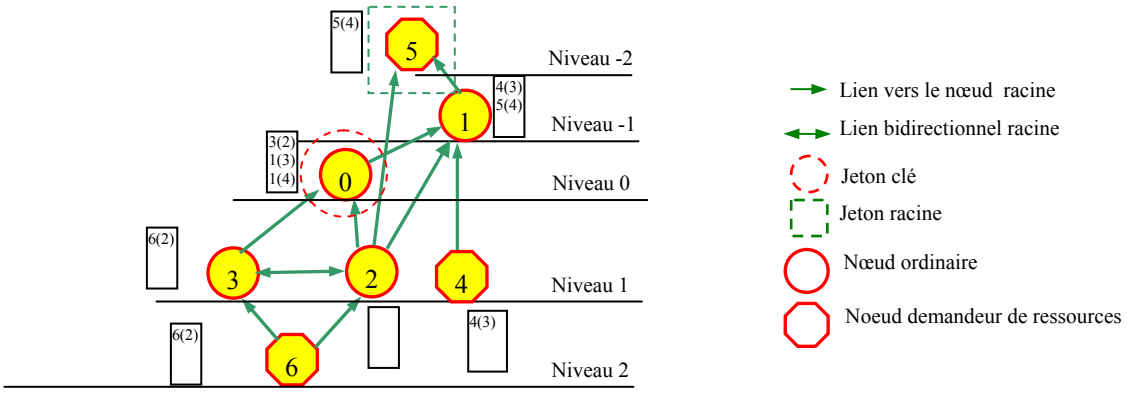


Figure 55 : *Circulation du jeton racine et réorganisation du DAG racine.*

- *Lorsque le nœud distributeur envoie le jeton clé* vers le nœud récolteur qui lui est différent. Dans ce cas, seule la structure de DAG distributeur qui subit une réorganisation. En fait, elle devient confondue à la structure de DAG racine. La procédure est plus simple que les deux précédentes, il suffit de copier les informations liées au DAG racine.

- *Lorsqu'un nœud i détecte une perte de lien physique* avec un voisin *j*, il procède à un traitement lié aux mises à jour des structures de DAG. Il commence par supprimer son identité de l'ensemble correspondant. Un autre traitement est effectué dans le cas où les niveaux associés à *i* et *j* sont différents comme suit:

  - Cas 1: Le nœud *j* est de niveau supérieur: Si une requête de celui-ci est présente localement, celle-ci est supprimé. Cela suppose que le nœud *j* re-route sa requête,
  - Cas 2: Le nœud *j* est de niveau inférieur: Si le lien avec *j* n'est pas son dernier lien sortant, le nœud *i* change éventuellement le chemin vers la racine et re-route ses requêtes. Dans le cas contraire, le nœud *i* est amené à assurer au moins un lien sortant pour ses requêtes. Il inverse alors, ses liens logiques avec ses voisins entrants (i.e., de niveau inférieur) de plus petit niveau en augmentant son niveau en conséquence. Il re-route ensuite ses éventuelles requêtes. Il est à constater que par conséquence, d'autres nœuds peuvent être impliqués par un changement partiel de leurs liens logiques.

La Figure 56 illustre la procédure d'inversion de liens selon les cas cités ci-dessus. Cette Figure reprend l'état de la Figure 53 sans la requête du nœud *5*. Supposons que le nœud *1* se déplace et rompe son premier lien avec le nœud *2* puis avec le nœud *0*. La première rupture implique une suppression mutuelle du correspondant de l'ensemble des nœuds du même niveau. La deuxième rupture illustrant les deux cas cités ci-dessus, fait perdre le dernier lien sortant du nœud *1* vers le nœud puits. Il choisi alors les nœuds entrant de niveau le plus petit, i.e. dans ce cas, seul le nœud *5* est viable ; il inverse son lien avec celui-ci. Le niveau du nœud *1* devient *3*, plus grand que celui du nœud *5* et celui du noeud *4* devient *4* car le nœud *1* de son point de vue est sortant, donc de niveau inférieur. Remarquons, étant donné que les deux structures de DAG sont confondues, la même procédure d'inversion de liens est appliquée au DAG clé. Notons que le nœud *0* a supprimé de sa file la requête au nom du nœud *1* (, requête du nœud *4*). Le nœud *1* re-route ainsi sa requête via le nouveau chemin tracé qui passe successivement par le nœud *5*, le nœud *2* pour atteindre le nœud *0*, i.e. le nœud destinataire. Les files de ces nœuds indiquent la nouvelle route distribuée, ce qui assure un chemin de retour au nœud *4*, propriétaire de cette requête.

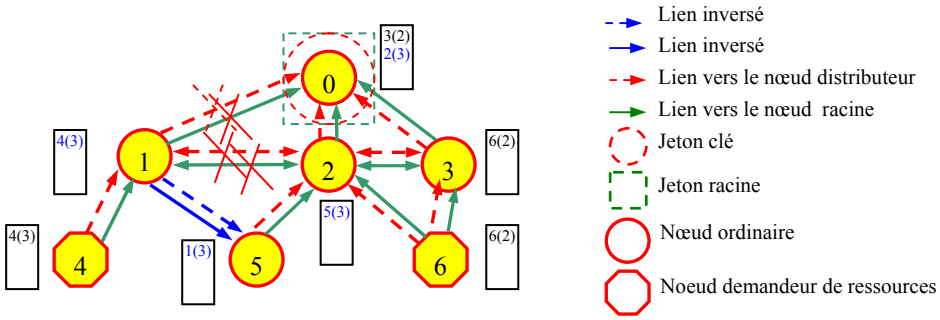


Figure 56: *Inversement partielle de liens suite à des ruptures de liens.*

- *La création d'un nouveau lien* implique seulement la mise à jour de l'un des trois ensembles liés aux liens avec le voisinage.

  La Figure 57 reprend l'état de la Figure 56 et montre la création d'un nouveau lien entre le nœud *5* et le nœud *0*. Chacun des deux nœuds ajoute l'autre dans l'ensemble approprié.

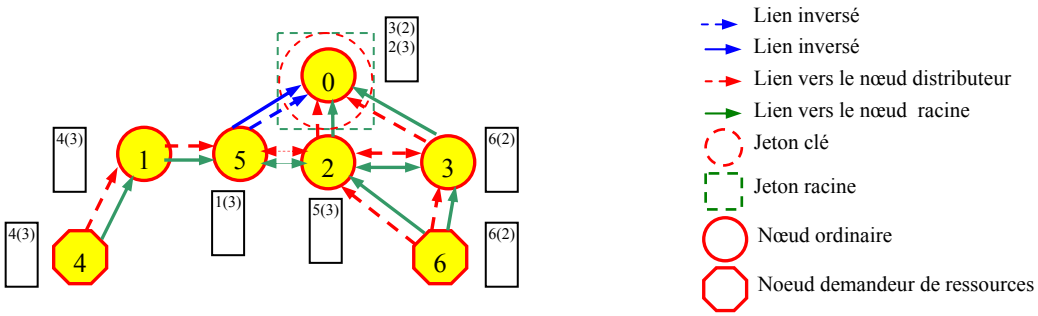


Figure 57 : *Mise à jour des DAG suite à une création de lien physique*.

3. Discussion

Pourquoi choisir deux jetons dans la solution ?

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

- Il permet d'éviter certains problèmes tels que la centralisation donc favorise l'équilibre de charges, i.e., la collecte des requêtes et la distribution des ressources peut se faire à deux endroits différents.
- En prévision de la tolérance aux pannes, en cas de perte d'un des jetons (celui qui possède l'autre génère le jeton perdu).
- Il réduit la complexité de messages liés aux déplacements des jetons. En effet, dans le cas de moyennes et fortes charges, les jetons se déplacent moins fréquemment du fait que pratiquement toutes les requêtes (qui sont nombreuses, sauf la dernière) dans la file locales sont servies sans déplacement du jeton distributeur. Par conséquent, les messages de restructuration des structures logiques liés à ce déplacement sont évités.
- Il permet d'éviter de transporter les requêtes par le jeton. En effet, dans cette solution aucune requête n'ai transportée étant donné que le détenteur du jeton clé, après avoir envoyé le jeton racine à son prochain, continu à satisfaire les requêtes locales.

Les requêtes sont-elles satisfaites dans leur ordre d'arrivée?

Dans le cas normal, les requêtes sont satisfaites dans leur ordre d'arrivée. En effet, quand les deux structures de DAG sont confondues, le détenteur des jetons quand il sort de sa SC, en supposant que sa file contient plusieurs requêtes, il envoie le jeton racine vers le propriétaire de la dernière requête dans la file. Puis, il commence à satisfaire les requêtes dans le même ordre de leur d'enregistrement (i.e. d'arrivée). Les prochaines requêtes, qui arrivent, seront récoltées par le nouveau nœud racine dans l'ordre d'arrivée. Quand le nœud distributeur termine de satisfaire les requêtes locales, sauf la dernière, il envoie le jeton clé vers le nouveau nœud récolteur qui va prendre le relais. Celui-ci commence par satisfaire sa propre requête (qui est la dernière requête qui n'a pas été satisfaite par l'ancien nœud distributeur. Dans le cas rupture de lien qui fait perdre le dernier lien sortant à un noeud, étant donnée que la (ou les) requête(s) de celui-ci est (ou sont) re-routée(s), elle(s) peut (ou peuvent) perdre son (ou leurs) tour(s). Cela peut être le cas pour la requête du nœud *4* dans l'exemple de la Figure 56 si d'autres requêtes arrivent au niveau du nœud *0* avant que le changement de route ai été achevé.

Pourquoi attendre la sortie de SC d'un nœud détenteur des deux jetons avant de faire passer le jeton racine à son prochain?

Il s'agit de permettre à ce nœud de récolter un nombre important de requêtes, qu'il va lui-même satisfaire, afin de diminuer la circulation fréquente de jetons et donc de réduire la complexité en messages.

Quand un nœud est amené à re-router sa requête, quel est le sort de son ancienne requête ?

En réalité, quand un nœud *j* supprime la requête d'un autre nœud *i* suite à une rupture de lien, les autres nœuds éventuels après ce nœud sur le chemin du jeton ne sont pas, par défaut (option choisie), au courant de cette suppression. Ceci signifie que la requête reste présente au niveau de ces nœuds; elle sera donc ''satisfaite''. Quand les ressources la concernant arrivent au niveau du nœud *j*, celui-ci déduit que le propriétaire de la requête n'ai plus voisin, donc libère les ressources et supprime la requête en guise de satisfaction. Rappelons que le nœud *j* peut recevoir, en échange de ressources, le jeton racine ou les deux jetons. Dans ce cas, il garde le ou les jetons reçus localement en supprimant la requête. La file locale devient nécessairement vide, par conséquent il attend la réception de nouvelles requêtes.

Dans la Figure 58, une rupture de lien se produit entre les nœuds *5* et *1*. Le nœud *1* a supprimé la requête du nœud *5*, mais le nœud *0*. Quand les *4* ressources destinées au nœud *5* arrivent au niveau du nœud *1*, celui-ci parce que la requête en tête de sa file requièrt 3 ressources, se rend compte que le destinataire n'ai plus voisin, donc libère ces deux ressources. Pour éviter de faire déplacer inutilement des ressources, quand un nœud est amené à supprimer la requête d'un autre nœud pour des raisons de rupture de lien, il informe tous les nœuds suivant jusqu'à la racine pour faire de même. Un problème apparaît si le message de ressources et celui d'information se croisent et donc le problème reste entier.

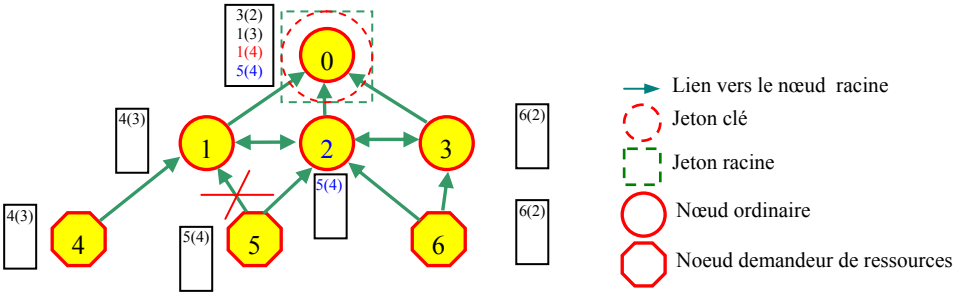


Figure 58: *Livraison de ressources avec absence de requête.*

4. Simulation

L'évaluation des performances du protocole est effectuée à l'aide de l'outil NS2 (*Network Simulator 2*) [136] version 2.30 sous le système d'exploitation LINUX Suze 10.1. La codification du protocole est effectuée à l'aide du Langage objet Otcl.

4.1 Environnement de simulation et paramètres

L'environnement dont les différentes simulations ont été effectuées, peut être décrit à travers un certain nombre de paramètres. Les paramètres constants sont: la surface de mouvement est de *1000 × 500 mètres*; le nombre de nœuds dans le réseau est de *25*; la durée de la SC est de *0.1 s*; le rayon de communication entre nœuds mobiles est de *250 m*; le temps de simulation est de *100 s*, le nombre de ressources par défaut dans le réseau est *10* et la durée de transfert d'un message est de *0.1 s*.

Les paramètres variables sont:

- La mobilité : Elle est donnée par le modèle aléatoire se basant sur le mouvement relatif des nœuds (voir Chapitre 1). A partir de la formule de la mobilité, nous avons : mobilité faible : *0* < Mob <= *3* ; mobilité moyenne : *3* < Mob <= *8* ; mobilité élevée : *8* < Mob. Pour la simulation, nous avons utilisé quatre valeurs de mobilité : *0*, *1.7*, *5*, *10.*
- La charge : Elle est définie par une loi de poisson de paramètre λ dont les valeurs prises pour la simulation 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 SC.
- La connectivité : Elle est calculée en terme de pourcentage des liens existants par rapport au nombre de liens possibles dans le graphe. Nous avons considéré les connectivités suivantes : *5%*, *25%*, *50%*, *75%* et *100%.*
- Le nombre de ressources: Ce paramètre est intrinsèque au protocole. Les valeurs utilisées sont: *10*, *20* et *30* ressources

4.2. Résultats et interprétations

Afin d'étudier les performances du protocole, les trois métriques suivantes ont été prises en compte:

- Temps moyen d'attente par entrée en SC (*Tatt/ SC*): C'est le nombre moyen d'unités de temps écoulées entre la demande d'accès en SC et l'accès lui-même. Elle est calculée par: *La somme des (Temps d'entrée en SC - Temps de la Requête) de tous les nœuds demandeurs/ Nombre d'entrées en SC*.
- Nombre moyen de messages générés par entrée en SC (*Nb_Msg/ SC*): Il est calculé par: *la somme de tous les messages émis / le nombre d'entrées en SC*.
- Délai de synchronisation (*DélaiSynch*): C'est le temps moyen écoulé en terme de nombre moyen de messages générés entre deux entrées successives en SC. Un nœud accède à sa SC dans l'un des trois cas, soit en recevant le message *Resources ()*, soit en recevant les deux jetons, soit en recevant le jeton clé. Dans le premier cas, *DélaiSynch= (Nombre de messages Resources () + Nombre de messages Release ())/ Nombre d'entrées en SC par le message Resources ()*. Dans le deuxième cas, *DélaiSynch= (Nombre de messages AllTokens () + Nombre de messages Release() + Nombre de messages ReverseLinkInfo())/ Nombre d'entrées en SC par le message AllTokens()*. Dans le troisième cas, *DélaiSynch=(Nombre de messages TokenKey() + Nombre de messages Release() + Nombre de messages ReverseLinkInfo())/Nombre d'entrées en SC par le message TokenKey()*. De manière générale *Délaisynch* est calculé par la somme de ces trois quantités.

4.2.1. Résultats se rapportants aux différentes métriques en fonction de la charge

Les mesures effectuées à ce niveau sont en fonction de la charge pour différentes valeurs de mobilités. Les différents résultats ont permis de dégager un comportement général du protocole en fonction de la charge.

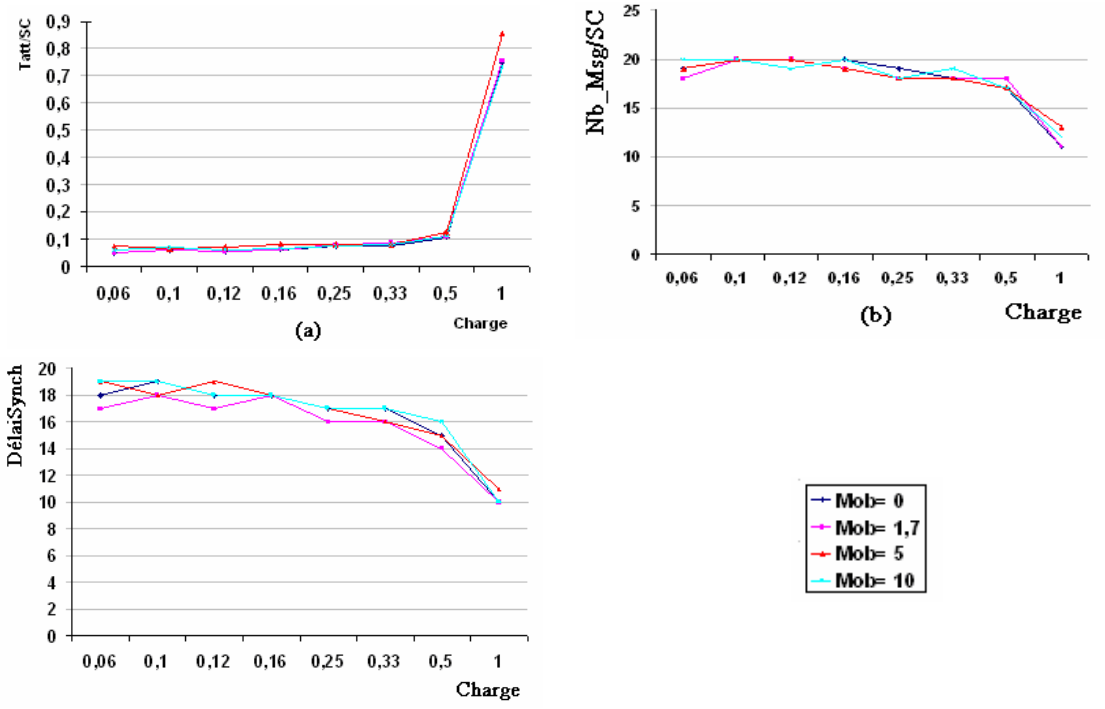


Figure 59: *(a)Temps d'attente par SC;(b) nombre moyen de messages par SC et (c) délai de synchronisation, tous en fonction de la mobilité.*

Temps d'attente moyen par entrée en SC (Tatt/ SC)

Les graphes du temps d'attente moyen par SC de la Figure 59(a), ont une allure commune monotone croissante pour les six premières valeurs de la charge (jusqu'à une demande chaque 3s). Dans le cas de fortes charges s'approchant de *1* s, les courbes montrent un pic. Il est à constater que l'écart entre les courbes correspondant aux différentes mobilités n'est pas significatif.

L'accroissement du temps d'attente avec la charge revient essentiellement aux manques progressifs de ressources disponibles avec l'augmentation du nombre de demandes. Autrement dit, pour les faibles charges 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. Pour les fortes charges, étant donné que toutes les ressources sont occupées, les nœuds doivent attendre leurs libérations; ce qui explique l'augmentation rapide du temps d'attente.

Nombre moyen de messages par entrée en SC (Nb_Msg/ SC)

Les différentes courbes de la Figure 59(b) présentent une tendance commune décroissante avec un écart très réduit. La réduction du nombre de messages est conjointement liée au temps moyen d'attente pour une entrée en SC. Ceci dit, plus la charge des demandes est importante, plus le nombre de ressources ne suffit plus et plus le temps d'attente augmente, moins les nœuds génèrent de messages de demandes de SC. Dès que les ressources se libèrent, les

processus peuvent entrer en SC. Dans ce cas, mis à part les messages de libération de ces ressources et ceux de satisfaction des demandes qui circulent, les autres types de messages se font rare. Ce qui laisse à conclure que le nombre moyen de messages par SC est "inversement proportionnel" à la charge.

Nous pouvons dire aussi, que la mobilité a une influence minime sur le trafic de messages. Elle influe principalement sur les chemins des jetons, des messages de ressources et de libération de ressources par les ruptures des routes.

Délais de synchronisation (DélaiSynch)

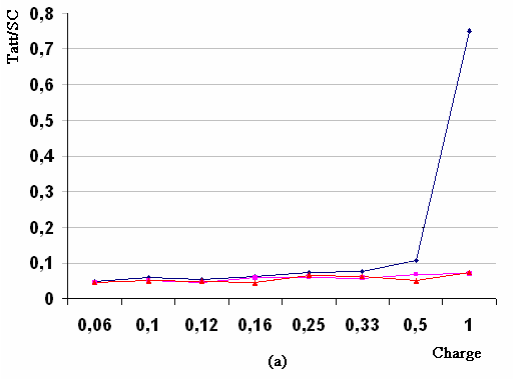
Les courbes de la Figure 59(c) ont une allure commune décroissante, ce qui signifie que plus forte est la charge, meilleures sont les performances. Ceci peut être expliqué par l'augmentation du temps d'attente à défaut de ressources. En effet, l'insuffisance de ressources engendre une stagnation du système à ce moment. Par conséquent, le trafic de messages chute 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.

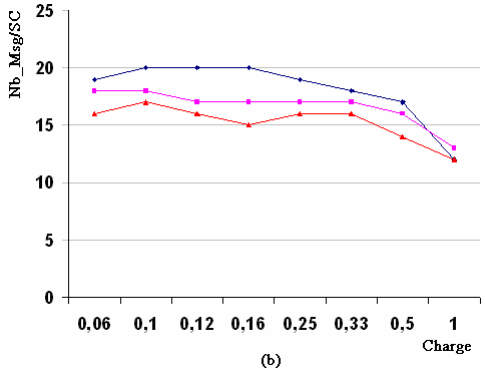
L'augmentation de la mobilité fait augmenter légèrement le délai de synchronisation. Cette augmentation s'explique par l'augmentation du nombre messages de type *ReverseLinkInfo (),* ce qui est prévisible. Néanmoins, l'influence de la mobilité reste minime.

4.2.2. Résultats se rapportant à la variation du nombre de ressources

Afin de montrer l'influence de la variation du nombre de ressources sur les performances, nous avons effectué des mesures avec différents nombres de ressources (système à 10, 20, 30 ressources). Nous tenons à préciser que, afin de montrer l'influence de la variation du nombre de ressources sur les performances du système, le nombre de ressources demandées par les processus reste réduit (*la demande maximal d'un nœud ne dépasse pas 10 unités de ressources*). Autrement, si le nombre de ressources demandées suit le nombre de ressources présentes dans le système, les résultats seront généralement semblables à ceux déjà donnés en Section 4.2.1.

Cas de réseau statique





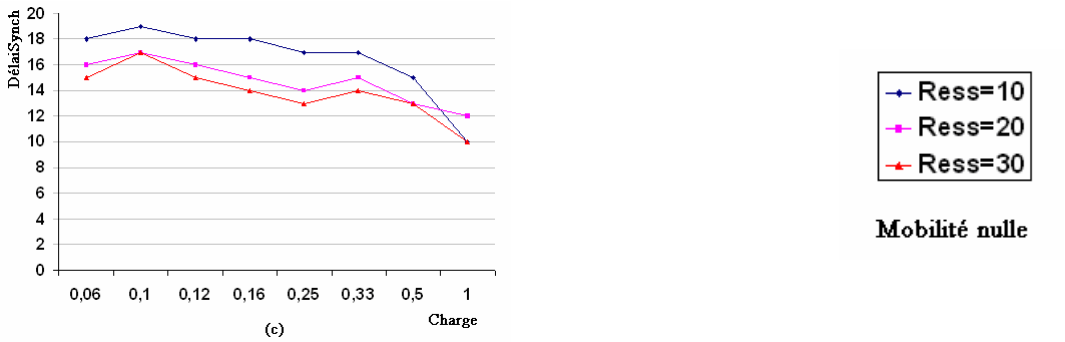


Figure 60: *(a)Temps d'attente par SC;(b) nombre moyen de messages par SC et (c) délai de synchronisation, tous en fonction du nombre de ressources.*

Temps d'attente moyen par entrée en SC (Tatt/ SC)

Les courbes de la Figure 60(a) montrent une croissance légère du temps d'attente avec l'augmentation de la charge et la diminution du nombre de ressources. En général, ces courbes sont proches les unes des autres, même dans le cas où le nombre de ressources est *10* dont les performances ne sont pas si dégradées que prévisible, excepté dans le cas de charges élevées (de *0,33* à *1*). Dans ce dernier cas, le manque de ressources est clairement ressenti par la divergence de la courbe correspondante. Autrement, dans le cas où le nombre de ressources est *20* ou *30*, les performances restent bonnes.

Le rapprochement des courbes entre elles pour les faibles et moyennes charges laisse à penser que certaines ressources (dans le cas de *20* ou *30* ressources) ne sont pas bien exploitées dans le sens où certaines ressources restent toujours disponibles. Cependant, ce soupçon n'est pas correct. En effet, en observant les courbes issues des mesures sur le nombre moyen de messages par SC de la Figure 60(b), nous remarquons que les courbes pour *20* et *30* ressources montrent un trafic de messages en diminution, ce qui veut dire que les nœuds attendent les libérations de ressources.

Nombre moyen de messages par entrée en SC (Nb_Msg/ SC)

Les courbes de la Figure 60(b) montrent une certaine constance du nombre moyen de messages par SC jusqu'aux moyennes charges. Au delà, une décroissance commune est observée. D'autre part, tout au long de la charge, on constate aussi un certain rapprochement entre les courbes pour les différents nombres de ressources. Globalement, on constate que plus le nombre de ressources augmente, moins il y a de messages générés.

Pour les faibles charges (de *0* à *0,16*), les ressources sont suffisantes pour satisfaire toutes les requêtes actuelles. Par conséquent, les files d'attentes sont pratiquement vides, Donc, lorsque le nœud racine sort de sa SC, il envoie les deux jetons (un seul message) au premier nœud lui demandant des ressources. Ceci compense les autres messages circulant. Plus la charge augmente, plus il y a des requêtes qui attendent au niveau des files et donc plus le système devient passif ; ce qui explique la diminution du nombre de messages. En outre, l'augmentation du nombre de ressources dans le système rend quelque peu le système mieux actif. Donc, le nombre de demandes dans les files d'attente diminue globalement, ce qui diminue le nombre de fois que ces requêtes soient reroutées à cause des déplacements des jetons, ce qui diminue le nombre total de messages.

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

Délais de synchronisation (DélaiSynch)

Les courbes de la Figure 60(c)) présentent une allure décroissante commune et montre clairement que les performances du protocole deviennent meilleures lorsque le nombre de ressources augmente notamment pour le cas de forte charge.

L'augmentation du nombre de ressources fait augmenter le nombre de satisfactions de requêtes, ce qui diminue le contenu des files d'attentes et par conséquent diminue le nombre de messages nécessaires aux changements de routes des requêtes. Ce qui explique l'écart entre les différentes courbes. D'autre part, l'augmentation de la charge, rend les nœuds demandeurs de ressources proches les uns des autres ce qui diminue le trajet des ressources et des jetons. Ceci explique la diminution du délai de synchronisation avec l'augmentation de la charge.

Cas de réseau dynamique

Dans cette section, il s'agit de présenter les résultats de simulation pour les trois métriques en fonction de chaque type de mobilité et en faisant varier le nombre de ressources dans le système.

Temps d'attente moyen par entrée en SC (Tatt/ SC)

Les courbes des différents graphes de la Figure 61 montrent une influence légère de la variation du nombre de ressources. Plus le système dispose de ressources moins un nœud attend pour entrer en SC, ce qui est prévisible. On constate néanmoins une augmentation brusque dans le cas de 10 ressources quand la charge s'approche de 1. On constate aussi que la mobilité n'a pas d'influence significative sur le temps d'attente étant donné que les courbes sont proches les unes des autres.

Ces constatations sont proches de celles de la Section 4.2.1, ce qui impliquent que les interprétations soient les mêmes.

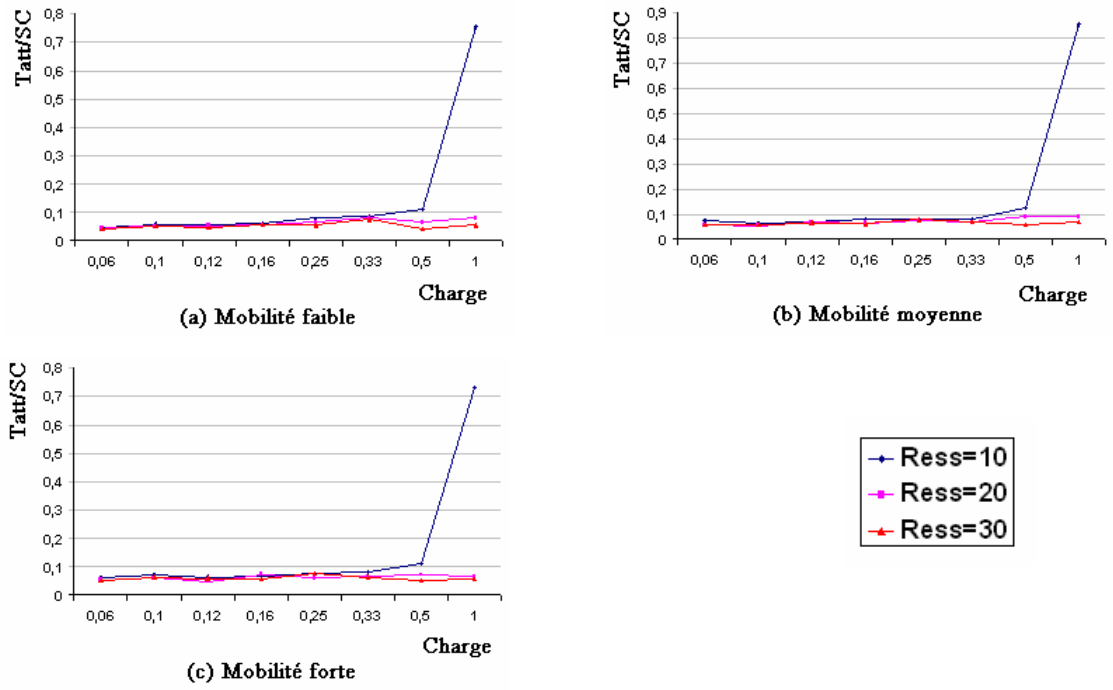


Figure 61: *Temps d'attente par SC (avec variation du nombre de ressources).*

Nombre moyen de messages par entrée en SC (Nb_Msg/ SC)

Les courbes de la Figure 62 présentent une décroissance commune tout en gardant un écart relatif apparent pour les différentes valeurs du nombre de ressources. Cependant, un chevauchement des différentes courbes pour chaque mobilité est constaté lorsque la charge s'approche de *1*. Ce qui veut dire que l'augmentation du nombre de ressources fait diminuer le temps d'attente, ce qui prévisible.

La décroissance du nombre de messages s'explique par le fait que quand la charge augmente, les files d'attentes commencent à se remplir et donc les nœuds concernés attendent les ressources, ce qui les empêche d'envoyer d'autres requêtes. Les messages qui circulent en ce moment sont ceux liés à la restructuration des structures logiques, aux déplacements de jetons, aux quelques messages de ressources et de requêtes. Le rapprochement entre les courbes s'explique aussi par la stagnation commune du système en ce moment. L'augmentation du nombre de ressources laisse le système plus actif et surtout dans le cas de *30* ressources dont le nombre de messages dépasse celui pour *20* ressources car les requêtes sont mieux servies, ce qui conduit aux nouvelles demandes. Globalement, l'augmentation du nombre de ressources fait diminuer le nombre de requêtes dans les files d'attente, ce qui diminue le nombre de fois que ces requêtes soient re-routées à cause des déplacements des jetons, ce qui diminue le nombre total de messages.

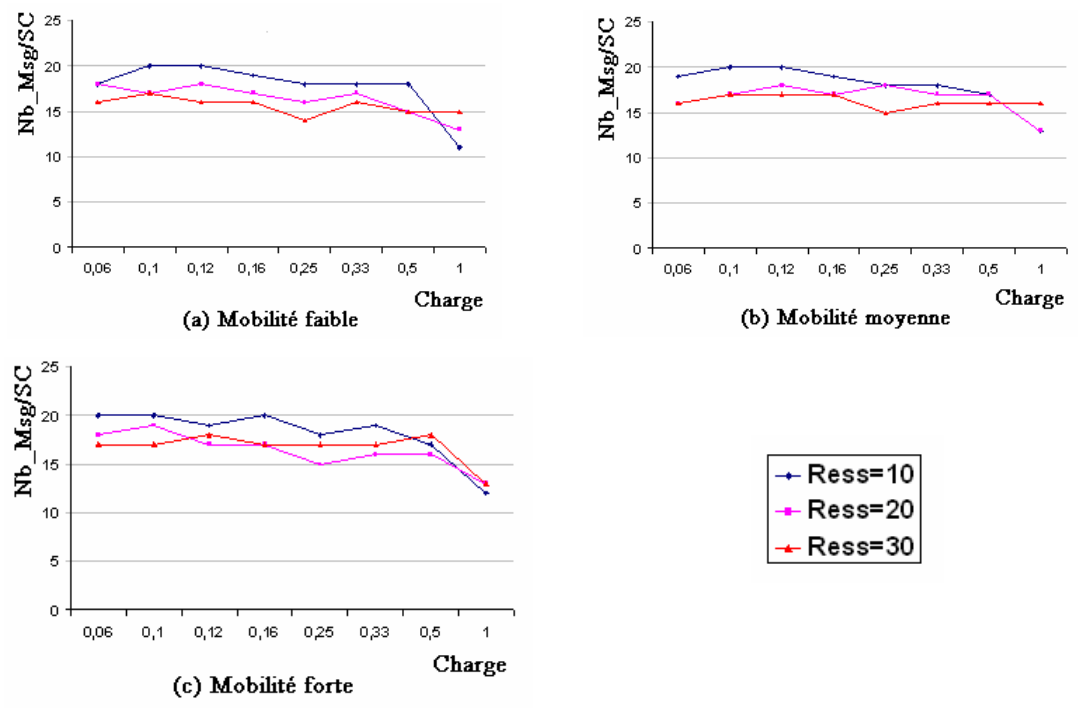


Figure 62: *Nombre moyen de messages par SC (avec variation du nombre de ressources).*

Délais de synchronisation (DélaiSynch)

Toutes les courbes de la Figure 63 ont une allure commune décroissante, en présentant un écart significatif pour les faibles et moyennes charges. Cet écart se rétrécie dès que la charge devient forte. D'autre part, l'augmentation de la mobilité fait augmenter légèrement le délai de synchronisation, ce qui laisse à conclure que la mobilité a peu d'effets sur le délai de synchronisation. D'autre part, l'augmentation du nombre de ressources fait diminuer le délai de synchronisation.

La diminution du délai de synchronisation avec l'augmentation de la charge s'explique d'une part, par la stagnation du système à cause du défaut de ressources, d'autre part, les nœuds demandeurs de ressources deviennent proches les uns des autres ce qui diminue le trajet des ressources et des jetons. L'augmentation du délai de synchronisation avec la mobilité s'explique par l'augmentation du nombre de messages de type *ReverseLinkInfo (),* ce qui est prévisible. Néanmoins, l'influence de la mobilité reste minime. Il est clair que l'augmentation du nombre de ressources fait augmenter le nombre de satisfactions de requêtes, ce qui diminue le contenu des files d'attentes et par conséquent diminue le nombre de messages nécessaires aux changements de routes des requêtes. Ce qui explique l'écart entre les différentes courbes. Le rapprochement des courbes dans le cas de fortes charges est motivé par le manque commun de ressources à ces charges.

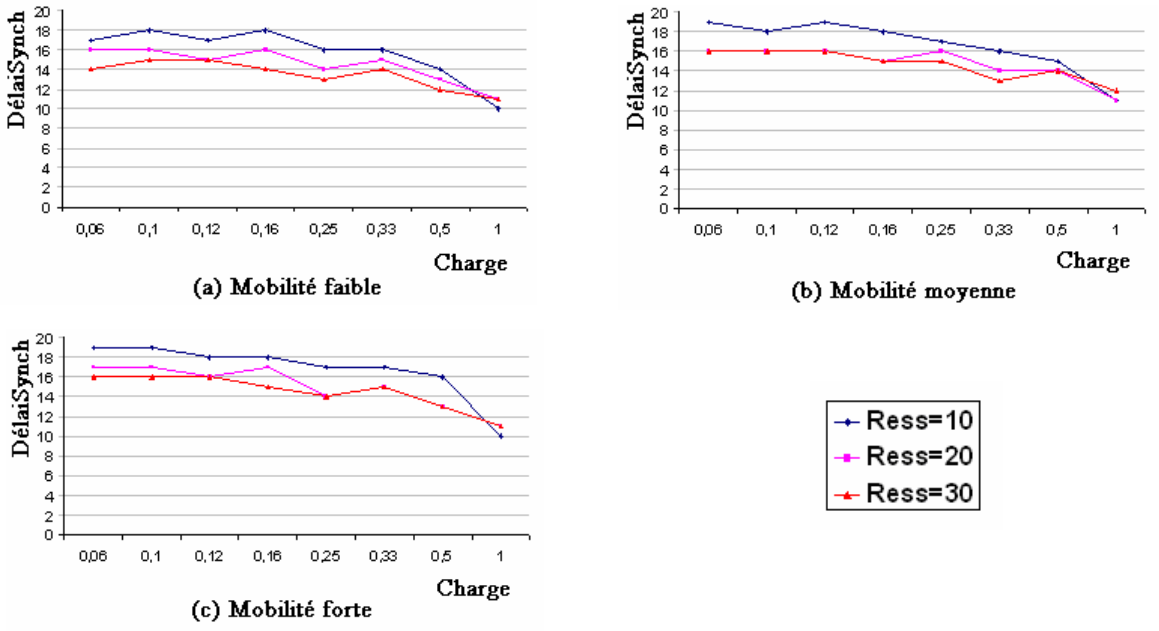
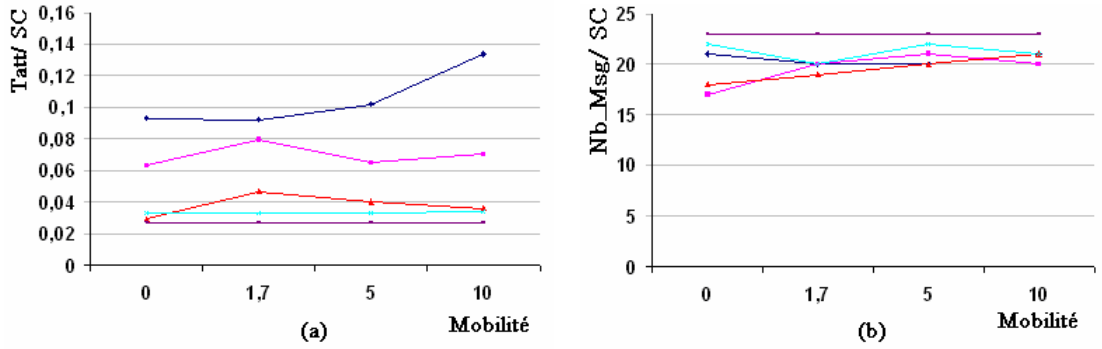


Figure 63: *Délai de synchronisation (avec variation du nombre de ressources).*

4.2.3. Résultats se rapportant à la variation de la connectivité

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



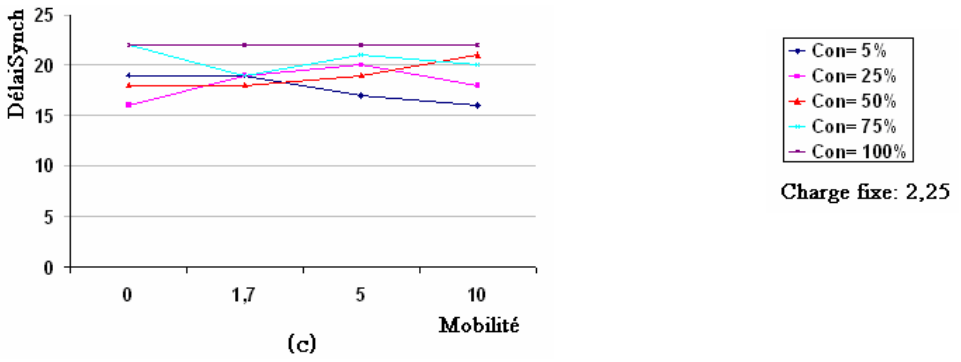


Figure 64: *(a)Temps d'attente par SC;(b) nombre moyen de messages par SC et (c) délai de synchronisation, tous en fonction de la connectivité.*

Temps d'attente moyen par entrée en SC (Tatt/ SC)

Les courbes de la Figure 64(a) montrent que l'influence de la connectivité sur le temps moyen d'attente par SC est évidente du fait de l'écart existant entre les différentes courbes. Plus précisément, 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%*). Cette constance s'explique par le fait que malgré que les nœuds se déplacent, ils restent pratiquement tous voisins.

Nombre moyen de messages par entrée en SC (Nb_Msg/ SC)

Les courbes de la Figure 64(b) montrent une légère augmentation en nombre de messages avec l'augmentation de la mobilité. Ce qui laisse à comprendre que la mobilité n'a pas eu d'influence sur cette métrique. Cependant, la majorité des courbes sont proches les unes des autres, à l'exception de celle correspondant au taux 100%. D'autre part, on constate que le nombre de messages par SC augmente légèrement en augmentant le taux de connectivité et la connectivité 50% semble donner un résultat optimal.

Lorsque le taux de connectivité augmente, les nœuds deviennent proches de la racine dans chacune des deux structures de DAG. Par conséquent, quand au moins un jeton se déplace, tous les nœuds reçoivent des messages d'inversement de liens. L'augmentation du nombre de messages avec l'accroissement de la mobilité, quoique minime, s'explique par le fait qu'il y a plus de ruptures de liens, donc plus de messages liés à la réorganisation des structures logiques.

Délais de synchronisation (DélaiSynch)

A partir des courbes de la Figure 64(c), le délai de synchronisation est légèrement inférieur au nombre de messages par SC (Figure 64(b)) pour toutes les connectivités. Ce résultat s'explique d'une part, par le calcul du nombre de messages qui inclus celui du délai de synchronisation plus d'autres messages liés principalement aux requêtes; d'autre part, étant donné que la charge est moyenne, l'écart est alors du à ce nombre de messages de requêtes.

4.3. Discussion et comparaison

Après toutes les mesures effectuées afin d'évaluer les performances du protocole, il ressort un certain nombre de constatations relatives au comportement général du protocole.

Les courbes représentant le délai d'attente par SC 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 devient forte. 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. D'autres part, l'augmentation du nombre de ressources sert mieux les nœuds, ce qui fait diminuer leur temps d'attente.

Les courbes représentant le nombre de messages en fonction de la charge ont en général une allure décroissante pour tous les cas de mobilité. Ceci montre que le protocole réagit de manière efficace dans les conditions de moyennes et fortes charges. D'autre part, l'augmentation du nombre de ressources est en faveur du protocole. Cette augmentation rend quelque peu le système mieux actif. Donc, le nombre de demandes dans les files d'attente diminue globalement, ce qui diminue le nombre de fois que ces requêtes soient reroutées à cause des déplacement des jetons, ce qui diminue le nombre total de messages.

Le délai de synchronisation diminue avec l'augmentation de la charge à cause d'une part, de la stagnation du système à défaut de ressources et d'autre part, du fait que les nœuds demandeurs de ressources deviennent proches les uns des autres ce qui diminue le trajet des ressources et des jetons. Le délai de synchronisation est légèrement sensible à l'augmentation de la mobilité à cause de l'augmentation du nombre de messages de type *ReverseLinkInfo (),* ce qui prévisible. Néanmoins, l'influence de la mobilité reste minime. L'augmentation du nombre de ressources fait diminuer le délai de synchronisation du fait que le nombre de satisfactions de requêtes augmente, ce qui diminue le contenu des files d'attentes et par conséquent diminue le nombre de messages nécessaires aux changements de routes des requêtes.

Les comportements observés durant les différents tests de simulation nous ont permis de déduire une dépendance immédiate du temps moyen d'attente avec le nombre moyen de messages par SC. En effet, plus le temps d'attente augmente, moins le nombre de messages est enregistré. Ce qui laisse à dire que le système stagne quand le délai d'attente augmente et s'active dans le cas contraire, ce qui prévisible. A l'origine de toutes ces constatations, on retrouve la dépendance avec le nombre de ressources disponibles.

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. Il est à noter que, étant donné que le protocole se base sur des structures logiques de DAG, leur maintien génère des messages additionnels surtout dans le cas de forte mobilité.

Les résultats que nous avons obtenus donnent une idée des performances de notre protocole face à des situations réelles. D'autres tests permettent de mieux se rendre de ses

performances, citons par exemple, la variation du nombre de nœuds, la recherche du nombre optimal de ressources en fonction du nombre de nœuds, etc.

Etant donné que le protocole décrit dans ce présent chapitre et celui décrit dans le Chapitre 7 ont été simulés dans deux environnements très proches et avec pratiquement les mêmes paramètres, une comparaison est alors envisageable. Celle-ci est possible pour deux métriques essentielles, le nombre moyen de messages par SC (voir Figure 46 du Chapitre 7 et Figure 59((b) du Chapitre 8) et le délai de synchronisation (voir Figure 47 du Chapitre 7 et Figure 59(c) du Chapitre 8). En comparant les courbes de ces graphes liés au nombre de messages, on constate qu'elles ont toutes une allure décroissante mais celles du premier protocole convergent vers la valeur *10* quelque soit la mobilité. Pour les faibles charges, le second protocole donne de meilleurs résultats. Ceci est du principalement à l'espacement de requêtes dans le premier protocole ce qui oblige les nœuds à diffuser leurs requêtes. Par contre, pour les charges moyennes et fortes c'est au tour du premier protocole de donner des résultats meilleurs. Ceci est du principalement à la diminution sensible des diffusions dans le premier protocole et aux messages supplémentaires liés aux restructuration des structures dans le second protocole. Quant aux graphes du délai de synchronisation, on constate clairement que le premier protocole donne de meilleurs résultats pour toutes les charges. Ce résultat est prévisible car le deuxième protocole engendre un coût élevé de messages supplémentaires liés aux restructurations des deux structures logiques, ce qui est prévisible. En conclusion, il s'avère que la solution du Chapitre 7 (qui ne fait pas recours aux structures logiques) donne des performances meilleures.

5. Conclusion

Dans ce chapitre, nous avons présenté une autre solution au problème de *h-k*- exclusion mutuelle dans les réseaux mobiles ad hoc. Cette solution ne fait pas recours à la couche de routage et se base sur l'utilisation de deux structures logiques de DAG à niveaux correspondant aux deux jetons qui y circulent respectivement. La première, i.e. la structure de DAG distribution permet la distribution de ressources aux nœuds demandeurs et la récupération de ressources libérées et la seconde, i.e. la structure de DAG racine permet de récolter les requêtes sur les ressources. Ces deux structures sont mappées directement sur les liens physiques du réseau sous-jacent. Différents mécanismes ont été définis pour récolter et satisfaire les requêtes en se basant sur les chemins définis par ces structures. Ces structures qui sont maintenues constamment lors des déplacements de jetons et des changements de liens physiques entre les nœuds en faisant recours à la méthode d'inversion de liens afin d'assurer la continuité du service. Le maintient de plusieurs chemins vers les nœuds détenteurs de deux jetons permet de minimiser le recours à l'inversions de liens. L'harmonie de fonctionnement entre les mécanismes associés aux déplacements de jetons, à la circulation des requêtes et de ressources et à la distribution des ressources fait que le protocole favorise l'efficacité du service. En comparaison avec la solution similaire de J-R. Jiang [55], cette structuration et les mécanismes associées permettent d'une part, d'équilibrer mieux la charge du service de *h-k* EM et d'autres part de réduire le nombre de messages échangés notamment ceux liés à la réorganisation des deux structures de DAG, et tout cela pour réduire et équilibrer la consommation d'énergie entre les différents nœuds.

A partir des différents tests ainsi que de l'ensemble des métriques effectuées, nous pouvons constater que le protocole donne des résultats encourageants dans sa globalité. Le délai d'attente par SC en fonction de la charge est stable pour tous les cas de mobilité dans les conditions de faible et moyenne charge, et s'élève progressivement pour s'accentuer en pic

quand la charge devient forte par le manque de ressources disponibles. En effet, les courbes reprennent leur allure stable lorsque le nombre de ressources augmente. Le nombre de messages diminue avec l'augmentation de la charge pour tous les cas de mobilité. Ceci montre que le protocole réagit de manière efficace dans les conditions de moyennes et fortes charges. Quand au délai de synchronisation, il diminue progressivement avec l'augmentation de la charge pour toutes les mobilités. La comparaison avec le protocole du Chapitre 7 montre que l'utilisation des structures logiques dans les réseaux mobiles ad hoc est coûteuse en terme d'entretient.

D'autres part, l'augmentation du nombre de ressources sert mieux les nœuds en rendant le système mieux actif. De même, l'augmentation de la connectivité. Néanmoins, les résultats obtenus dans le cas statique sont meilleurs que ceux du cas dynamique. Tous ces résultats sont prévisibles relativement aux mécanismes et outils utilisés par le protocole.

Cependant, comme perspective il est intéressant de réaliser les travaux suivants:

- Rendre le protocole tolérant aux pannes de nœuds, au partitionnement du réseau,
- Construire une structure de DAG dans un réseau dynamique,
- Montrer les limites du protocole relativement à la scalabilité du réseau,
- Adapter le protocole à l'EM simple et à la *k*- EM dans les réseaux mobiles ad hoc et examiner l'apport de ces adaptations vis à vis des protocoles existants dans la littérature et finalement,
- Situer ce protocole avec celui décrit dans le Chapitre 7 et notamment celui de J-R. Jiang [55].

Conclusion générale

L'étude des services fondamentaux d'allocation de ressources est d'une extrême importance pour construire de nouveaux systèmes pour les réseaux mobiles ad hoc. Ces réseaux qui présentent des challenges majeurs à cause des nouvelles caractéristiques introduites et qui ne sont pas connues des systèmes distribués classiques. Dans cette thèse, nous avons abordé trois problèmes liés à l'allocation de ressources dans les réseaux mobiles ad hoc. Ces problèmes sont, en l'occurrence, l'exclusion mutuelle, la *k*- exclusion mutuelle et la *h* parmi *k* exclusion mutuelle qui sont respectivement décrits par les trois parties de ce document.

Dans la première partie, une étude a été réalisée sur les solutions classiques principales et distribuées d'EM connues dans l'environnement distribué statique selon les deux approches principales. Ces solutions sont présentées d'une manière qui permet de regrouper ensemble celles présentant des outils et mécanismes semblables ou proches. Une comparaison sommaire de ces solutions est reprise dans un tableau récapitulatif (Table 1). Les solutions développées pour les réseaux mobiles ad hoc [18] font aussi l'objet d'une description qui permet de ressortir leurs cotés encourageants et leurs insuffisances. Notre protocole d'exclusion mutuelle [19,20, 21] décrit dans la suite de cette partie s'écarte de tous ceux connus dans ce domaine, il se base sur les liens physiques et n'utilise aucune structure logique. Les requêtes sont acheminées suivant un rayon dynamique de diffusion. Le jeton est géré selon une méthode qui combine la circulation et la demande. De plus, la politique de satisfaction des nœuds favorise en même temps les nœuds les plus proches et les requêtes les plus anciennes. La mobilité des nœuds fait partie intégrante de la solution proposée grâce aux différents mécanismes entrepris. Aussi, les pannes de nœuds, le partitionnement, la fusion et par conséquent, tous les problèmes qui en résultent (pertes et duplications de jetons) sont prises en charge dans la solution. A partir des différents tests et métriques effectuées, nous pouvons constater que le protocole donne des résultats satisfaisants dans sa globalité. Le nombre de messages qu'il génère est assez faible surtout pour les moyennes et fortes charges. Le protocole se montre bien adapté à la mobilité, car cette dernière n'influe pas beaucoup sur ses performances globales. Le nombre de messages émis par SC est assez faible surtout pour les moyennes et fortes charges car le rayon tend vers la valeur *1*. Le délai de synchronisation dépend de la mobilité mais se stabilise au fur et à mesure que la charge du réseau augmente. Le nombre d'entrées en SC est proportionnel à la charge. Aussi, le nombre de messages redondants chute avec l'augmentation de la charge. Le taux de redemandes est relativement faible dans sa globalité puisqu'une demande d'entrée en SC n'est que très rarement non satisfaite dès sa première tentative.

Dans la deuxième partie, une étude a été réalisée sur les solutions les plus connues de la *k*-EM dans les systèmes distribués (selon les deux approches principales connues) et sur les concepts sur lesquelles elles s'appuient. Une simple comparaison est alors réalisée sur la base d'un tableau (table 4) qui récapitule les éléments importants concernant ces protocoles. Cette description est suivie de celle plus ou moins en détail et critique de la seule solution de *k*- EM connue dans les réseaux mobiles ad hoc. Notre protocole de *k*- exclusion mutuelle présenté dans la suite de cette partie ne généralise ni n'adapte aucune autre solution qui le précède. Il est paramétré en fonction de l'état du système afin d'interagir avec tous les mouvements qui peuvent survenir. Il est orienté demande de jeton et suppose l'existence de *k* jetons dans le système. Face aux contraintes imposées par l'environnement mobile ad hoc, plusieurs techniques ont été proposées. Celles-ci sont sollicitées et mises en œuvre par le système chaque fois qu'il est nécessaire dans le but de favoriser la sûreté et la vivacité du protocole. Parmi ces techniques, nous citons: les relances, l'utilisation de la file d'attente temporaire dans l'*espoir* de traiter les requêtes ayant perdu le chemin du jeton correspondant, l'utilisation de jetons de type *Collector* pour favoriser la satisfaction d'un grand nombre de requêtes tout en activant des jetons oisifs, l'utilisation des jetons en cours de transit (avec certaines restrictions), la cession de jetons afin d'éviter leur perte même momentanée, la collecte d'informations pour construire une connaissance locale du système sans aucun coût, …etc. En outre la participation des nœuds non intéressés par la SC dans le service de *k*- exclusion mutuelle n'est que partielle. En général, notre protocole de *k*- EM offre ses meilleures performances dans les conditions de forte charge comparées à celles de faibles charges. Néanmoins, dans les conditions de faible charge, il réagit de manière efficace surtout lorsque les jetons sont en nombre suffisant. La variation de la connectivité ne semble pas avoir une grande influence sur les performances du protocole. Cependant, l'utilisation des nombres premiers crée une lenteur dans le calcul spécialement pour les hautes connectivités; ce qui nécessite une grande puissance de calcul. La comparaison avec le protocole *KRL* [147] a été possible seulement dans le cas de mobilité nulle et le cas mobile avec une faible mobilité en prenant comme critères, le nombre moyen de messages générés par entrée en SC et la durée d'attente moyenne pour une entrée en SC. D'après les graphes obtenus, il semble que les résultats offerts par notre protocole sont meilleurs que ceux du protocole *KRL* relativement aux métriques communes.

Dans la troisième partie, une étude a été réalisée sur les solutions les plus connues de la *h* parmi *k* EM dans les systèmes distribués et les concepts y afférents. Une comparaison sommaire de ces solutions est reprise dans un tableau récapitulatif (table 5). Cette description est suivie de celle plus ou moins en détail et critique des deux solutions de *k*- EM connues dans les réseaux mobiles ad hoc. Notre premier protocole de *h*-*k*- EM [16] présenté dans la suite de cette partie reprend certains concepts de la solution dans [20]. Il se base alors sur les liens physiques, n'utilise aucune structure logique. Les requêtes sont acheminées suivant un rayon d'envoi dynamique. Le jeton est géré selon une méthode qui combine la circulation et la demande. De plus, la politique de satisfaction des nœuds favorise en même temps les nœuds les plus proches, les requêtes les plus anciennes et les demandeurs de moins de ressources (ce qui évite la sous utilisation de ressources). La mobilité des nœuds fait partie intégrante de la solution proposée grâce aux différents mécanismes présentés dans ce qui précédait. Aussi, le protocole prend en charge les pannes de nœuds, le partitionnement, le fusionnement et par conséquent, tous les problèmes qui en résultent (perte de jeton, perte de ressources, duplications de jetons et de ressources). Les résultats de simulation montrent que le protocole donne des résultats satisfaisants dans sa globalité. Ces résultats sont en accord avec ceux obtenus dans le protocole d'exclusion mutuelle cadre [20]. Les complexités des différentes métriques sont à peu près semblables, mais avec des valeurs ajoutées induites par les messages liés à la gestion de ressources. Le nombre de messages nécessaires à une entrée

en SC est assez faible surtout pour les moyennes et fortes charges. Le protocole se montre bien adapté à la mobilité, car cette dernière n'influe pas beaucoup sur ses performances globales. Le nombre de messages émis par SC est assez faible surtout pour les fortes charges car le rayon tend vers la valeur *1,2*. Le délai de synchronisation dépend de la mobilité mais se stabilise au fur et à mesure que la charge du réseau augmente en convergeant vers *2.5*. Le taux de satisfaction tend vers la valeur *1* pour les moyennes et fortes charges, les traces d'exécution montrent que les requêtes non satisfaites sont celles qui viennent juste avant la fin de la simulation; ils doivent attendre la libérations de ressources. Aussi, le nombre de messages redondants chute avec l'augmentation de la charge. Le taux de redemandes dans sa globalité est relativement faible lorsque la charge augmente puisqu'une demande d'entrée en SC est satisfaite après *0,5* redemande. Notre second protocole présenté au Chapitre 8 se base également sur les liens physiques, ne fait pas recours aux services de la couche de routage, mais considère deux structures logiques de DAG à niveaux correspondant aux deux jetons qui y circulent respectivement. La première, i.e. la structure de DAG distribution permet la distribution de ressources aux nœuds demandeurs et la récupération de ressources libérées et la seconde, i.e. la structure de DAG racine permet de récolter les requêtes sur les ressources. Ces deux structures sont mappées directement sur les liens physiques du réseau sous-jacent. Différents mécanismes ont été définis pour récolter et satisfaire les requêtes en se basant sur les chemins définis par ces structures. Ces structures qui sont maintenues constamment lors des déplacements de jetons et des changements de liens physiques entre les nœuds en faisant recours à la méthode d'inversion de liens afin d'assurer la continuité du service. Le maintient de plusieurs chemins vers les nœuds détenteurs de deux jetons permet de minimiser le recours à l'inversion de liens. L'harmonie de fonctionnement entre les mécanismes associés aux déplacements de jetons, la circulation des requêtes, des ressources et la distribution des ressources fait que le protocole favorise l'efficacité du service. En comparaison avec la solution similaire de J-R. Jiang [55], cette structuration et les mécanismes associées permettent d'une part, d'équilibrer mieux la charge du service de *h-k* EM et d'autres part, de réduire le nombre de messages échangés notamment ceux liés à la réorganisation des deux structures de DAG, et tout cela pour réduire et équilibrer la consommation d'énergie entre les différents nœuds. Les résultats de simulation montrent que le protocole donne ses meilleurs résultats pour les moyennes et fortes charges de demandes, que l'augmentation du nombre de ressources est en faveur du service et que la mobilité a peu d'influence sur ses performances. En comparaison avec la première solution donnée au Chapitre 7, il s'avère que la seconde solution donne des résultats légèrement moins performants. Ils sont dus essentiellement au coût supplémentaire de maintient continu des deux structures logiques.

Ce travail réalisé est loin d'être fini, il donne une ébauche de solutions au problème d'allocation de ressources dans un environnement en plein essor. Plusieurs travaux complémentaires futures sont à recenser.

- Il est utile de réaliser une comparaison de performances entre les différentes solutions décrites dans cette thèse, notamment :
  - la solution d'EM décrite au Chapitre III avec la solution de *1*- EM générée de la solution décrite dans le Chapitre V ainsi qu'avec la solution *1- 1* EM générée de la solution décrite dans le Chapitre VII.
  - la solution de *k*- EM décrite dans le Chapitre V avec la solution de *1- k* EM générée de la solution décrite dans le Chapitre VII.
- D'autres part, pour mieux gérer la réaction du service d'allocation de ressources vis à vis d'autres caractéristiques introduites par ce type de réseaux, il est utile :
  - d'anticiper le changement de topologie de communication et la déconnexion volontaire de noeuds du réseau, et

- d'intégrer des mécanismes spécifiques de conservation d'énergie, notamment la gestion du mode veille.

- Il est aussi utile de réaliser une comparaison de la solution donnée au Chapitre 8 avec celle de J-R. Jiang [55] qui partage avec elle quelques idées.
- Pour tous les protocoles conçus, les simulations réalisées n'ont pas tenus compte de tous les aspects de ces protocoles. Il est donc utile de les mettre en valeur à travers de nouvelles simulations pour déterminer leurs influences sur les performances de ces protocoles.
- Les problèmes d'allocation de ressources sont aussi variés qu'ils en existent d'autres. Si on continue dans le chemin de généralisation, un processus peut effectuer une seule demande sur plusieurs classes de ressources et chacune avec un nombre donné d'exemplaires [26, 145]. Les lecteurs/rédacteurs est un autre problème classique qui consiste en le partage d'une seule ressource avec la contrainte que les rédacteurs utilisent chacun la ressource en exclusion mutuelle. Nous citons aussi l'exclusion mutuelle de groupes (GEM) [29, 63, 64, 151, 65, 128] qui s'inspire de l'exclusion mutuelle et le Group *k*- exclusion mutuelle (ou GK- EM) [59] qui est plus restrictif que le GEM. Ces deux derniers problèmes sont notre préoccupation actuelle.
- A travers cette thèse, il ressort clairement une difficulté majeure d'écriture de solutions à partir de sa spécification (i.e. conception). Pour résoudre ce problème qui dépasse le domaine d'allocation de ressources, il est utile de penser à des méthodologies permettant la génération de textes de protocoles à partir de spécifications. Parallèlement, des langages d'écritures de textes de protocoles s'avèrent d'une extrême importance.

Bibliographie

[1] Agrawal and A.El abbadi. ‘’Analysis of quorum-based protocols for distributed (K+1)-mutual exclusion, ’’ *IEEE Transactions on Parallel and Distributed Systems*, 1997.

[2] D. Agrawal and A. El Abbadi, “An efficient and fault-tolerant solution for distributed mutual exclusion,” *ACM Transactions on Computer Systems*, Vol. 9, N°. 1, pp. 1-20, Feb. 1991.

[3] D. Agrawal and A. El Abbadi, ‘’An efficient solution to the distributed mutual exclusion problem,‘’ In *Principles of Distributed Computing*, pages 193–200, August 1989.

[4] D. Agrawal and A. E. Abbadi, ’’Exploiting logical structures in replicated databases,’’ *Information Processing Letters*, N° 33, pp. 255–260, 1990.

[5] D. Agrawal and A. El Abbadi, ‘’The generalized tree quorum protocol: An efficient approach for managing replicated data,’’ University of California

[6] D. Agrawal and A. El Abbadi, “The generalized tree quorum protocol: An efficient approach for managing replicate data,” *ACM Transactions on Databases Systems*, Vol. 17, N° 4, pp. 689-717, Dec. 1992.

[7] A. A. Albert And R. Sandler, “An introduction to finite projective planes,” *Holt, Rinehart and Winston*, New York 1968.

[8] B.R. Badrinath, A . Acharya, and T. Imielinski, “Structuring distributed algorithms for mobile hosts,” *Proc. of the 14th Intern. Conf. on Distr. Comp*., pp. 21-28, 1994.

[9] R. Baldoni, ‘’ An O($n^{M/(M+1)}$) distributed algorithm for the *k*-out-of- *M* resource allocation problem,’’ *Proc. of 14-th IEEE Int. Conf. on Distributed Computing Systems*, Pozman, Poland, pp. 81-87, 1994.

[10] R. Baldoni and B. Ciciani, ‘’Distributed algorithms for multiple entries to a critical section with priority,’’ *Information Processing Letters*, Vol. 50, pp. 165–172, 1994.

[11] R. Baldoni, ‘’Mutual Exclusion in Distributed Systems,‘’ *PhD Thesis*, Universita di Roma, La Sapienza, (1994).

[12] R. Baldoni, A. Virgillito, and R. Petrassi, ''A distributed mutual exclusion algorithm for mobile ad-hoc networks,'' *Proc. of the Seventh IEEE Symposium on Computers and Communications (ISCC 2002)*, Taormina, Italy, pp. 539 – 545, 1 – 4, July 2002.

[13] J. Bar-Ilan and D. Peleg, ''Distributed resource allocation algorithms,'' *Lecture Notes in Computer Science 647 (WDAG'92)*, pp. 276-291, 1992.

[14] M. Benchaïba, **"** *A new solution to the h out of k problem in mobile ad hoc networks*, '' Laboratoire des Systèmes Informatiques, Dept. Informatique- Fac. Electronique & Informatique- USTHB, B.P. 32 El-Alia, Bab Ezzouar- Alger – Algérie, benchaiba@lsi-usthb.dz, LSI-TR-0108, Janvier 2008.

[15] M. Benchaïba, **"** *Towards a solution to the h out of k problem in mobile ad hoc networks*, '' Laboratoire des Systèmes Informatiques, Dept. Informatique- Fac. Electronique & Informatique- USTHB, B.P. 32 El-Alia, Bab Ezzouar- Alger – Algérie, benchaiba@lsi-usthb.dz, LSI-TR-0807, Novembre 2007.

[16] M. Benchaïba and M. Ahmed nacer, ''A distributed token based *h*-out of-*k* mutual exclusion protocol for mobile ad hoc networks,'' ***Accepted in*** *The International Journal of Ad Hoc and Ubiquitous Computing(IJAHUC),* Vol.0, N. 0, 2010?

[17] M. Benchaïba and M. Ahmed Nacer, "A Token Based IP Address Assignment Protocol for Mobile Ad Hoc Networks", *LSI-TR-1504*, November 2004, LSI, Dept. of Computer Sciences, USTHB, B.P. 32 El-Alia, Bab Ezzouar- Algiers – Algeria.

[18] M. Benchaïba, A. Bouabdallah, N. Badache, and M. Ahmed-Nacer, ''Distributed mutual exclusion algorithms in mobile ad hoc networks: An overview,'' *ACM SIGOPS Operating Systems Review*, Vol. 38 , Issue 1, pp. 74 – 89, Jan. 2004.

[19] M. Benchaïba et M. Haddad**, "** Une approche de solution au problème d'exclusion mutuelle pour les réseaux mobiles ad hoc, '' *Proceedings de l'ISPS'07, Alger- Algérie*, pp. 215-225, 5-7 Mai 2007.

[20] M. Benchaïba, M. Haddad and M. Ahmed-Nacer, '' Un protocole d'exclusion mutuelle basé jeton pour les réseaux mobiles ad hoc,'' *LSI-TR-0406*, Mars 2006, LSI, Dept. of Computer Sciences, USTHB, B.P. 32 El-Alia, Bab Ezzouar- Algiers – Algeria.

[21] M. Benchaïba, M. Haddad, et M. Ahmed-Nacer,'' Vers une solution au problème d'exclusion mutuelle dans les réseaux mobiles ad hoc, '' *Actes du colloque francophone GRES 2006*, pp. 163-182, 9-12 Mai 2006.

[22] A. Bouabdallah. and J.C. König, ''An improvement of the Maekawa's mutual exclusion algorithm to make it fault-tolerant,'' *Parallel Processing Letters*, Vol. 2, pp. 283-290, 1992.

[23] A. Bouabdallah , ''On mutual exclusion in faulty distributed systems,'' *ACM Operating Systems Review*, Vol. 28, N° 1, pp. 80-87, 1994.

[24] A. Bouabdallah and J.C. König, ''A distributed algorithm for the mutual exclusion problem,'' *Parallel and Distributed Computing in engineering systems. Tzafestas et al. (Editeurs), Elsevier Science Publisher B.V*, North- Holland, pp. 285-290, 1992.

[25] A. Bouabdallah., J.C. König, and M.B Yagoubi, ‘’A fault-tolerant algorithm for the mutual exclusion in real- time distributed systems,’’ *Journal of computing and Information*, Vol. 1 N° 1, pp. 438-454, 1994.

[26] A. Bouabdallah and C. Laforest, ‘’ A distributed token-based algorithm for the dynamic resource allocation problem,’’

[27] S. Bulgannawar and N.H. Vaidya ‘’A distributed k-mutual exclusion Algorithm,’’ *Proc. of the 15th Int. Conf. on Distributed Computing Systems*, pp. 153-160, 30 May- 2 Jun, 1995.

[28] O. Calvalho and G. Roucairol. ‘’On mutual exclusion in computer networks,’’ *Technical correspondence, Communication of the ACM*, Vol. 26, N°2, pp. 146-147, Feb. 1983.

[29] S. Cantarell, A. K. Datta, F. Petit, and V. Villain, ‘’Group mutual exclusion in token rings,’’ in *the 8th International Colloquium on Structural Information and Communication Complexity Proceedings*, pp. 61-76, 2001.

[30] G. Cao, M. Singhal “An Adaptive Distributed Channel Allocation Strategy for Mobile Cellular Networks,” *Journal of Parallel and Distributed Computing, Vol.* 60, pp. 451-473 (2000).

[31] I. D. Chakeres and E. M. Belding-Royer, “The utility of hello messages for determining link connectivity,” in Proceedings of the International Symposium on Wireless Personal Multimedia communications, Sheraton Waikiki, Honolulu, Hawaii, pp. 504-508, October, 2002.

[32] K.M. Chandy and L. Lamport, ‘’Distributed snapshots: Determining global states of distributed systems,’’ *ACM Trans. On Computer Systems*, Vol. 3 N°1, pp. 63-75, 1985.

[33] K.M. Chandy and J. Misra, ’’The drinking philosophers problem,’’ *ACM Transactions on programming languages and Systems*, Vol. 6, N°4, pp. 632-646, 1984.

[34] Y-I. Chang and B-H. Chen, ‘’An extended binary tree quorum strategy for *k*- mutual exclusion in distributed systems,’’ *in Proceedings of the Pacific Rim International Symposium on Fault-Tolerant Systems (PRFTS'97)*, pp. 110-115, Dec. 1997.

[35] E. J. H. Chang and R. Roberts, ’’An improved algorithm for decentralized extrema-finding in circular configurations of processes,’’ Communications of the. ACM Vol. 22, N° 5, pp. 281-283, May 1979.

[36] Y. Chang, M. Singhal, and M. Liu, “A fault tolerant algorithm for distributed mutual exclusion,” *In Proc. of 9th IEEE Symp. On Reliable Dist. Systems*, pp. 146-154, 1990.

[37] Shun Yan Cheung, Mustaque Ahamad, and Mostafa H. Ammar, ‘’Multi-dimensional voting: A general method for implementing synchronization in distributed systems,’’ *In Proceedings of 10th International Conference of Distributed Computing Systems*, pp. 362–369, 1990.

[38] Y. Chen and J. L. Welch,''Self-stabilizing mutual exclusion using tokens in mobile ad hoc networks,'' *DIAL-M 2002*, pp. 34-42.

[39] IEEE Computer Society LAN MAN Standards Committee, ''Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications,'' *IEEE Std 802.11-1997, The Institute of Electrical and Electronics Engineers*, New York, New York, 1997.

[40] M. Choy and A. K. Singh,'' Efficient fault-tolerant algorithms for distributed resource allocation,'' *ACM Transactions on Programming Langages and Systems*, Vol. 17, N° 3, pp. 535-559, 1995.

[41] D.M. Dhamdhere and S.S. Kulkarni, ''A token based k-resilient mutual exclusion algorithm for distributed systems,'' *Information Processing Letters*, vol. 50, pp. 151-157, 1994.

[42] E. Dijkstra, ''Self-stabilization in spite of distributed control,'' *Communication of the ACM*, 17, pp. 643-644, 1974.

[43] L.M. Feeney, ''A Taxonomy for routing protocols in mobile ad hoc networks,'' *In International Symposium on Handheld and Ubiquitous Computing (HUC'99), volume 8*, Karlsruhe, Germany, September 1999.

[44] E. Gafni and D. Bertsekas, "Distributed algorithms for generating loop-free routes in networks with frequently changing topology," *IEEE Transactions on Communications*, Vol. Com-29 , N°1, pp. 11-18, 1981.

[45] H. Garcia-Molina and D. Barbara, ''How to assign votes in a distributed systems principles,'' *Journal of the ACM*, Vol. 32, N^{o} 4, pp. 841- 860, 1985.

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

[47] Ronald L.Graham, Donald E.Knuth et Oren Patashnik ''Mathématiques concrètes,'' *International Thomson publishing*, 2^{ième} édition, 1997.

[48] Z.J. Haas, and M.R. Pearlman, "The performance of query control schemes for the zone routing protocol," *IEEE/ACM Transactions on Networking*, Vol. 09, N° 4, pp. 427-438, 2001.

[49] J. Helary, N. Plouzeau, and M. Raynal, "A distributed algorithm for mutual exclusion in an arbitrary network," *Computer Journal*, Vol. 31, N°4, pp. 289-295, 1988.

[50] J.M. Helary, N. Plouzeau et M. Raynal, ''A distributed algorithm for mutual exclusion in an arbitrary network,'' *Rapport technique N°281*, 16 pages, IRISA 1986.

[51] A. Housni, ''Introduction de la priorité dans les algorithmes d'exclusion mutuelle répartis," *Thèse 2002*, école doctorale d'informatique, d'automatique et de productique.

[52] IEEE std. 802.11, "Wireless LAN media access control (MAC) and physical layer (PHY) specification," 1999.

[53] G. Jakllari, W. Luo, and S. V. Krishnamurthy, ''An Integrated Neighbor Discovery and MAC Protocol for Ad Hoc Networks Using Directional Antennas,'' IEEE Transactions on Wireless Communications, Vol. 6, N° 3, March 2007.

[54] Jennifer E. Walter, ''Distributed algorithms for mobile computing systems,'' *Ph.D. dissertation*, Department of Computer Science, Texas A&M University, College Station, December, 2000.

[55] J-R. Jiang, "A distributed *h*-out of-*k* mutual exclusion algorithm for ad hoc mobile networks," *Proceeding of the 2nd International Workshop on Parallel and Distributed Computing Issues in Wireless Networks and Mobile Computing*, Marriott Marina, Fort Lauderdale, Florida, USA, pp. 196-202, April 2002.

[56] J-R. Jiang, ''A fault-tolerant *h*-out of-*k* mutual exclusion algorithm using cohorts coteries for distributed systems'' in *Proc. of the 5th International Conference on Parallel and Distributed Computing, Application and Technologies PDCAT'04*, pp. 267-273, 2004.

[57] J-R. Jiang, S-T. Huang, Y-C. Kuo, ''Cohorts structures for fault-tolerant *k* entries to a critical section,'' *IEEE Transactions on Computers*, Vol. 48, N°. 2, pp. 222 – 228, February 1997.

[58] J-R. Jiang, ''Distributed *h*-out of-*k* mutual exclusion using *k*-coteries,'' *3rd International Conference on Parallel and Distributed Computing, Application and Technologies (PDCAT'02)*, Kanazawa, Japan, pp.218–226.

[59] J-R. Jiang, ''Group k- mutual exclusion for distributed systems,'' *Proc. of the 15th IASTEAD Int. Conf. on Parallel and Distributed Computing and Systems*, Nov. 3-5, 2003, USA.

[60] J-R. Jiang, "Quorum structures for fault-tolerant distributed mutual exclusion," *Ph.D. Dissertation*, Tsing Hua University, July 1995.

[61] J. Jiang, T-H. Lai, N. Soundarajan "On Distributed Dynamic Channel Allocation in Mobile cellular Networks, " *IEEE Trans. on Parallel and Distributed Syst*ems, Vol. 13, pp. 1024 – 1037, Oct. 2002

[62] D. Johnson and D.A. Maltz, "Dynamic source routing in ad hoc wireless networks," *In T. Imielinski and H. Korth editors, Mobile Computing*. Kluwer Academic Publishers, Boston, pp. 153-181, 1994.

[63] Y.-J. Joung, ''Asynchronous group mutual exclusion,'' *Distributed Computing (DC)*, Vol.13, N°4, pp.189-206, 2000.

[64] Y.-J. Joung, ''Quorum-based algorithms for group mutual exclusion,'' *IEEE Transactions on Parallel and Distributed Systems*, Vol. 14, N° 5, pp. 463 - 475, May 2003.

[65] Y.J. Joung, "The Congenial talking philosopher problem in computer networks," *Distributed computing*, Vol. 15, Pages 155-175, 2002.

[66] R. Jurdak, C. V. Lopes, and P. Baldi, ‘’A survey, classification and comparative analysis of medium access control protocols for ad hoc networks, ‘’ IEEE Communications Survey, First Quarter 2004, Col. 6, N° 1, pp. 2-16.

[67] H. Kakugawa, S. Fujita, M. Yamashita, and T. AE, "Availability of k-coterie," *IEEE Transactions on Computers*, Vol. 42, N° 5, pp. 553-558, May1993.

[68] H. Kakugawa, S. Fujita, M. Yamashita, and T. AE, ‘’A distributed k- mutual exclusion algorithm using K-coterie’’ *Information Processing Letters*, vol.49, Issue 4, pp.213-218, Mars 1994.

[69] H. Kakugawa and M. Yamashita, “Local coteries and a distributed resource allocation algorithm,” *Transactions of Information Processing Society of Japan*, Vol. 37 N°. 8, pp. 1487-1495, Aug. 1996.

[70] I. Katzela and M. Naghshineh, “Channel assignment schemes for cellular mobile telecommunication systems: A comprehensive survey,“ *IEEE Personal Communications*, pp. 10-31, June 1996.

[71] L. Kleinrock and F. A. Tobari, “Packet switching in radio channels: Part I- Carrier sense multiple-access modes and their throughput-delay characteristics,” *IEEE Trans. Commun*., Vol. Com-23, pp. 1400-1416, December 1975.

[72] A. Kumar, ‘’Hierarchical quorums consensus: A new algorithm for managing replicated data,’’ *IEEE Transactions for Computers*, Vol. 40, N°9, pp. 996–1004, 1991.

[73] Y-C. Kuo and S-T. Huang, ‘’A geometric approach for constructing coteries and *k*-coteries,’’ *IEEE Transactions on Parallel and Distributed Systems*, Vol. 8, N° 4, pp. 402-411, April 1997.

[74] Y-C. Kuo, ‘’Composite *k*-arbiters,’’ *IEEE Transactions on Parallel and Distributed Systems*, ‘’ Vol. 12, N° 11, pp. 1134 - 1145, November 2001.

[75] Y-C. Kuo, ‘*’k*-arbiter join operation,’’ *Journal of Information Science and Engineering*, Vol. 17, pp. 615-631, 2001.

[76] L. Lamport, “Time, clocks, and the ordering of events in a distributed system,” *Communications of the ACM*, Vol. 21, N°7, pp. 558-565, July 1978.

[77] G. Le Lann, “Distributed systems, towards a formal approach,” *IFIP Congress*, Toronto, pp. 155-160, 1977.

[78] T. Larsson and N. Hedman, ‘’Routing protocol in wireless ad-hoc: A simulation study,’’ *Master thesis,* Lulea University of Technology, Stockholm*,* 1998.

[79] C. Macchi, J.E. Guilbert, et treize co-auteurs, ‘’ Téléinformatique- Transport et traitement de l’information dans les réseaux et systèmes téléinformatiques et télématiques,’’ *Dunod informatique*, 1987.

[80] M. Maekawa, "A √N algorithm for mutual exclusion in decentralized systems," *ACM Transactions on Computer Systems*, Vol. 3, N° 2, pp. 145-159, May 1985.

[81] M. Maekawa, A.E. Oldehoeft and R.R. Oldehoeft, ''Operating systems, Advanced concepts,'' *Benjamin-Cummings*, 1987.

[82] M. 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, Dec. 1992.

[83] N. Malpani, Yu Chen, N. H. Vaidya and J. L. Welch**,** ''Distributed token circulation in mobile ad hoc networks,'' *IEEE Transactions on Mobile Computing*, Volume 4, Issue 2, pp. 154 - 165, March-April 2005.

[84] Y. Manabe and S. Aoyagi, '' A distributed *k*- mutual exclusion algorithm using k-coterie,'' *IEICE Technical Report COM93-43*, September 1993.

[85] Y. Manabe, R. Baldoni, M. raynal, and S. Aoyagi, ''*k*- abriter: A safe and general scheme for h- out k- mutual exclusion problems,'' *Theoretical Computer Science*, Vol. 193, No. 1-2, pp. 97-112, Feb. 1998.

[86] Y. Manabe and N. Tajima, ''(h-k) Arbiter for h-out of-k resources allocation problem,'' *Theoretical Computer Science*, Vol. 310, pp. 379-392, 2004.

[87] D. L. Mills, ''Internet time synchronization: The network time protocol,'' *IEEE Transaction on Communications*, Vol. 39, N°. 10, pp. 1482-1493, 1991.

[88] S. Mishra and P. Srimani, "Fault-tolerant mutual exclusion algorithms," *Journal of Systems software*, Vol. 11, N°2, pp. 111-129, Feb. 1990.

[89] A. Misra, S. Das, A. McAulley, and S. K. Das, ''Autoconfiguration, Registration, and Mobility Management for Pervasive Computing,'' *IEEE Personal Communications*, Vol. 8, Issue 4, pp. 24-31, August 2001.

[90] M. Mizuno, M.L. Neilsen, and R. Rao, "A token based distributed mutual exclusion algorithm based on quorum agreements," *11th International Conference on Distributed Computing Systems*, pp. 361-368, 20-24 May 1991.

[91] M. Mock, R. Frings, E. Nett, and S. Trikaliotis, ''Clock synchronization for wireless local area networks,'' *in Proceedings of the 12th Euromicro Conference on Real Time Systems*, Stockholm, 2000, pp. 183-189.

[92] F. Mueller, ''Prioritized token-based mutual exclusion for distributed systems,'' In *International Parallel Processing Symposium*, pages 791–795, 1998.

[93] M. Naimi and M. Trehel, ''A distributed algorithm for mutual exclusion based on data structures and fault tolerance, '' *Proc. IEEE Phoenix Conf. on Communication*, pp. 256-276, (1987).

[94] M. Naimi, ‘’Distributed algorithm for *k*- entries to critical section based on the directed graphs,’’ *Operating Systems Review*, Vol. 27, N° 4, pp. 67-75, 1993.

[95] M. Naimi and M. Trehel, “An improvement of the log n distributed algorithm for mutual exclusion,” *Proceedings of 7th IEEE International Conference on Distributed Computing Systems*, pp. 371-377, 1987.

[96] M. Naimi and M. Trehel, ‘’How to detect a failure and regenerate the token in the log(n) distributed algorithm for mutual exclusion,’’ Proc. of the Second Int Workshop on Distributed Algorithms, *Lecture Notes in CS*, pp. 155-166, 1987.

[97] K. Nakano and S. Olariu, “A Survey on leader election protocols for radio networks,” *International Symposium on Parallel Architectures, Algorithms and Networks* (ISPAN '02), Philippines, May 2002.

[98] M.L. Neilsen and M. Mizuno, “A DAG-based algorithm for distributed mutual exclusion,” *11th International Conference on Distributed Computing Systems*, pp. 354-360, 20-24 may, 1991.

[99] M. L. Neilsen and M. Mizuno, ‘’Coterie join algorithm,’’ *IEEE Transactions on parallel and Distributed Systems*, Vol. 3, N° 5, pp. 582- 590, 1992.

[100] M. Neilsen and M. Mizuno, “No dominated k- coteries for multiple mutual exclusion,” *Information Proceeding Letters*, 50, pp. 247- 252, 1994.

[101] M. L. Neilsen, M. Mizuno, and M. Raynal, ‘’A general method to define quorums,’’ In *Proceedings of 12th International Conference of Distributed Computing Systems*, pp. 657–664, 1992.

[102] S. Nesargi, R. Prakash, ‘’Distributed Wireless Channel Allocation in Networks with Mobile Base Stations’’, *IEEE Transactions on Vehicular Technology,* Vol. 51, N° 6, pp. 1407-1421, November 2002.

[103] S. Nesargi and R. Prakash, “MANETConf: Configuration of hosts in a mobile ad hoc network,” *In the Proceedings of IEEE INFOCOM ’02*, New York, June 2002.

[104] S. Nishio, K.F. LI, and E.G. Manning, ‘’A resilient mutual exclusion algorithm for computer networks,’’ *IEEE Transactions on Parallel and Distributed Systems*, Vol. 1, No. 3, pp. 344-355, July 1990.

[105] V. Park and S. Corson, “Temporally-ordered routing algorithm (TORA) version 1 functional specification,” *corson-draft-ietf-manet.tora-spec-00.txt*, *IETF*, Internet draft, 1997.

[106] B. Patt-Shamir and S. Rajsbaum, ‘’A theory of clock synchronization,’’ *in Proceedings of 26th Annual ACM Symposium of Theory of Computing (STOC)*, pp. 810-819, 1994.

[107] C.E. Perkins and P. Bhagwat, “Highly dynamic destination sequenced distance-vector routing (DSDV) for mobile computers,” *Proc. ACM SIGCOMM 1994*, pp. 234-244, 1994.

[108] C.E. Perkins and E.M. Royer, “Ad-hoc on-demand distance vector (AODV) routing,” *Proc. IEEE Workshop on Mobile Computing Systems and Applications (WMCSA 1999)*, pp. 90-100, 1999.

[109] K. Raymond, ’’A distributed algorithm for multiple entries to a critical section,’’ *Information Processing Letters*, 30:189–193, February 1989.

[110] K. Raymond, “A tree-based algorithm for distributed mutual exclusion,” *ACM Transactions on Computer Systems*, Vol. 7 N°. 1, pp. 61-77, Feb. 1989.

[111] M. Raynal, ‘’A distributed solution to the *k*-out of-*m* resources allocation problem,’’ *In Lecture Notes in Computer Science*, 497, pp. 599–609. Springer-Verlag, 1991.

[112] M. Raynal, “Prime numbers as a tool to design distributed algorithms,” *Information Processing Letters*, Vol. 33, N°. 1, pp. 53-58, Oct. 1989.

[113] M. Raynal, ‘’Simple taxonomy for distributed mutual exclusion algorithms,’’ *Operating Systems Review*, Vol. 25, N° 2, pp. 47-49, April 1991.

[114] M. Raynal, ‘’Synchronisation et état global dans les systèmes répartis,’’, *Tome 2 d’une introduction aux principes des systèmes répartis*, ISSN 0399-4198.

[115] M. Raynal, ‘’Synchronization and global states in distributed system,’’ *Ed. Eyrolles*, 202 pages, 1992.

[116] M. Raynal et J.M. Helary, ‘’ Synchronisation et contrôle des systèmes et des programmes répartis’’, Editions Eyrolles, 1988.

[117] G. Ricart and A. K. Agrawala, “An optimal algorithm for mutual exclusion in computer networks,” *Communications of the ACM*, Vol. 24, N°1, pp. 9-17, Jan. 1981.

[118] G. Ricart and A.K. Agrawala, “Author response to ‘on mutual exclusion in computer networks’, by Carvalho and Roucairol,” *Communication of the ACM*, vol. 26, N°2, pp. 147-148, Feb. 1983.

[119] E. M. Royer and T. Chai-Keong, ‘’A review of current routing protocols for ad hoc mobile wireless networks,’’ *IEEE Personal Communications*, Vol. 6, No. 2, pp. 46-55, April 1999.

[120] B. Sanders, “The information structure of distributed mutual exclusion algorithms,” *ACM Transactions on Computer Systems*, Vol. 5, N° 3, pp. 284-299, Aug. 1987.

[121] A. Silberschatz et P.B. Galvin, ‘’ Principes des systèmes d’exploitation,’’ *4 eme édition, Addison Wesley,* 1994.

[122] A. Silberschatz and I.L. Peterson, ‘’Operating System Concepts,’’ *Addison-Wesley*, Alternate edition, 1988.

[123] S. Singh and J.F. Kurose, ‘’Electing good leaders,’’ *Journal of Parallel and Distributed Computing*, vol. 21, no. 2, pp. 184-201, May 1994.

[124] M. Singhal, "A heuristically-aided algorithm for mutual exclusion in distributed systems," IEEE Trans. On Computers Vol. 38 N°5, pp. 651-662, may 1989.

[125] M. Singhal, ''A taxonomy of distributed mutual exclusion," *Journal of Parallel and Distributed Computing*, Vol. 18, N°1, pp. 94-101, 1993.

[126] M. Singhal and D. Manivannan, ''A distributed mutual exclusion algorithm for mobile computing environments,'' ICIIS'97, Dec. 1997.

[127] P. K. Srimani and R. L. N. Reddy, ''Another distributed algorithm for multiple entries to a critical section,'' *Information Processing Letters*, 41(1):51–57, January 1992.

[128] A. Swaroop, and A. K. Singh, '' A distributed group mutual exclusion algorithm for soft real time systems,'' *Proc. of World Academy of Science*, Engineering and Technology, Vol. 26, Dec. 2007.

[129] I. Stojmenovic, ''Handbook of Wireless Networks and Mobile Computing,'' *NewYork, NY: Wiley Series on Parallel and Distributed Computing*, 2002.

[130] M. Trehel and R. Housni, '' Introduction of the priority in distributed mutual exclusion algorithms,'' *SCI'99&ISAS'99,* Orlando, U.S.A. July 31, August 4, 1999.

[131] Wu Shu, ''An efficient distributed token-based mutual exclusion algorithm with a central coordinator,'' *Journal of Parallel and Distributed Processing*, Vol. 62, N°10, pp. 1602-1613, 2002.

[132] W. Su, Sung-Ju. Lee, and M. Gerla, "Mobility prediction and routing in ad hoc wireless networks*," Int. Journal of Network Management*, Vol. 11, Issue 1, Jan-Feb. 2001.

[133] I. Suzuki and T. Kazami, "A distributed mutual exclusion algorithm," *ACM Transactions Computer Systems*, Vol. 3 N°4, pp. 344-349, 1985.

[134] A. Tanembaum,'' Systèmes d'exploitation,'' *3 eme édition, Nouveaux Horizons*, 2008.

[135] J. Tajima and K. Imamura, "A strategy for flexible channel assignment in mobile communication systems," *IEEE Trans. Vehicular Tech*nology, Vol. 37, N°2, May 1988.

[136] V.P. Team, "The network simulator-ns2," *VINT Project Team*, Available at http://www.isi.edu/nsnam/ns/, Nov. 2000.

[137] R. H. Thomas, ''A majority consensus approach to concurrency control for multiple copy databases,'' *ACM Transactions on Database Systems*, Vol. 4, N° 2, 1979.

[138] S. Thomson and T. Narten,''IPv6 stateless autoconfiguration,'' *RFC 2462*, December 1998.

[139] F. A. Tobari and L. Kleinrock, "Packet switching in radio channels: Part III- Polling and (dynamic) split-channel reservation multiple access,'' *IEEE Trans. Commun*., Vol. Com-24, N°8, pp. 832-845, August 1976.

[140] F. A. Tobari and L. Kleinrock, "Packet switching in radio channels: Part II- The hidden terminal problem in carrier sense multiple-access and the busy-tone solution," *IEEE Trans. Commun*., Vol. Com-23, N°12, pp. 1417-1433, December 1975.

[141] N. H. Vaidya, ''Weak duplicate address detection in mobile ad hoc networks,'' *Technical report*, University of Illinois at Urbana-Champaign, January 2002.

[142] G. Varghese, "Self-stabilization by computer flushing," *in Pro. 13th ACM Symposium On Principles of Distributed Computing*, August 1994.

[143] M. G.Velazquez ''A survey of distributed mutual exclusion algorithms,'' *Colorado state university - Technical report CS-93-116*, September 1993.

[144] S. Vasudevan, B. Decleene, N. Immerman, J. Kurose, and D. Towsley, "Leader election algorithms for wireless ad hoc networks", *In Proc. of IEEE DISCEX III*, 2003.

[145] Y. Wada and Z. Cheng, "An efficient method for allocating resources based on an unobstructed squeezing technique,'' *ACM SIGOPS Operating Systems Review*, Vol. 36, Issue 3, pp. 33 – 45, July 2002,

[146] E. J. Walter, G. Cao, and M. Mohanty, ''A token forwarding k-mutual exclusion algorithm for wireless ad hoc networks,'' *Computer Science Department, Texas A&M University, College* Station, TX 77840-3112.

[147] J.E. Walter, G. Cao, and M. Mohanty, ''A *k*- mutual exclusion algorithm for ad hoc wireless networks,'' *Proceedings of the First Annual Workshop on Principles of Mobile Computing (POMC 2001)*, August 2001.

[148] W. Wu, J. Cao, and J. Yang, '' A Scalable mutual exclusion algorithm for mobile ad hoc networks,'' *14th IEEE International Conference on Computer Communications and Networks (ICCCN'05)*, San Diego, California, USA, pp. 165-170, 17-19 October, 2005.

[149] J. Walter and S. Kini, "Mutual exclusion on multihop, mobile wireless networks," *Texas A&M Univ., College Station, TX 77843-3112*, TR97-014, Dec 9, 1997.

[150] J. Walter, J. Welch, and N. Vaidya, ''A mutual exclusion algorithm for ad hoc mobile networks,'' *Wireless Networks*, Vol. 9, No. 6, pp.585 - 600, Nov. 2001.

[151] K-P. Wu and Y.-J. Joung, '' Asynchronous group mutual exclusion in ring networks,'' *IEEE Proceedings- Computers and Digital Techniques*, Vol. 147, N°1, pp. 1-8, 2000.

[152] C-Z. Yang, ''A token-based *h*-out of-*k* distributed mutual exclusion algorithm for mobile ad hoc networks,'' *3rd International Conference on Information Technology: Research and Education (ITRE 2005)*, pp. 73-77, 2005.

[153] J. Zander, ''Radio resource management in future wireless networks: Requirements and limitations,'' *IEEE Comms. Magazine*, pp. 30-36, Aug 1997.

[154] X. Zeng, R. Bagrodia, and M. Gerla, “Glomosim: A library for parallel simulation of large-scale wireless networks,” *Proc. of the 12th Workshop on Parallel and Distributed Simulations (PADS’98)*, May 1998.