N° d'ordre : 13/2006-M/IN

REPUBLIQUE ALGERIENNE DEMOCRATIQUE ET POPULAIRE
MINISTERE DE L'ENSEIGNEMENT SUPERIEUR ET DE LA RECHERCHE SCIENTIFIQUE
UNIVERSITE DES SCIENCES ET DE LA TECHNOLOGIE
« HOUARI BOUMEDIENE »

FACULTE D'ELECTRONIQUE ET INFORMATIQUE

MEMOIRE
Présenté pour l'obtention du diplôme de MAGISTER
EN : INFORMATIQUE
SPECIALITE : PROGRAMMATION ET SYSTEMES

Par
HANAFI NEE BOUKHA ZOUINA

Sujet

La K-exclusion mutuelle dans les réseaux mobiles ad hoc

Soutenu le 24 / 12 / 2006, devant le jury composé de :

Mr Badache Nadjib, Professeur, USTHB	**Président**
Mr Benchaiba Mahfoud, Chargé de cours, USTHB	**Directeur de thèse**
Mr Ahmed Nacer Mohamed, Professeur, USTHB	**Examinateur**
Mr Azoune Hamid, Maître de conférences, USTHB	**Examinateur**
Mr Belkhir Abdelkader, Maître de conférences, USTHB	**Examinateur**

DEDICACES

A ma mère . . .

A toute ma famille

A tous mes amis

REMERCIEMENTS

Je tiens à remercier Mr M. Benchaiba Pour m'avoir proposé ce sujet et dirigé le travail durant toute la durée de sa réalisation. Je le remercie pour Tous ses conseils fructueux et son aide précieuse qu'il m'a apporté et surtout pour sa patience et son indulgence.

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

Je remercie également toutes les personnes qui mon apporté leur aide (Saad, Tahar, Karima, . . .) notamment en ce qui concerne les services de bureautique.

SOMMAIRE

INTRODUCTION GENERALE

Depuis leur apparition en 1970, les réseaux sans fil sont devenus de plus en plus populaires. Ils fournissent aux utilisateurs la possibilité de communiquer indépendamment de leur localisation. Les réseaux sans fil classiques sont souvent connectés à un réseau statique, il s'agit des réseaux mobiles. Tous les postes mobiles communiquent entre eux à travers le réseau filaire. En parallèle, il existe un autre modèle de réseaux mobiles sans station de base fixe ni liaison filaire, ce type de réseaux s'appelle **réseau mobile ad hoc**. Dans certains environnements tels que les communications en champs de bataille, les opérations de sauvetage en cas de désastre les réseaux mobiles ad hoc sont très efficaces et fournissent un moyen idéal pour la communication et l'accès aux informations. Ils peuvent aussi jouer un rôle important dans les forums, conférences, l'enseignement à distance, etc...

Les réseaux mobiles ad hoc ne requièrent aucune infrastructure. Les équipements mobiles possèdent des capacités physiques (source d'énergie, bande passante, etc...) modestes. Les unités mobiles sont autonomes et peuvent se déplacer arbitrairement (dans une voiture, une navette, etc…). La communication est établie par des liens sans fil. Les déconnexions, le partitionnement du réseau ainsi que la formation de nouveaux liens sont la conséquence de la mobilité. En effet, les réseaux mobiles ad hoc sont caractérisés par le changement fréquent de la topologie.

Comme un système autonome basé sur la communication multi-sauts, un réseau mobile ad hoc possède ses propres mécanismes de routage. Les nouvelles contraintes introduites par l'environnement mobile ad hoc font qu'il est nécessaire de réviser les applications déjà conçues pour les systèmes distribués classiques. Les travaux existants dans ce domaine incluent plusieurs algorithmes tels que les protocoles de routage, les algorithmes d'élection (*leader election*), les protocoles de *braodcasting* et de *multicasting*, etc... mais peu de travaux sont effectués dans le cadre de l'allocation de ressources.

Comme dans les systèmes distribués, le problème d'allocation de ressources est posé dans l'environnement mobile ad hoc. En effet, les nœuds mobiles partagent des ressources communes. Les processus qui constituent le système distribué peuvent demander la même ressource en même temps. Si la ressource nécessite des accès exclusifs, un arbitrage est exigé afin d'allouer cette ressource équitablement. Ceci est le problème d'exclusion mutuelle distribuée. En parallèle, une ressource peut exister en plusieurs exemplaires dans le système, par exemple une base de données répliquée; l'exclusion mutuelle dans ce cas est en plus, d'éviter qu'un processus attende longtemps dans une file la libération d'une ressource alors qu'une copie de cette ressource est disponible. Il s'agit de la K-exclusion mutuelle; K représentant le nombre d'exemplaire d'une même ressource.

Dans les systèmes distribués, les algorithmes existants se distinguent par la stratégie adoptée pour la résolution du problème d'exclusion mutuelle et de K-exclusion mutuelle. On distingue d'une part les algorithmes basés sur la permission où l'autorisation d'entrer en section critique est décernée par un consensus général ou partiel; d'autre part les algorithmes basés sur le jeton qui utilisent celui-ci comme un arbitre, c'est-à-dire que la détention du jeton est considérée comme un privilège pour accéder à la ressource partagée.

Dans le but d'améliorer les performances des algorithmes d'exclusion mutuelle et de K-exclusion mutuelle, certaines solutions proposent une organisation du réseau selon une

topologie appropriée par exemple en arbre ou en anneau. L'introduction de la notion de coterie (ensemble de quorums) a été d'un apport qualitatif quant à l'amélioration du temps de réponse et de la résistance aux pannes des algorithmes. Cette dernière notion a été généralisée pour donner naissance à une approche nouvelle et adaptée au problème de K-exclusion mutuelle. Cette approche, basée sur les K-coteries, s'avère prometteuse car elle permet de diminuer considérablement le nombre de messages générés par entrée en section critique.

Dans les réseaux mobiles ad hoc, peu de travaux sont effectués quant à l'exclusion mutuelle et notablement moins pour la K-exclusion mutuelle. Les algorithmes d'exclusion mutuelle dans les réseaux mobiles ad hoc s'inspirent des algorithmes déjà existants dans les systèmes distribués moyennant une adaptation au nouvel environnement caractérisé par l'absence d'infrastructures et de contrôle centralisé, la mobilité des nœuds et des capacités limitées des unités mobiles. Ce même environnement, nous pousse à penser que les solutions basées sur les coteries et K-coteries ne sont pas très appropriées (jusqu'à preuve du contraire) puisqu'elles nécessitent une interaction intense du système pour la construction et la maintenance de ces structures. En outre, l'utilisation des jetons est de rigueur car elle permet une diminution considérable des messages générés par rapport aux systèmes à permission. C'est dans cet esprit que nous avons abordé le problème de la '*K-exclusion mutruelle dans les réseaux mobiles ad hoc*' ayant pour objectif de concevoir un algorithme qui tente de suggérer des solutions aux problèmes posés au moindre coût.

L'aspect dynamique et aléatoire des réseaux mobiles ad hoc a motivé l'idée de notre algorithme intitulé '***Ksearch***'. Ksearch est orienté recherche de jeton, il ne suppose aucune topologie logique et se veut *modulaire*, *paramétré* et *interactif* par rapport à tous les évènements qui peuvent se produire. En effet, face à la même situation, la décision ou l'action n'est pas toujours la même: cela dépend des informations existantes à ce moment.

La technique de recherche du meilleur chemin dans une structure d'arbre est classique. Cependant, dans notre contexte il est nécessaire de l'étoffer avec un certain nombre de concepts et outils mathématiques et algorithmiques afin de palier aux effets de l'absence d'une structure d'arbre logique stable et permanente; d'où l'intérêt des heuristiques, des nombres premiers et de la gestion particulière des files d'attentes et des jetons.

Un nœud i demandeur d'accès, s'il ne possède pas de jetons libres, se trouve devant plusieurs contraintes: il ne connaît que ses voisins directs et n'a aucun chemin vers un nœud détenteur de jetons. En outre, il peut à tout moment perdre ses liens avec ses voisins et nouer avec de nouveaux nœuds. C'est ainsi que tout le système est appelé à coopérer afin d'offrir un mécanisme efficace pour acheminer les requêtes: les heuristiques semblent être en ce sens un bon recours. En effet, le noeud i envoie sa requête sur la voie qui a '*le plus récemment*' acheminé un éventuel jeton. Si elle n'existe pas, un chemin aléatoire sera construit. Ce chemin ne doit pas contenir de cycles. C'est pour cela que chaque nœud traversé par la requête est sauvegardé de manière distribuée. Le jeton une fois atteint doit suivre au plus, le chemin inverse de la requête.

Plusieurs optimisations sont proposées telles que l'optimisation du chemin de jeton, l'utilisation de jetons de type '*collector*', cession de jetons en cas de crainte d'isolement, introduction des files d'attentes temporaires, …etc.

Dans ce document, nous présentons en détail, le travail réalisé à la suite d'une étude bibliographique approfondie.

Ce document est organisé en deux parties délimitées par une introduction et une conclusion générales et à la fin une bibliographie et deux annexes:

- La première partie concerne l'état de l'art et consiste à étudier le problème de la K-exclusion mutuelle dans les réseaux mobiles ad hoc en passant par l'étude de l'exclusion mutuelle et la K exclusion mutuelle dans les systèmes distribués ainsi que l'exclusion mutuelle dans les réseaux mobiles ad hoc. Nous avons ainsi dégagé deux chapitres:

- Le premier aborde le problème de l'exclusion mutuelle et de la K-exclusion mutuelle dans les systèmes distribués en présentant les solutions déjà existantes dans ce contexte;
- Le deuxième reprend le problème de l'exclusion mutuelle et de la K-exclusion mutuelle, mais dans un environnement nouveau, celui des réseaux mobiles ad hoc;

- La deuxième partie intitulée 'Contribution' concerne la solution que nous proposons pour le problème de la K-exclusion mutuelle dans l'environnement mobile ad hoc. Elle est présentée en deux chapitres:
  - Le premier donne le détail de la conception de notre algorithme; À savoir, sa politique générale, les principes sur lesquels il est basé ainsi que l'ensemble des processus qui le composent en précisant les différentes procédures et variables utilisées avec une description détaillée des types de messages générés;
  - Le deuxième présente une étude de simulation: il est question d'évaluer les performances de notre algorithme dans un contexte prédéfini. Nous présentons dans ce chapitre le simulateur NS2 qui représente la plate forme de ce travail, les différents paramètres pris en compte dans les mesures effectuées, les critères de performances évaluées ainsi que les différents résultats obtenus avec une interprétation qui met l'accent sur le déroulement de l'algorithme et tente d'expliquer certains résultats par rapport aux conditions de simulation et au journal des évènements généré par le programme.

A la fin du document deux annexes sont présentées: la première, donne les préliminaires sur les caractéristiques des réseaux mobiles ad hoc et la deuxième dresse un ensemble de tableaux dans lesquels sont détaillés tous les résultats obtenus des différentes simulations effectuées.

Chapitre 1 La K-exclusion mutuelle dans les systèmes distribués

A. L'exclusion mutuelle dans les systèmes distribués

I. Introduction

L'apparition des réseaux d'ordinateurs a contribué à l'émergence de l'informatique dite distribuée. La maîtrise des systèmes distribués et des applications conçues sur les réseaux d'ordinateurs passe par la connaissance des techniques, d'outils et d'algorithmes qui constituent ce que l'on appelle algorithmique distribuée. En effet, les concepteurs de logiciels de base des systèmes distribués sont confrontés à une nouvelle problématique de contrôle distribué: offrir à un utilisateur la possibilité de composer des services existants sur des calculateurs distants, tout en garantissant que cela ne perturbe pas la disponibilité de ces services ainsi que la cohérence globale du système.

Plus formellement, un système distribué est une collection de sites autonomes appelés ***nœuds*** qui communiquent entre eux par échange de messages à travers des canaux de communication. Ils ne disposent ni de mémoire ni d'horloge communes. Chaque site a une connaissance partielle ou incomplète de l'état de l'ensemble du réseau. La distribution des programmes à travers ces sites permet au processus de s'exécuter de manière concurrente, de partager des ressources mais aussi de fonctionner indépendamment.

Ainsi, les processus d'un système distribué peuvent entrer en compétition pour l'accès à des ressources dites partagées, par exemple un serveur, un fichier, un périphérique; ou bien se communiquer des informations de façon à coopérer à la réalisation d'un but commun. La résolution des problèmes engendrés par la compétition et la coopération des processus passe par la définition des concepts respectivement de synchronisation et de communication.

Un des problèmes dus à la compétition est celui connu sous le nom de problème d'exclusion mutuelle. Chaque processus possède un segment de code appelé section critique dans lequel il peut accéder à une ressource. Les algorithmes distribués d'exclusion mutuelle doivent assurer le partage de ressources par plusieurs processus dans le cas où une ressource est allouée à un seul processus à la fois. Ils doivent synchroniser les accès aux ressources partagées de manière équitable, sans famine (starvation free) et sans verrouillage mortel (deadlock free). En d'autres termes, un algorithme d'exclusion mutuelle doit posséder les deux propriétés suivantes:

- ✓ ***La sûreté***: qui garanti l'exclusion mutuelle, c'est à dire qu'à tout moment au plus un seul processus exécute sa section critique;
- ✓ ***La vivacité***: qui assure que la section critique est toujours accessible par un processus demandeur au bout d'un temps fini; autrement dit, une situation d'interblocage ne se produit jamais. En outre, toute exécution de la section critique se déroule en un temps fini. Ceci signifie qu'aucun processus demandeur de la section critique n'attend indéfiniment la libération de la ressource sollicitée; d'où l'absence de situation de famine.

Deux approches peuvent être utilisées pour l'implémentation des mécanismes d'exclusion mutuelle dans les systèmes distribués:

- L'approche centralisée: un nœud particulier assure la fonction de coordinateur, toutes les requêtes lui sont destinées. Il doit avoir toutes les informations sur le système et octroyer au demandeur la permission d'accéder à la ressource partagée;
- L'approche distribuée: la décision est distribuée à travers tout le système et la solution au problème d'exclusion mutuelle est plus compliquée à cause de la difficulté d'obtenir une connaissance complète du système.

Dans cette section, l'intérêt est essentiellement porté sur les algorithmes distribués pour leur forte mise en œuvre des concepts de coopération et de compétition dont nous avons besoin le long de

cette étude. Dans un premier temps une classification générale des algorithmes d'exclusion mutuelle sera présentée, ensuite quelques solutions seront étudiées dans le détail en mettant en évidence leurs caractéristiques et les mécanismes et outils adoptés pour la résolution du problème posé. Ces solutions seront accompagnées d'une étude critique qui mettra en valeur les performances obtenues en terme de nombre de messages générés par entrée en section critique. Les mécanismes de tolérance aux pannes seront étudiés afin de déterminer les moyens de détection des pannes et les solutions qui permettent au système de revenir à son état stable. Cette section sera terminée par une conclusion.

II. Classification des algorithmes d'exclusion mutuelle distribués

L'étude détaillée que nous avons effectuée sur un grand nombre d'algorithmes d'exclusion mutuelle a permis de classer ces derniers selon un critère principal à savoir '*L'argument nécessaire et suffisant pour qu'un processus demandeur exécute sa section critique*'. Dans une première classe, l'argument est l'obtention d'un jeton; un message spécial qui circule à travers tous les nœuds du système passant le droit d'accès à la ressource critique au nœud qui le détient 'le nœud privilégié'. Il s'agit ici de l'approche ***jeton***. La circulation du jeton n'est pas aléatoire, elle est régie par une politique précise. La présence d'un jeton unique dans le système assure l'exclusion mutuelle. Cependant, l'approche à jeton est susceptible à la perte de ce dernier.

Dans la deuxième classe, le droit d'entrer en section critique est formalisé par la réception d'une permission de la part d'un ensemble de nœuds dans le système. Il s'agit de l'approche à ***permission***: un processus n'entre en section critique que s'il reçoit la permission de tous les nœuds de l'ensemble. Ceci assure l'exclusion mutuelle sachant que 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 section critique; d'où l'introduction de la notion de quorums.

Giford [21] fut le premier à introduire cette notion. Garcia, Molina et Barbara [22] ont introduit la notion de ***coterie*** définie comme un ensemble non nul de quorums dans lequel deux quorums quelconques doivent avoir un nœud au moins en commun (*propriété d'intersection*) et aucun quorum n'est un sous ensemble d'un autre (*propriété de minimalité*). Plusieurs protocoles basés sur les quorums ont été développés. La solution a un impact direct sur le coût de communication en terme de nombre de messages échangés pour chaque requête invoquée.

D'autres algorithmes ont imposé une structure logique construite sur la topologie physique du réseau [5][6][8][10][13][33][3[illegible].

L'étude des algorithmes basés sur la permission a permis de dégager une approche basée sur la *connaissance locale de l'état du système*. Autrement dit, un processus demandeur prend sa décision d'entrer en section critique localement en se basant sur un ensemble d'informations qu'il collecte et qui représentent une connaissance de l'état du système et de son propre état par rapport à celui-ci.

Les algorithmes récents ont introduit des mécanismes de tolérance aux fautes afin de détecter et recouvrer les pannes qui surviennent.

Nous présentons dans la suite notre classification et, pour chaque classe, nous détaillons le principe général, les mécanismes et outils utilisés.

II.1. Algorithmes basés sur le jeton

Un jeton est un message spécial circulant à travers les nœuds du système en passant le droit d'accès à la section critique au processus qui le détient. En effet, les processus demandant d'entrer en section critique ne peuvent y accéder que s'ils ont acquis le privilège matérialisé par le message de jeton. Le jeton est donc chargé de contrôler l'accès d'un seul processus parmi tous les autres (processus du système) à une ressource critique. L'unicité du jeton assure l'exclusion mutuelle (propriété de sûreté). L'étude que nous avons effectuée sur les algorithmes distribués d'exclusion mutuelle basés jeton nous a conduit à proposer une classification de ces algorithmes en rassemblant dans une même classe les algorithmes dont le principe général est similaire.

II.1.1. Algorithmes basés sur la diffusion

Principe général

Les algorithmes de cette approche ont recours à la diffusion totale où l'information émise, autrement dit la requête d'entrer en section critique par un processus demandeur, est diffusée à tous les autres processus du système. Nous illustrons cette approche grâce à l'algorithme de Suzuki-Kazami [1] qui en est un modèle.

Algorithme de Suzuki-Kazami

Le principe de base de cet algorithme [1] est de transférer le privilège d'entrer en section critique en utilisant un message unique représentant ce privilège d'accès. Initialement, le nœud 1 possède le privilège. Un nœud désirant obtenir le privilège envoie un message de requête à tous les autres nœuds. Le nœud recevant le message privilège est autorisé à entrer dans sa section critique autant de fois qu'il le désire et aussi longtemps que le privilège ne lui a pas été sollicité.

Un message de requête en provenance d'un nœud j a la forme *REQUEST*(j,n) où j est l'identificateur du nœud et n (n=1,2,…) est le numéro de séquence indiquant que le nœud j est entrain de faire sa $(n+1)^{ième}$ invocation de la section critique. Chaque nœud manipule un tableau *RN* de taille N (N étant le nombre de nœuds dans le système) pour enregistrer le plus grand numéro de séquence reçu de chaque nœud dans la case correspondante. Lorsque le message *REQUEST*(j,n) arrive à un nœud i, celui-ci calcule *RN*[j]:= max(*RN*[j],n). Le nœud i utilise *RN*[i] pour générer son propre numéro de séquence.

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

L'algorithme de Suzuki-Kazami [1] nécessite N messages pour chaque entrée en section critique ou 0 messages si le nœud demandeur est détenteur du jeton. La communication est supposée fiable; les délais de transfert sont finis mais imprévisibles et la préservation de l'ordre des messages n'est pas requise.

Algorithmes suivant le même principe

Le même principe a été retenu dans l'algorithme proposé par Ricart-Agrawala [2] qui dans les mêmes conditions, offre les mêmes performances en terme de nombre de messages. Cependant, la taille du message de jeton est plus petite vu sa forme: *TOKEN*(token_data) où *token_data* est un tableau de taille N tel que *token_data*[j] contient le numéro de séquence de la dernière requête satisfaite du nœud j. Outre ce tableau, le tableau *request_data* de taille N est maintenu au niveau de chaque processus et contient le numéro de séquence de la dernière requête reçue de chaque nœud du système. Ainsi, en utilisant ces deux tableaux, il est facile de se passer de l'utilisation d'une file de requêtes.

La technique de diffusion peut différer d'un algorithme à un autre. En effet, Dans l'algorithme de Helary-Plouzeau-Raynal [7], le nœud désirant entrer en section critique envoie sa requête seulement à ses voisins directs et attend l'arrivée du jeton. Le message de requête contient l'identification du nœud originaire, une estampille basée sur l'horloge logique de Lamport[23], l'identification du nœud ayant transmis la requête et les nœuds auxquels la requête a été adressée. Un nœud recevant une requête connaît alors son origine ou son émetteur et détermine les nœuds auxquels

elle n'a pas encore été envoyée et auxquels il va adresser la même requête (propagation). Le chemin emprunté par la requête est ainsi facile à retracer. Lorsque le nœud détenteur du jeton sort de sa section critique, il cherche dans sa file de requêtes la plus ancienne, lui attribue une nouvelle estampille selon l'heure logique locale et envoie le jeton qui va emprunter le chemin inverse de la requête elle même. Si aucune requête n'est présente, le nœud garde le jeton. Aucune topologie n'est imposée par cet algorithme, la seule connaissance nécessaire pour un nœud est celle des noms de ses voisins directs. Le nombre de messages nécessaires pour une entrée en section critique dépend de la topologie physique du réseau: dans une topologie linéaire, le nombre varie de N à 2(N-1) messages; dans une topologie en anneau, il varie entre N et 2N; Si c'est un maillage complet, le nombre de messages nécessaires est N.

Singhal, dans son algorithme [14] propose une diffusion de la requête basée sur une connaissance d'état. En effet chaque site maintient des informations sur l'état des autres sites et détermine quels sites seront les destinataires de la requête. Ce sont tous les sites qui se trouvent dans l'état 'Requesting the critical section'. En outre, il a été tenu compte des effets de la perte de messages (suite à une panne de nœud ou de lien) sur le fonctionnement de l'algorithme ainsi une procédure de recouvrement a été présentée.

Mishra-Srimani [3] et Nishio-Li-Manning [4] ont étendu l'algorithme de Suzuki-Kazami [1] en introduisant un mécanisme de tolérance aux pannes afin de recouvrir les cas de panne d'un nœud ou d'un lien qui peuvent avoir comme conséquence la perte de jeton. Ces algorithmes [3][4] assurent la détection de la perte de jeton et la régénération de celui-ci avec élimination des jetons dupliqués si le cas se présente.

II.1.2. Algorithmes basés sur une structure logique

Principe général

Les processus du système sont structurés selon une topologie logique construite sur la topologie physique du réseau. Plusieurs algorithmes [5][6][13][33][38] adoptent cette approche; Nous prendrons comme modèle l'algorithme de Raymond [5] basé sur une structure d'arbre.

Algorithme de K.Raymond

Dans cet algorithme [5], les nœuds sont organisés en structure d'arbre non orienté comme indiqué dans la *figure 1*. Tous les messages utilisés sont envoyés le long des liens de cet arbre. Celui-ci étant une structure logique construite en utilisant le minimum de liens physiques permettant de l'obtenir.

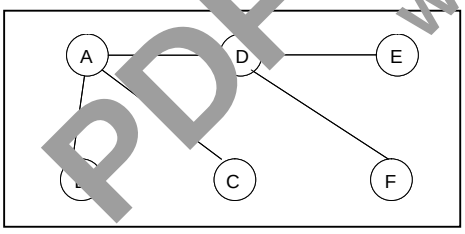


Figure 1. Les nœuds organisés en arbre

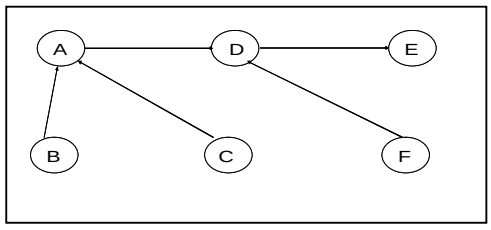


Figure 2 Arbre avec les liens orientés vers le nœud E

Chaque nœud possède une information suffisante sur ses nœuds voisins. Par exemple, dans la *figure 1*, le nœud A connaît l'existence de ses voisins B, C et D mais n'a aucune 'idée' de l'existence des nœuds E et F.

Le droit d'accès à la section critique est obtenu grâce à la possession d'un jeton (ou privilège). Ce privilège est soit détenu par un nœud (le nœud privilégié) soit en transit vers un nœud demandeur. Lorsque aucun nœud n'est demandeur, le privilège reste chez le dernier nœud l'ayant obtenu.

Chaque processus est doté d'une variable *HOLDER* qui indique la localisation du nœud privilégié vu du nœud lui-même. Par exemple, si dans *la figure 1* le nœud E détient le jeton alors voici les valeurs de la variable *HOLDER* au niveau de chaque processus:

$HOLDER_A$=D, $HOLDER_B$=A, $HOLDER_C$=A, $HOLDER_D$=E, $HOLDER_E$=self, $HOLDER_F$=D

Si nous avons à représenter l'information $HOLDER_X$ = Y comme une flèche du nœud X vers le nœud Y, nous obtenons un graphe orienté vers le nœud privilégié comme l'indique la *figure 2*.

Lorsqu'un nœud désire entrer en section critique, il envoie sa requête au nœud indiqué dans sa variable *HOLDER*. Ce nœud retransmet la requête au nœud indiqué dans la variable *HOLDER* et ainsi, de proche en proche la requête atteindra le nœud privilégié. Ce dernier choisi parmi ses voisins (qui ont transmis une requête) celui auquel il va répondre (le premier dans sa file) et lui envoie le jeton en mettant à jour sa variable *HOLDER*. Ainsi, de proche en proche, le chemin vers le jeton sera reconstruit à nouveau jusqu'au nœud destinataire qui sera autorisé à exécuter sa section critique et dont la variable *HOLDER* est mise à jour avec la valeur *self*. Si la file de requêtes de l'émetteur du jeton n'est pas vide, celui-ci envoie une requête au nœud privilégié passant par le nœud indiqué dans la variable *HOLDER* comme c'est décrit précédemment.

Chaque processus utilise localement les variables suivantes:

HOLDER: La valeur de *HOLDER* indique le nœud par lequel passe le chemin vers le jeton. Si HOLDER=self , ceci signifie que le processus concerné représente le nœud privilégié et doit mette à jour cette valeur en cas de transmission du jeton;

USING: Un booléen indiquant si le processus est entrain d'exécuter sa section critique;

REQUEST_Q: Une file de requêtes gérée en FIFO. Les seules valeurs possibles pour *REQUEST_Q* sont les identificateurs des nœuds voisins ayant émis une requête mais pas encore satisfaits, ou encore *self* si le processus est lui-même demandeur. La taille maximum de cette file est donc égale au nombre de voisins + 1.

ASKED: Un booléen indiquant si le nœud a déjà émis une requête. Si *ASKED = true* aucune requête n'est émise du nœud. Ceci signifie que dès que le privilège est obtenu, *ASKED = false*. Ce mécanisme permet de prévoir l'envoi de messages de requêtes inutilement et assure que la file ne contient pas des entrées dupliquées.

La condition nécessaire pour qu'un nœud privilégié envoie le jeton est la suivante:

HOLDER = self ^ not USING ^ *REQUEST_Q* ≠ empty ^ ***Head***(*REQUEST_Q*) ≠*self*

Cet algorithme [5] suppose fiable la communication sur le réseau, il requiert en moyenne un nombre de messages O(log(N)) pour chaque entrée en section critique. En outre, il assure la reprise d'un nœud après sa panne en restaurant son état à travers des informations qu'il collecte de ses voisins.

Algorithmes suivant le même principe

D'autres algorithmes sont basés sur le même principe en construisant leur solution sur des topologies telles que la topologie de DAG (Direct Acyclic Graph) proposée par Chang-Singhal-Liu [33], D.M.Dhamdhere-S.S.Kulkarni [38] et Neilsen-Mizuno [6] et la topologie d'anneau proposée par Lelann [19].

Chang-Singhal-Liu [33] ont commenté l'algorithme de Raymond dont la structure simple des données dans le jeton n'oblige pas le transfert de la file et la structure d'arbre imposée sur le réseau diminue le trafic des messages par rapport à un réseau complet. Cependant, dans le cas d'une rupture de liens, il n'offre pas d'alternative; ce qui risque de provoquer une situation de famine; C'est ainsi que Chang-Singhal-Liu [33] proposent une structure de DAG qui, elle, permet l'alternative.

De même que précédemment D.M.Dhamdhere-S.S.Kulkarni [38] ont proposé d'améliorer le protocole de Raymond [5] en assurant une capacité de tolérance de K pannes dans le système à condition que chaque nœud ait au moins un voisin; la structure du réseau adoptée est un DAG.

C'est dans cet esprit que Neilsen-Mizuno [6] ont proposé leur algorithme basé aussi sur une structure de DAG. Dans cet algorithme, grâce à la structure de DAG présentée, le nombre de messages nécessaires pour une entrée en section critique est réduit de moitié: physiquement les nœuds sont complètement liés par des liens fiables; Logiquement, ils sont organisés en une structure de DAG selon laquelle un seul nœud représente la racine et chaque nœud n'est doté que d'un seul lien sortant. Trois variables sont maintenues au niveau de chaque processus *LAST*, *NEXT* et *HOLDING*. La

variable *LAST* représente au niveau de chaque nœud l'identification du dernier nœud lui ayant émis une requête, la variable *NEXT* indique le prochain nœud à recevoir le privilège; Si le nœud en est actuellement le dernier (*LAST* = 0), Sa variable *NEXT* est mise à 0. On peut ainsi retracer la file d'attente depuis le nœud privilégié jusqu'au nœud dont la variable *NEXT* vaut 0. Initialement, *NEXT* = 0, *LAST* égale l'identificateur du nœud privilégié. Voici le mécanisme d'allocation de ressources en exclusion mutuelle:

- Lorsqu'un nœud désire entrer en section critique, il envoie une requête au nœud indiqué par sa variable *LAST* et met sa variable *LAST* à 0; il devient ainsi le nœud racine;
- Lorsqu'un nœud intermédiaire (non racine) reçoit une requête d'un nœud X il envoie celle-ci au nœud indiqué par sa variable *LAST* et met *LAST* = X;
- Lorsqu'un nœud racine (c'est-à-dire *LAST* = 0) reçoit une requête d'un nœud X, il met sa variable *NEXT* à X. Notons que si le nœud détient le privilège et n'est pas en section critique c'est à dire *Holding* = true, il envoie immédiatement le jeton au nœud indiqué par sa variable *NEXT* sinon il le garde.

Grâce à ces variables, les nœuds se passent de la gestion d'une file de requête ce qui allège considérablement l'algorithme. Ce même principe a été retenu par l'algorithme de Naimi-Trehel-Arnold dans [8].

Hélary-Mostefaoui [10] ont eux aussi proposé un algorithme basé sur une structure logique. Cette structure est appelée 'hyper cube ouvert', elle est originale et possède de remarquable propriétés de stabilité et de localité grâce auxquelles l'algorithme proposé présente de bonnes performances et une grande résistance aux défaillances de sites: la complexité maximum en nombre de messages par requête est $\log_2 N+1$, et $O(\log_2 N)$ messages supplémentaires en moyenne sont nécessaires pour traiter chaque panne de site.

L'algorithme de Lelann [13] qui fut l'un des premiers algorithmes d'exclusion mutuelle, proposa une structure statique d'anneau permettant de faire circuler continuellement un jeton qui dans le cas de son passage par un nœud demandeur, il lui permettra d'entrer en section critique.

II.1.3. Algorithmes basés sur la structure de groupe

Principe général

Le principe de cette approche est de structurer les nœuds du système en quorums dans lesquels un seul jeton peut circuler. Le message de requête n'est envoyé par le nœud demandeur qu'aux nœuds appartenant à son groupe (quorum); ceci permet de réduire considérablement le nombre de messages échangés.

Concept de coterie et quorum

Le concept de coterie a été proposé par Garcia-molina et Barbara dans [22], il permet de diminuer le nombre de messages pour chaque entrée en section critique et augmente la probabilité qu'au moins un processus puisse utiliser une ressource malgré la présence de pannes au niveau des processus ou des liens (availability).

***Définition*:** Soit U l'ensemble de tous les processus. L'ensemble $C=\{Q1,Q2,\ldots,Qm\} \neq \phi$ est une coterie si et seulement si les conditions suivantes sont vérifiées:

1-*pas d'ensemble vide* : pour tout i, i=1..m, $Qi \neq \phi$;

2-*propriété d'intersection* : pour tout i,j, $Qi \cap Qj \neq \phi$;

3-*minimalité* : pour tout i,j ($i \neq j$), $Qi \not\subset Qj$ et $Qi \neq Qj$.

Les éléments d'une coterie sont appelés quorums.

Le concept de coterie et de quorums est utilisé d'une multitude de manières. L'exclusion mutuelle est assurée par le fait que deux quorums distincts ne sont jamais disjoints; les nœuds d'intersection entre deux quorums servent d'arbitres pour satisfaire les différentes invocations.

Algorithme de Mizuno-Neilsen-Rao

Cet algorithme [12] est une généralisation de l'algorithme de Suzuki-Kasami [1] et utilise le mécanisme proposé par Nishio-Li-Manning [4] pour recouvrir le cas de panne et régénérer le jeton moyennant quelques modifications. Il utilise une structure similaire aux coteries appelée *quorum agreements*. Un quorum agreement est représenté par une paire d'ensembles (Q, Q^{-1}) sous U = {1,2, …, N} ensemble de tous les nœuds du système et définie comme suit:
Une collection d'ensemble Q est un ensemble de quorums sous U si et seulement si :
1. $G \in Q$ implique $G \neq \phi$ et $G \subseteq U$;
2. (minimalité) : il n'existe pas d'ensembles $G, H \in Q$ tels que $G \subset H$

Soit $I_Q = \{H \subseteq U / G \cap H \neq \phi \quad \forall G \in Q\}$. L'*antiquorum* de l'ensemble Q dénoté par Q^{-1} est donné par

$Q^{-1} = \{H \in I_Q / H' \not\subset H \quad \forall H' \in I_Q\}$ Alors la paire $QA = (Q,Q^{-1})$ est appelée '*quorum agreement*' sous U.

Chaque nœud du système est membre d'au moins un sous ensemble de Q et plusieurs sous ensembles de Q^{-1}. Pour un nœud i, les quorums Ri de Q sont appelés ensemble de requêtes (*request set*) et ceux de Q^{-1} sont appelés ensemble d'acquisition (*acquired set*). Lorsqu'un nœud i désire entrer en section critique, il envoie sa requête à tous les autres membres de *Ri*, si un de ces membres détient le jeton le mécanisme de recherche de jeton s'arrête; autrement chaque membre j de *Ri* est aussi membre d'au moins un sous ensemble de *Aj*. Un nœud k dans un sous ensemble *Aj* détient le jeton et ce nœud n'est pas un membre de *Ri*. Ce nœud envoie un message *acquired* pour signifier aux autres nœuds de *Aj* qu'il vient d'obtenir le jeton. Dès qu'un nœud j de *Aj* reçoit le message *acquired* vérifie s'il n'a pas de requêtes en attente. Il confirme qu'il a reçu une requête du nœud i et envoie une requête de la forme *REQUEST*(RNj) vers le nœud k où *RNj* est un tableau de taille N contenant le plus grand numéro de séquence reçu du nœud j; Le nœud k détenteur du jeton met à jour le tableau RN_k comme suit $RN_k[i] := max(RN_j[i],RN_k[i])$ (pour tout i=1,N) et place la requête de i dans la file de requêtes dans un ordre croissant des identificateurs des nœuds demandeurs. Une fois que le nœud détenteur du jeton quitte sa section critique, il envoie le jeton au prochain nœud en tête de file, le message de jeton est de la forme *PRIVILEGE(TOKEN_Q,LN)* où *TOKEN_Q* est la file de requêtes et *LN* est un tableau de taille N tel que *LN[j]* est le numéro de séquence de la dernière requête satisfaite du processus j. Si *TOKEN_Q* est vide, le nœud k garde le jeton.

La performance de cet algorithme en terme de nombre de messages échangés pour une entrée en section critique dépend de la structure de quorums sous-djacente. En effet, le nombre de messages peut passer de:
-0 quand le nœud demandeur détient le jeton,
- à $((|Ri| -1) + (|Ai| -1) + 1)$ quand le jeton est dans l'ensemble Ri et
- $((|Ri| - 1) + |Ai| * (|Ai| - 1) + 1)$ quand le jeton n'est pas dans Ri.

II.2. Algorithmes basés sur la permission

Dans ces algorithmes, le processus demandeur attend d'obtenir la permission d'accès d'un ensemble de processus dans le système. Une fois obtenue, il est autorisé à entrer en section critique. Un seul processus à la fois peut obtenir le droit d'accès. Chaque nœud accorde sa permission à un seul nœud demandeur à la fois. Ceci assure l'exclusion mutuelle.

L'absence d'horloge globale dans un système distribué crée un problème crucial: l'ordonnancement des événements entre processus du système (l'envoi et la réception des messages échangés). Cette lacune a conduit certains auteurs à proposer des techniques logicielles telles qu'un mécanisme de séquencement ou d'estampillage. Lamport [23] fut le premier à introduire la notion d'horloge logique qui permet d'assurer un ordre total des requêtes dans le système.

Deux aspects doivent être considérés afin de réduire le nombre de messages échangés pour entrer en section critique :

- Le nombre suffisant de droits (permissions) à collecter ;

- Quels nœuds peuvent donner ce droit ?

Certains algorithmes [23][24][25] requièrent qu'un nœud demandeur doive obtenir la permission de tous les nœuds du système. Dans d'autres algorithmes, les nœuds sont divisés en ensembles non nuls et tels que deux ensembles quelconques ne sont pas disjoints. Ces ensembles sont appelés ***quorums***. Ainsi, chaque nœud demandeur doit obtenir la permission seulement des nœuds de l'ensemble auquel il appartient.

Suite à l'étude que nous avons réalisé sur les algorithmes d'exclusion mutuelle basés sur la permission, nous proposons, à la base des mécanismes et outils utilisés, la classification suivante:

II.2.1. Algorithmes basés sur un consensus

Principe général

Un processus demandeur désirant entrer en section critique doit en demander et obtenir la permission de tous les autres processus. Autrement dit, un processus demandeur ne pourra accéder à la ressource critique que s'il a reçu toutes les permissions attendues. Plusieurs algorithmes peuvent être classés dans cette approche; nous citons l'algorithme de Ricart-Agrawala [24] que nous détaillerons dans la suite, celui de carlvalho-Roucairol [25] qui a proposé une solution pour diminuer le nombre de nœud desquels la permission est demandée et, dans le même esprit celui de Sanders [27] et Bouabdallah-König [39]:

Algorithme de Ricart-Agrawala

Dans cet algorithme [24], un numéro de séquence est associé à chaque requête. Ce numéro est comparable à l'estampille de Lamport [23]. Chaque processus désirant entrer en section critique incrémente le plus haut numéro de séquence dont il a connaissance et l'associe à sa requête, celle-ci sera envoyée au N-1 nœuds du système. Le processus émetteur attend un message de retour (permission) de tous les N-1 nœuds; dès son arrivée il procède à l'exécuter sa section critique. Un nœud j recevant une requête d'un nœud i, retourne immédiatement un message d'accord dans le cas où il n'est pas lui même demandeur. Dans le cas contraire, il compare son propre numéro de séquence avec celui de la requête; s'il possède un numéro plus grand, il envoie son accord sinon il diffère cette opération jusqu'à ce que sa requête soit satisfaite auquel cas il envoie son accord à toutes les requêtes différées. Dans le cas d'égalité des numéros de séquence, le conflit est réglé en comparant les numéros des nœuds eux mêmes.

Cet algorithme maintient un ensemble de variables locales à chaque processus qui sont :
me: constante indiquant l'identification du nœud;
N: constante indiquant le nombre de nœuds dans le système;
Our_Sequence_Number: valeur générée au niveau du processus en incrémentant la valeur du plus grand numéro de séquence reçu;
Highest_Sequence_Number: le plus grand numéro de séquence vu du nœud, obtenu à la base du numéro de séquence de la dernière requête envoyée ou reçue;
Outstanding_Reply_Count: indique le nombre de messages de permissions non encore reçus;
Requesting_Critical_Section: un booléen qui indique si le noeud est demandeur de la resource critique;
Reply_Deferred: un tableau de booléens initialisé à false. *Reply_Deffered*[j] est mise à true si l'envoie de la permission au nœud j est différé.
Shared_Vars: un sémaphore utilisé localement avec les primitives P et V pour mettre à jour les variables locales (telles que Our_Sequence_Number, Requesting_Critical_Section) en exclusivité. Il est clair que ces variables sont partagées avec d'autres processus du même nœud.

Deux types de messages sont échangés :

- un message de requête ayant la forme *Request*(*Our_Sequence_Number*,*me*);
- un message de permission dont la forme est *Reply*.

Dans cet algorithme [24], le nombre de messages requis pour chaque entrée en section critique est de 2(N-1) c'est-à-dire que chaque nœud demandeur doit envoyer (N-1) messages de requêtes et recevoir (N-1) messages *Reply* ou permissions.

Algorithmes suivant le même principe

Calvalho et Roucairol dans [25], ont amélioré les performances de l'algorithme de Ricart-Agrawala [24]: Lorsqu'un nœud i reçoit une permission d'un autre nœud, il la retient jusqu'à ce qu'une autre requête soit reçue de ce même nœud. Une autre fois que le nœud i voudra entrer en section critique, il n'envoie sa requête qu'aux nœuds dont il n'a pas gardé la permission. Ceci permet de réduire le nombre de messages de requêtes à envoyer. En effet, ce nombre varie de 0 à 2(N-1) messages. A la réception d'une requête, le même fonctionnement que [24] est réalisé. En outre, le mécanisme de séquencement des requêtes est maintenu.

Sanders [27] propose pour la première fois dans son algorithme les structures d'information 'information structure' en vue de limiter l'ensemble de nœuds auxquels la permission d'entrer en section critique est demandée. Formellement, une structure d'information peut être déclarée par une paire de sous-ensembles de processus associée à chaque processus: Pour un processus i il a *Inform Set* dénoté par *Ii* et le *Request Set* dénoté par *Ri*. Un processus envoie un message à tous les processus de son *Inform Set* à chaque transition *IN-CS* (En section critique)→ *NOT-IN-CS* (en dehors de la section critique) et *NOT-IN-CS* → *WATING* (en attente de la libération de la ressource). Avant d'entrer en section critique chaque processus doit obtenir la permission de tous les éléments de l'ensemble *Request Set*.

A.Bouabdallah et J.C.König [39] ont proposé pour l'algorithme précédent [27] une structure d'informations basée sur la connaissance d'une valeur *D* représentant le diamètre du plus long chemin minimum entre deux processus x et y.

Singhal utilise dans son algorithme [29] le concept de structure d'information et, comme sanders [27] génère au niveau des chaque nœud i une structure d'information Ri (*Request Set*) et Ii (*Inform Set*) qui jouent le même rôle. La différence est dans l'aspect dynamique de *Ri* et *Ii*. En effet *Ri* est continuellement modifié en ajoutant les nœuds auxquels la permission a été envoyée et en supprimant les nœuds desquels la permission est reçue. L'ensemble *Ii* est modifié en lui ajoutant les nœuds desquels une requête est reçue tenant que le nœud i est soit en attente des permissions demandées soit en section critique. Le nœud auquel i envoie sa permission est supprimé de *Ii*.

II.2.2. Algorithmes basés sur une connaissance d'état

Principe général

Les algorithmes de cette approche s'appuient sur une connaissance locale à la base de laquelle un processus 'décide' d'entrer en section critique. Cette connaissance peut être obtenue de plusieurs manières par exemple en scrutant l'état du système par un message d'interrogation comme dans l'algorithme de Raynal [35] et Lamport [23].

Algorithme de Raynal[11]

Cet algorithme [35] exploite les propriétés des nombres premiers. A chaque nœud i est associé un attribut Ai, cet attribut est un nombres premier différents de 1 et maintient une variable Xi initialisée à Ai sauf pour le nœud k où $X_k = 1$. Q est le produit de toutes les valeurs Ai (i=1, …, N). Lorsqu'un nœud i désire entrer en section critique, il envoie un message de requête de la forme ***REQUEST*(i)** aux N-1 nœuds du système et attend le message de retour *REPLY*(j,Xj). Lorsque toutes les réponses sont arrivées, le nœud i calcule T qui est égale au produit de tous les Xj reçus. Si T=Q/Ai, i entre en section critique; autrement, il attend un moment et renvoie son message de requête à tous les autres nœuds.

Quant un nœud i quitte sa section critique, il recalcule Xi = Xi*Ai/Aj où j = (i+1) mod N. Ainsi, la permission d'entrer en section critique est attribuée de manière circulaire (en anneau).

Un nœud modifie sa variable Xi uniquement après avoir quitté sa section critique. L'effet de cette modification est que le nœud i perd la permission d'entrer en section critique pour permettre au prochain nœud de l'anneau d'obtenir cette permission.

En fait, le message de requête n'est autre qu'un message interrogeant tous les nœuds du système sur leurs attributs ce qui va permettre au nœud de prendre une décision vis-à-vis de son entrée en section critique. En outre, cette décision est basée sur la connaissance de toutes les réponses, ce qui fait que le système doit être sans perte de messages. L'algorithme souffre du problème d'interblocage dans le cas où le prochain nœud ne désire pas entrer en section critique; auquel cas sa variable Xi n'est jamais modifiée. Par conséquent, il ne transmettra jamais le privilège au suivant;

La complexité en nombre de messages est de $(N-1)^2$.

Algorithmes suivant le même principe

L'algorithme de Lamport [23] appartient aussi à cette famille. En effet, chaque processus demandeur ne peut entrer en section critique que si sa requête est la plus ancienne dans le système; cette connaissance est dû à deux facteurs: la file d'attente maintenue au niveau de chaque nœud et l'estampillage des requêtes grâce auquel tous les nœuds ont la même connaissance de l'ordre de ces dernières. Un processus demandeur envoie une requête à tous les N-1 nœuds. Un processus recevant un message de requête envoie un accusé *Reply* estampillé (selon les règles énoncées dans le paragraphe) à l'émetteur de ce message. Lorsqu'un processus sort de sa section critique, il envoie un message de type '*release*' à tous les autres processus pour notifier que sa requête a été satisfaite. Chaque processus dans le système recevant ce type de message modifie sa file et vérifie s'il n'est pas (en fonction des estampilles) le prochain à entrer en section critique auquel cas, il satisfait sa requête. Chaque entrée en section critique nécessite 3(N-1) messages, ce qui représente déjà une bonne performance pour un des premiers algorithmes d'exclusion mutuelle.

L. Lamport fût le premier à proposer un mécanisme d'ordonnancement des évènements basé sur une horloge logique. Ainsi, une estampille est associée à chaque requête selon les règles suivantes:

- Chaque processus incrémente son horloge logique entre deux évènements successifs;
- Si a est l'événement d'émission d'un message par un processus Pi et b est l'événement de réception du même message par un processus Pj alors Ci(a)<Cj(b) où Ci(a) est l'estampille associée à l'événement a par le processus Pi et Cj(b) est l'estampille associée à b par le processus Pj.

II.2.3. Algorithmes basés sur la structure de groupe

Principe général

Etant donnée une coterie, l'exclusion mutuelle peut être assurée de sorte qu'un nœud demandeur d'accès à une ressource critique ne peut y accéder que s'il a reçu la permission de tous les nœuds d'au moins un quorum appartenant à cette coterie. Plusieurs algorithmes [34][28] sont proposés et se différencient en général par la manière de construire ces coteries. L'algorithme de Maekawa [26] est l'un des premiers à avoir adopté cette approche.

Algorithme de Maekawa

Dans cet algorithme [26], les nœuds sont regroupés dans des sous ensembles S tels que les propriétés suivantes sont satisfaites:

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

b- soit le sous ensemble S_i, $1 \le i \le N$, S_i contient toujours i;

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

d- $\forall$ j, $1 \le j \le N$, j est contenu dans D sous ensembles S_i, $1 \le i \le N$.

✓ La propriété a assure l'existence d'au moins un nœud commun entre deus sous ensembles quelconques. Ce nœud servira d'*arbitre*;

- ✓ La propriété c implique que chaque nœud doit envoyer et recevoir le même nombre de messages pour accéder à la section critique;
- ✓ Les propriétés c et d assurent que chaque nœud assume les mêmes responsabilités vis à vis de l'exclusion mutuelle;
- ✓ La propriété b est ajoutée pour réduire le nombre de messages de requête envoyés. En effet, si le nœud i est membre de son propre sous ensemble (quorum), il n'aura pas à émettre une requête vers lui-même s'il est demandeur.

Le principe de l'algorithme est le suivant:
Lorsqu'un nœud i désire entrer en section critique, il tente de verrouiller tous les membres de son groupe S_i en envoyant à chacun un message *Request*. S'il réussit, alors il exécute sa section critique sinon, il attend que les membres déjà verrouillés soient libérés pour les capturer et les verrouiller.

Un nœud ne peut attribuer sa permission qu'à un nœud à la fois. A la réception d'une requête d'un nœud i, chaque nœud K vérifie s'il n'a pas encore donné sa permission (non verrouillé) ou s'il n'existe pas une requête ayant un numéro de séquence plus petit. Dans ce cas, le nœud K envoie sa permission au nœud i (donc devient verrouillé) à travers un message *Inquire*, autrement il place la requête en queue de file. Si le nœud K possède dans sa file une requête ayant un numéro de séquence plus petit que celui du nœud i, il envoie à i un message *Failed*.

Quand un nœud ayant déjà émis une requête aux membres de son groupe reçoit au moins un message *Failed*, il retourne un message *Relinquish* à tous les nœuds qui lui ont répondu par *Inquire* pour signifier qu'il n'a pas réussi à verrouiller tous les nœuds de son groupe et les libérer; Cette réaction permet d'éviter des situations d'interblocage. Si le nœud a réussi à verrouiller tous les nœuds de son groupe, il leur envoie un message *Release* après avoir exécuté sa section critique, ce qui permettra de les déverrouiller.

Algorithmes suivant le même principe

Dans l'algorithme de Maekawa, les propriétés a, b, c et d énoncées précédemment doivent être vérifiées dans la construction des groupes afin d'assurer l'exclusion mutuelle, dans l'algorithme de Bouabdallah-König [34] la propriété suivante doit en plus être satisfaite:
$\forall$ i,j $1 \le i,j \le N$, $|S_i \cap S_j| \ge 2*t$ ce qui assure une tolérance à la panne de plus de t-1 nœuds sans exécuter aucune procédure de recouvrement.

Agrawal-El abbadi [28] ont utilisé le même principe que Maekawa pour la résolution du problème d'allocation de ressources en exclusion mutuelle. La différence réside dans la manière de construire les groupes (les ensembles quorums) de nœuds. En effet, dans [28] une structure d'arbre binaire logique est proposée. Un mécanisme de construction d'arbres de quorums est dérivé de la structure d'arbre. Une cotérie est constituée de tous les arbres de quorum générés à partir de l'arbre binaire. Deux quorums distincts ne sont pas disjoints et aucun quorum n'est inclus dans l'autre.

III. Mécanismes de tolérance aux pannes

Dans un système distribué, l'une des préoccupations les plus importantes est la tolérance aux pannes. Or, la plupart des algorithmes d'exclusion mutuelle [1][2][23][25] se focalisent sur la problématique même c'est-à-dire 'comment assurer l'exclusion mutuelle en évitant les problèmes de verrouillage mortel et de famine et en obtenant les meilleurs performances possibles'. Les hypothèses sur lesquelles s'appuient ces algorithmes (à savoir pas de panne de nœuds, les liens de communication sont fiables, …etc) ne reflètent pas la réalité. En effet, la nature des systèmes distribués fait que plusieurs pannes peuvent se produire:

- Panne du canal de communication;
- Panne d'un ou plusieurs hôtes;
- Partitionnement et reconfiguration du système.

Les conséquences de ces pannes peuvent être dangereuses voir même fatales selon l'état du système lorsque la panne survient. Par exemple, une panne de site ou de lien de communication peut causer une perte de messages ou d'informations au sein d'un site ou encore le partitionnement du

système. Dans certains cas, les liens de communication peuvent délivrer des messages comportant des erreurs ce qui peut fausser les décisions prises.

Il est difficile de concevoir des solutions de recouvrement capables de prendre en charge tous les cas de pannes qui peuvent se produire. Néanmoins, il est possible de recouvrer un ou plusieurs types de pannes comme c'est le cas dans [14][28][33][34][38].

Un système capable de tolérer n'importe quelle panne est appelé '*système auto stabilisant*' ou '*self stabilizing system*'. C'est un système qui converge sans aucun contrôle centralisé vers un état légitime (stable) partant d'un état arbitraire. Ce concept a été proposé par Dijkstra [37].

IV. Tableau récapitulatif des algorithmes d'exclusion mutuelle

Le tableau suivant présente une synthèse des algorithmes d'exclusion mutuelle présentés dans cette étude. Nous proposons de les énumérer par ordre chronologique de leur apparition ce qui permettra de suivre leur évolution. Outre la vivacité et la sûreté que doit assurer chaque algorithme, la problématique principale pour laquelle toutes ces approches ont été développées est de minimiser la complexité en terme de nombre de messages:

Algorithme	Année	Nombre total de messages	Approche	Observations
Lelann [13]	1977	[0,N]	Jeton	-utilise une structure logique d'anneau
Lamport [23]	1978	3*(N-1)	Permission	/
Ricart- Agrawala [24]	1981	2*(N-1)	Permission	/
Calvalho-Roucairol [25]	1983	De 0 à 2*(n-1)	Permission	/
Ricart-Agawala [2]	1983	N	Jeton	/
Suzuki-Kazami [1]	1985	N	Jeton	/
Maekawa [26]	1985	3*sqrt(N) à 5*sqrt(N)	Permission	-tolérant aux pannes de nœuds, -utilise les quorums
Sanders [27]	1987	| Ii- {i}| +2*(| Ri-{i}|)	Permission	-tolérant aux pannes de nœuds, -construction de structures d'informations statiques et dynamiques
Helary-Plouzeau-Raynal [7] Topologie en arbre Topologie linéaire Topologie en anneau Maillage complet	1988	 [N,(N-1+D)] [N,2(N-1)] [N,2N] N	Jeton	/
Raymond [5]	1989	O(log(N))	Jeton	-tolérant à la perte d'informations -utilise une structure logique d'arbre
Neilsen-Mizuno [6] Topologie linéaire Topologie en étoile	1989	D+1 N 3	Jeton	- utilise un DAG
Singhal [14]	1989	N	Jeton	-utilise des heuristiques pour trouver le jeton -tolérant à la perte d'information
Nishio-Li-Manning [4]	1989	N	Jeton	-recouvrement en cas de pannes
Raynal [35]	1989	$2*(N-1)^2$	Permission	-utilise les nombres premiers -utilise une structure logique en anneau
Mishra-srimani [3]	1990	L*N+(N-1)	Jeton	-recouvrement en cas de pannes
Chang-Singhal-Liu[33]	1990	O(2*logN)	Jeton	-recouvre les pannes de nœuds et de liens -utilise une structure de DAG
D.M.Dhamdhere-S.S.Kulkarni [38]	1991	[O(d.K),O(d.K^2+	Jeton	-tolérant à K pannes de liens -utilise une structure de DAG
Agrawala-El-abbadi [28]	1991	O(log(N))	Permission	-utilise la structure d'arbres de quorums -tolère jusqu'à N – [logN] pannes de nœuds
Singhal [29]	1992	[(N-	Permission	-utilisation de structures d'informations dynamiques
A.bouabdllah et J.CKönig [39]	199	[qrt(N)),O(sqrt( ).log$_\Delta$N)]	Permission	-tolérant aux pannes de nœuds -utilise un DAG
A.bouabdllah et J.CKönig [34]	~ 992	sqrt(tN))	Permission	-tolère jusqu'à t pannes -utilise les groupes
Mizuno-Neilsen-Rao [12]	1993	[((| Ri |) + (| Ai | -1) + 1), ((| Ri | -1) + 2 * (| Ai | - 1) + 1)]	Jeton	-régénère le jeton en cas ce perte -utilise le concept 'quorum agreement'
J.M.Hélary-A.Mostefaoui [1	1993	log N-1	Jeton	-utilise une structure logique dynamique appelée 'hyper cube ouvert' -traite les pannes de nœuds
Naimi-Trehel-Arn [8]	1996	≈Log(N)	Jeton	-utilise une structure logique dynamique d'arbre

Table 1. Tableau récapitulatif des algorithmes d'exclusion mutuelle

Où,
N est le nombre de nœuds dans le système
Δ est le nombre maximum de voisins d'un nœud.
D le plus long chemin minimum sur le réseau
Ri ensemble *REQUEST*
Ii ensemble *INFORM*
Ai ensemble d'acquisition
d longueur maximale d'un chemin dans le système
e nombre de liens établis sur le réseau
t nombre de nœud dont la panne est tolérée

V. Conclusion

Dans cette section, nous avons présenté les algorithmes d'exclusion mutuelle dans les systèmes distribués. Nous avons décrit leurs principales caractéristiques. Ces algorithmes ont été distingués par l'approche adoptée. En effet, deux classes principales d'algorithmes ont été présentés: les algorithmes basés ***jeton*** et les algorithmes basés ***permission.***

Dans le premier groupe, différents mécanismes ont été utilisés pour la localisation du jeton, sa circulation à travers les nœuds demandeurs et l'ordonnancement des évènements dans le système. Les algorithmes basés jeton sont susceptibles à la perte du jeton et sa duplication dans le cas de régénération de celui-ci.

Dans les algorithmes basés permission, la préoccupation principale était de trouver le nombre minimum de nœuds desquels un nœud demandeur devait attendre la permission d'entrer en section critique. Différentes structures sont utilisées afin de réduire le coût en terme de nombre de messages transférés.

Les algorithmes d'exclusion mutuelle dans les systèmes distribués doivent réduire au maximum le nombre de messages échangés et ceci peut être obtenu si la structure logique des nœuds est bien choisie.

Dans la section suivante, nous étudions la problématique de la K-exclusion mutuelle dans les systèmes distribués.

B. La K-exclusion mutuelle dans les systèmes distribués

I. Introduction

Dans le chapitre précédent, les algorithmes présentés traitaient le cas où il n'existe qu'une seule ressource dans le système. Ce modèle peut servir pour le contrôle d'accès à une base de données distribuée par exemple. Cependant, il peut exister plus d'une ressource identique dans un système distribué. Le problème d'arbitrage de k ressources identiques est appelé problème de k-exclusion mutuelle distribué. Ici, k processus peuvent exécuter leurs sections critiques à la fois. Par exemple, considérons un réseau local auquel plusieurs processus sont connectés. Si le mode d'accès aux canaux est de type CSMA/CD (Carrier Sense Multiple Access with Collision Detect), la performance du réseau diminue considérablement si les machines envoient fréquemment des messages. Afin d'éviter une telle situation, on peut appliquer la k-exclusion mutuelle. La bande passante du réseau peut être découpée en plusieurs intervalles où chaque intervalle sera considéré comme une ressource et le fragment de code dans lequel un processus envoie un grand nombre de données via le réseau est appelé section critique.

Un algorithme de k-exclusion mutuelle peut être construit à partir de k algorithmes d'exclusion mutuelle si k ressources identiques ont des identifications différentes et qu'un processus désirant cette ressource en choisi une parmi k et adresse sa requête pour y accéder. C'est une solution simple mais peut être coûteuse: Supposons que tous les processus requièrent la même ressource (la même identification), ils doivent attendre longtemps la libération de celle-ci malgré que d'autres ressources identiques peuvent être libres. C'est pour cette raison que plusieurs algorithmes de k-exclusion mutuelle distribuée ont été développés. Le premier fut celui de K.Raymond [50] qui est une simple extension de l'algorithme de Ricart-Agrawala [24]. Srimani-Reddy [51] ont généralisé l'algorithme de Suzuki-Kassami [1]. D'autres algorithmes [58][59] ont utilisé une topologie logique maintenue dynamiquement par le système afin de faciliter aux processus l'obtention du jeton. Plusieurs solutions utilisent les K-coteries [52][53][54][55][56] comme structure logique fondamentale permettant de réduire le nombre de messages échangés et d'obtenir un haut niveau de tolérance aux pannes.

Dans les algorithmes que nous avons étudié, nous distinguons deux approches principales: l'approche basée sur la ***permission*** [50][52] et l'approche basée sur le ***jeton*** [51].

II. Algorithme de K.Raymond

Dans cet algorithme [50], Raymond a étendu l'algorithme de Ricart et Agrawala [24] afin de permettre à K nœuds du système d'exécuter leurs sections critiques simultanément. Lorsqu'un nœud i désire entrer en section critique, il envoie un message de requête aux N-1 nœuds du système, le message contient le plus haut numéro de séquence du système connu par i. Le nœud i attend de recevoir la permission de N-K nœuds au minimum. Lorsque le message *reply* arrive (considéré comme une permission), le nœud i comptabilise le nombre de nœuds qui sont en dehors de leurs sections critiques. Si ce nombre est au minimum égal à N-K+1, le nœud i est libre d'entrer en section critique. Les messages *reply* qui ne sont pas encore parvenus peuvent arriver pendant que le nœud i exécute sa section critique ou après avoir libéré sa ressource.

Lorsqu'un nœud j reçoit une requête du nœud i, il met à jour son numéro de séquence. Si j est dans sa section critique ou s'il est lui même demandeur et sa requête est plus ancienne que celle qu'il a reçu, il retarde l'envoi de sa permission. Lorsque le nœud j sort de sa section critique, il envoie sa permission à tous les nœuds retardés.

Le nombre de messages échangés pour chaque entrée en section critique varie entre 2N-K-1 et 2(N-1) messages. Ce protocole suppose les liens de communication fiables, les délais de transfert finis et l'ordre des messages n'est pas requis.

III. Algorithme de Srimani-Reddy

Cet algorithme [51] est basé sur l'algorithme de Suzuki et Kazami [1] avec l'utilisation de K jetons afin de privilégier K nœuds simultanément. Chaque jeton contient des informations sur l'état du système (telles que la file des requêtes non satisfaites, …). Ces informations peuvent être mises à jour par le nœud qui reçoit le jeton. Chaque nœud maintient un tableau de taille N pour enregistrer le numéro de séquence de la dernière requête satisfaite de tous les nœuds. Ce tableau est mis à jour par les informations contenues dans le jeton et est utilisé pour mettre à jour la file de requêtes contenue dans chaque jeton. Ceci permet d'éviter d'envoyer un jeton supplémentaire à un nœud dont la requête est satisfaite. Chaque nœud garde la trace du nombre de jetons qu'il possède.

Lorsqu'un nœud désire entrer en section critique et ne détient pas un jeton, il incrémente son numéro de séquence, envoie un message de requête aux N-1 nœuds du système et attend l'arrivée du jeton. Plusieurs jetons peuvent arriver. Si le nœud i détient déjà un jeton, il est libre d'entrer en section critique, le nœud i décrémente alors le nombre de jetons en sa possession. Une fois qu'il quitte la section critique, il met à jour les informations du jeton qu'il vient d'utiliser concernant sa propre requête servie. Si ce jeton est muni d'une file non vide, ce dernier est envoyé au nœud dont la requête est en tête de file. Si la file est vide, le nœud i incrémente le nombre de jetons qu'il détient.

Lorsqu'une requête arrive au nœud j en provenance du nœud i, le nœud j met à jour tous ses jetons avec la nouvelle information concernant le dernier numéro de séquence en provenance du nœud i. Si j possède un jeton et n'est pas lui même demandeur car il possède un jeton en plus alors l'information dans le jeton est mise à jour et le jeton est envoyé au nœud i.

Ce protocole suppose les liens de communication fiables, les délais de transfert finis et l'ordre des messages n'est pas requis. La complexité en terme de nombre de messages générés par entrée en section critique est de 0 si le nœud détient le jeton au moment où il exprime son besoin d'entrer en section critique et d'au plus N+K-1 sinon.

IV. Algorithme de S.Bulgannawar&N.Hanchanal

Cet algorithme [58], suppose l'existence d'un réseau fortement connecté. Les nœuds sont numérotés de 1 à N et les jetons sont aussi numérotés de 1 à K. A l'état initial, le nœud t détient le jeton *t*. Chaque nœud maintient un tableau de K pointeurs dont chaque entrée correspond à un jeton et où chaque jeton définit une forêt; une forêt étant une collection d'arbres. A chaque nœud j, *pointer[t]* correspond à la forêt du jeton *t* et contient l'identité de son prédécesseur correspondant à la forêt du jeton *t*. Si *pointer[t]* = j, cela signifie que le nœud j détient le jeton *t* et en est la racine. A tout instant, Le nombre de liens sortant de chaque nœud par rapport à une forêt, doit être égal à 1, en parallèle le nombre de liens entrant est plus grand. Initialement, la forêt correspondante au jeton *t* est composée d'un seul arbre dont la racine est le nœud t. En général, le nœud en attente de recevoir le jeton *t* et le nœud qui détient celui-ci sont tout deux racines pour un arbre de la forêt du jeton t.

Lors de la demande d'accès à la section critique, si un nœud i détient au moins un jeton, il entre en section critique immédiatement. Si aucun jeton n'est disponible, une heuristique quelconque est utilisée pour déterminer quel jeton sera sollicité. Soit le jeton *t*. Le nœud i, envoie donc une requête à son prédécesseur sur l'arbre actuel de la forêt correspondante au jeton *t* indiqué par *pointer[t]*.

A la réception d'une requête du nœud y pour le jeton *t*, l'action à entreprendre par le nœud i dépend de son état:

Cas 1: Le nœud i ne détient aucun jeton et est demandeur d'accès: ici, l'action à entreprendre dépend du jeton sollicité par la requête:

Cas a: Si le jeton sollicité par la requête est le même que celui attendu par i, alors le nœud i insère le nœud y dans sa file d'attente '*node_queue*' qui est générée à cet effet;

Cas b: Le jeton sollicité est différent: Dans ce cas, une requête est formulée et envoyée au nœud désigné par *pointer[t]* et *pointer[t]* prend la valeur de y (inversement des liens).

Cas 2: Le nœud i ne possède pas de jeton et n'est pas intéressé par la section critique: La requête est envoyée vers le nœud indiqué par *pointer[t]* et *pointer[t]* := y;

Cas 3: Le nœud i possède un jeton *u* (qui peut être lui-même t ou un autre jeton) et i n'est pas en section critique (ce qui implique que la file '*token_queue*' correspondante au jeton *u* est vide): Dans ce cas, *pointer[t]* reçoit la valeur y et le jeton *u* est envoyé à y;
Cas 4: Le nœud i possède un jeton *u*, mais se trouve en section critique: La requête de y est insérées dans la file '*token_queue*' correspondante au jeton *u*.

A la sortie de la section critique, si la file d'attente est vide, un message '*Inform'* est envoyé à tous les voisins du nœud i afin de les aviser qu'il détient un jeton, ce qui leur permettra de modifier leur structure *pointer* dans ce sens. Ce message est utile pour réduire la distance entre un nœud et un jeton. Sinon, le jeton est envoyé au noeud se trouvant au sommet de la file '*request_queue*' correspondante au jeton libéré.

Lorsqu'un nœud i reçoit un jeton, il vérifie si le jeton reçu est lui-même le jeton attendu; met à jour ses structures de données de sorte à maintenir la structure logique du réseau en prenant le rôle de racine pour la forêt relative au jeton reçu. Il procède ensuite à l'exécution de sa section critique.

V. Algorithme de M.Naimi

Cet algorithme [59] suppose l'existence d'un graphe connecté, sans perte de messages et dont les délais de transmission sont arbitraires mais finis. Aucun mécanisme de séquencement n'est proposé pour les messages. Deux structures de données sont utilisées:
a- La première est une structure de DAG (Direct Acyclic Graph) logique où les arcs sont orientés dans la direction des prédécesseurs. Initialement, un arbre logique est défini en dessus de la structure physique du réseau et K jetons sont attribués au nœud racine. Cette structure est utilisée afin de permettre la transmission des messages de requêtes. Le nœud demandeur d'accès à la section critique envoie une requête à son prédécesseur, insert sa propre identité dans sa file d'attente, devient racine et attend un jeton. Le graphe orienté est ainsi transformé à un autre graphe orienté, ce qui donne une foret comme dans l'algorithme présenté dans [58]; Si le nœud demandeur détient au moins un jeton, il entre immédiatement en section critique;
b- La deuxième structure est une file d'attente construite au niveau de chaque nœud. Lorsqu'un nœud sort de sa section critique, il envoie le jeton vers le nœud se trouvant au sommet de la file et supprime sa propre identité de celle-ci.

Lorsqu'un nœud x reçoit une requête de son voisin y (cela signifie que x est le prédécesseur de y), deux cas sont possibles:
Cas 1: Le nœud x détient au moins un jeton libre, dans ce cas, il envoie immédiatement un jeton au noeud y;
Cas 2: Le nœud x ne détient pas de jetons libres, il insert la requête de y dans sa file d'attente et transmet une requête à tous ses prédécesseurs.

Dans les deux cas, le nœud y devient prédécesseur du nœud x sur le graphe orienté, ce qui modifie dynamiquement la structure du graphe.

VI. Utilisation des K-coteries pour la K-exclusion mutuelle

Le concept de coterie a été présenté dans le chapitre précédent; il garanti l'exclusion mutuelle du fait que les quorums qui composent la coterie ne sont pas disjoints deux à deux (propriété d'intersection). Or, dans notre cas la K-exclusion mutuelle doit assurer que K processus au plus peuvent être simultanément dans leurs sections critiques, ce qui n'est réalisable que s'il y a exactement K quorums disjoints sachant que K+1 quorums ne doivent pas l'être. Ceci nous conduit intuitivement à la définition du concept de K-coterie qui est en fait, une extension de celui de coterie afin de l'adapter au problème de K-exclusion mutuelle.

VI.1. Présentation de l'approche K-coterie

***Définition*:** Un ensemble non vide C constitué de sous ensembles Q de U (ensemble de tous les processus du système) est appelé K-coterie si et seulement si les conditions suivantes sont réunies:

1-***Propriété de non intersection*:** Pour tout h (h < K) éléments $Q_1,\ldots,Q_h \in C$ tel que $Qi \cap Qj = \phi$ $(i \neq j)$ pour $1 \leq i,j \leq h$, il existe un élément $Q \in C$ tel que $Q \cap Qi = \phi$ pour $1 \leq i \leq h$;
2-***Propriété d'intersection*:** Pour tout K+1 éléments $Q_1,\ldots, Q_{k+1} \in C$, il existe une paire Qi et Qj tel que $Qi \cap Qj \neq \phi$;
3-***Propriété de minimalité*:** Pour deux éléments distincts quelconques Qi et Qj dans C, $Qi \not\subset Qj$ et $Qi \neq Qj$.

Remarques:
- Un élément d'une K-coterie est aussi appelé Quorum;
- Selon la définition de la K-coterie, la condition selon laquelle l'union des quorums est égale à l'ensemble initial U ($\cup_i Q_i = U$) n'est pas nécessairement vérifiée.

VI.2. Approches pour la construction des K-coteries

Beaucoup d'algorithmes sont basés sur les K-coteries [52][53][54][55][56]. Leur principe est simple: chaque processus désirant entrer en section critique doit obtenir la permission de tous les membres de son quorum. La différence essentielle entre ces algorithmes réside sur la manière de construire ces K-cotries; Les avantages des K-coteries sont tirés de ceux des coteries: tolérance au fautes et coût bas en termes de complexité de messages. Plusieurs propriétés ont été ajoutées aux K-coteries afin d'augmenter leur efficacité. Ainsi, d'autres concepts sont apparus tels que les K-coteries non dominées [53], le concept de k-majority [56], les structures de cohorts [55], arbre de quorum (quorum tree) [54], …etc. En général, les performances des algorithmes basés sur les K-coteries dépendent de la taille des quorums qui les composent.

Dans ce qui suit nous présentons ces concepts en relevant leurs avantages.

- <u>K-coteries non dominées</u>[53]

Une K-coterie C sous U (ensembles des processus du système) est dominée s'il existe une autre K-coterie sous U qui domine C, sinon C est une K-coterie non dominée. En d'autres termes :

Soit C1 et C2 deux K-coteries sous U. Alors C1 domine C2 si:

1. $C1 \neq C2$;
2. $(\forall H \in C2)\ [\exists G \in C1$ tel que $G \subseteq H]$

L'avantage des K-coteries non dominées [53] est qu'elles fournissent un plus haut niveau de disponibilité et une plus grande résistance aux pannes des nœuds et des liens de communication. Plusieurs méthodes de construction des K-coteries non dominées sont présentées dans [53].

- <u>Structures de cohorts</u>[55]

A travers la structure de cohorts[55], nous pouvons construire des k-coteries qui possèdent des qualités comparables aux autres structures.

On dénote par coh(k,n) = (C1, …,Cn), une liste d'ensembles disjoints deux à deux appelés cohorts ayant les propriétés suivantes:

1- $| C1| = K$;
2- $\forall i, 1 < i \leq n : | Ci| > \max(2K-2,K)$.

exemple: Dans un système à 10 noeuds numérotés de 1 à 10, on peut construire l'ensemble:
coh(2,3) = {{1,2}{3,4,5}{6,7,8,9,10}}.

Un ensemble Q est un quorum sous un ensemble de cohorts coh(K,n) si un sous ensemble de coh(K,n) Ci est un ensemble primaire de cohorts pour Q et un sous ensemble de coh(K,n) Cj (j>i) est un ensemble support de cohorts pour Q (supporting cohort) où,

1- Un cohort C est un cohort primaire pour Q si $| Q \cap C | = | C | - (K-1)$ (c'est-à-dire: Q contient tous les membres de C sauf K-1 membres);
2- Un cohort C est un cohort support pour Q si $| Q \cap C | = 1$ (c'est-à-dire: Q contient exactement un membre de C).

Les quorums construits sur un cohort coh(K,n) sont appelés quorums de cohorts, ils peuvent former une K-coterie [55].

Exemple: soit l'ensemble coh(2,2)){{1,2},{3,4,5}}, on peut construire une 2-coterie formée des quorums suivants :
Q1 = {3,4}, Q2 = {3,5}, Q3 = {4,5}, Q4={1,3}, Q5 = {1,5}, Q6 = {1,5}, Q7 = {2,3},
Q8 = {2,4}, Q9 = {2,5} où l'ensemble {3, 4, 5} représente le cohort primaire pour les quorums Q1, Q2 et Q3 sans aucun ensemble support; L'ensemble {1,2} est primaire pour les quorums Q4, …,Q9, leur cohort support étant l'ensemble {3,4,5}.

Il est à noter qu'en cas de partitionnement du réseau ou de panne de nœuds particuliers, il est toujours possible de construire une autre K-coterie sous le même cohort en isolant les nœuds en panne [55].

- <u>Structures logiques</u>

Deux stratégies sont proposées dans [54] pour la construction de K-coteries: 'Structure généralisée d'arbre binaire de quorums et structure étendue d'arbre binaire de quorums'. Elles sont basées sur la structure d'arbre binaire de quorums de [36]. Les deux stratégies sont tolérantes aux pannes de $(n-k\lfloor \lg_2 n/k \rfloor)$ nœuds dans les meilleurs cas et de $k(\lfloor \lg_2 K \rfloor)$ nœuds dans les pires cas. Ces deux stratégies présentent aussi une plus petite taille de quorum par rapport aux stratégies présentées précédemment.

- **Arbre binaire:** Un arbre binaire est un ensemble fini de nœuds tels que:

1- Il existe un nœud spécial appelé racine dans le niveau 0;
2- Les nœuds restants sont partitionnés en deux ensembles S1 et S2 tels que chacun d'eux est un arbre binaire. S1 et S2 sont appelés sous arbres binaires.

- **Arbre binaire de quorums:** Les nœuds du système sont initialement organisés en arbre binaire. Un arbre binaire de quorums pour la 1-exclusion mutuelle est défini récursivement comme étant:

1- La racine et un arbre binaire de quorums du sous arbre gauche, ou bien
2- La racine et un arbre binaire de quorums du sous arbre droit, ou bien
3- Un arbre binaire de quorums du sous arbre gauche et un arbre binaire de quorums du sous arbre droit.

- **Arbre binaire de quorums généralisé** Un arbre binaire de niveau (h+1) est une collection de noeuds arrangés par niveau Où $h \geq 0$ et $n = 2^{h+1} - 1$ (n est le nombre de nœuds dans la système).

Un ensemble Q est un quorum ayant la structure d'arbre binaire généralisé pour la K-exclusion mutuelle si la condition suivante est vérifiée: Pour un niveau i, tel que $2^i \leq K < 2^{i+1}$, $0 \leq i \leq h$, Q contient un arbre binaire de quorum pour la 1-exclusion mutuelle pour tout sous arbre S_{K-1}, S_K, …, S_{2K-2}.

- **Arbre binaire de quorums étendu** [54]**:** Dans une structure d'arbre binaire généralisée, les nœuds $j \leq (K-2)$ ne sont jamais utilisés; ces nœuds ne participent donc pas à la structure de K-coterie construite; d'où l'idée de créer des nœuds virtuels jusqu'au niveau (K-2) afin de faire participer tous les nœuds du système. On obtient ainsi un arbre binaire étendu (où n est le nombre de nœuds de l'arbre et NS est le nombre de nœuds représentant des sites réels) sur lequel on peut construire un arbre binaire de quorums étendu pour former notre K-coterie.

La structure d'arbre binaire étendue de niveau (h+1) est équivalente à la structure d'arbre binaire généralisée à l'exception que les nœuds 0,1, …, (K-2) sont virtuels;
Où $2i \leq K < 2i+1$ et $0 \leq i \leq h$.

Un ensemble Q est un quorum ayant la structure d'arbre binaire étendu pour la K-exclusion mutuelle si la condition suivante est vérifiée: Pour un niveau i, tel que $K = n - NS + 1$, $2^i \leq K < 2^{i+1}$ et $0 \leq i \leq h$, Q contient un arbre binaire de quorum pour la 1-exclusion mutuelle pour tout sous arbre S_{K-1}, S_K, …, S_{2K-2}.

VII. Algorithme de H.Kahugawa-S.Fujita-M.Yamashita-T.Ae

Cet algorithme [52] est basé sur la permission et utilise l'horloge logique proposée par Lamport [23] afin d'assurer un ordre total entre les requêtes.

Soit C une K-coterie construite sur le réseau. Chaque processus P possède les variables locales *YES*, *NOTNOW* et *PERM*. La variable *YES* (resp. *NOTNOW*) comprend l'ensemble des processus ayant permis par un message *OK* (resp. refusé par un message *WAIT*) l'entrée en section critique; La variable *PERM* comprend les processus auxquels P a envoyé le message *OK* et qui n'ont pas encore reçu le message *release* qui spécifie que P a quitté sa section critique. *PERM* contient ou bien un singleton ou bien est vide car P n'envoie sa permission qu'à un seul processus à la fois.

- *Quand P désire entrer en section critique*, il sélectionne un quorum Q de C et envoie une requête de la forme *REQUEST*(t,P) à tous les membres q de Q lui même inclus et attend une réponse *OK* ou *WAIT* de q. Si tous les processus q répondent par un *OK*, P peut entrer en section critique. Si quelques processus répondent par *WAIT*, P insert les processus ayant répondu *OK* (resp. *WAIT*) dans l'ensemble *YES* (resp. *NOTNOW*), sélectionne un autre quorum Q' qui minimise $|Q' \cap YES|$ et tel que $Q' \cap NOTNOW = \phi$ (1) dans la mesure du possible et répète la procédure en envoyant la requête aux processus appartenant à (Q'-*YES*). Si P ne trouve pas de quorum satisfaisant la condition (1), il attend de recevoir un message *OK* de tous les membres de *NOTNOW*.

Durant le déroulement du processus précédent, le processus P peut recevoir des messages *OK* en provenance d'éléments de *NOTNOW*, auquel cas ces processus seront supprimés de l'ensemble *NOTNOW* et insérés dans l'ensemble *YES* et si l'ensemble *YES* comprend un quorum alors P est libre d'entrer en section critique;

- *Lorsque P quitte sa section critique*, il envoie un message *release* à tous les processus de l'ensemble *YES* U *NOTNOW*;
- *Lorsqu'un processus P reçoit une requête d'un processus q*, P envoie un message *OK* si *PERM* est vide et affecte la requête à *PERM*. Si *PERM* contient déjà une requête S, P ajoute la requête de q dans la file. Si l'estampille de q n'est pas la plus petite (par rapport à la file et à *PERM*) alors P envoie un message *WAIT* à q sinon il envoie un message *QUERY* au processus S afin de reprendre la permission déjà accordée. Le processus S peut retourner un message *release* ou bien *relinquish* en cas d'abandon. Si P reçoit le message *relinquish*, il met le contenu de *PERM* dans la file, affecte q à *PERM* et envoie un message *OK* à q.
- *Lorsque P reçoit un message release de* (la requête de q se trouvant dans *PERM*), si la file n'est pas vide, alors la requête ayant l'estampille la plus basse (plus ancienne) sera affectée à *PERM* et le processus émetteur de cette requête reçoit un message *OK*, ainsi, P envoie un message *WAIT* à tous les processus dans la file auxquels P n'a pas envoyé ce message depuis le dernier message *QUERY* envoyé.
- *Quand un processus P reçoit un message QUERY d'un processus q*, si P n'est pas en section critique et q est dans *YES*, alors il fait passer q de *YES* à *NOTNOW* et envoie un message *relinquish* à q.

VIII. Tableau récapitulatif des algorithmes de K-exclusion mutuelle

Le tableau suivant permet de comparer les performances des algorithmes de K-exclusion mutuelle. En général, les algorithmes utilisant les K-coteries adoptent le même fonctionnement et leurs performances dépendent de la taille des quorums construits.

Algorithme	Année	Nombre de messages	Approche
K.Raymond [50]	1989	De 2N-K-1 à 2*(N-1)	Permission
Srimani-Reddy [51]	1992	De 0 à N+K-1	Jeton (K jetons)
Kakugawa & all [52]	1993	De 3*S à 6*S	Permission avec utilisation des K-coteries
S.Bulgannaxar&N.H.Vaidya[58]	1994	< N+K-1	Jeton (K jeton), utilisation de foret dynamique
M.Naimi[59]	/	De 0 à 2*(N-1)	Jeton (K jetons), utilisation de DAG (foret) dynamique

Table 2. Tableau récapitulatif des algorithmes de K-exclusion mutuelle

Où,

K: nombre de processus autorisés simultanément;
N: nombre de nœuds dans le système;
S: taille maximale du plus grand quorum.

IX. conclusion

Dans le problème de K-exclusion mutuelle, les accès concurrents aux ressources partagées ou section critique doivent être synchronisés de telle sorte qu'au plus K processus peuvent accéder à leurs sections critiques simultanément ($1 \leq K \leq N$).

Plusieurs algorithmes ont été proposés, les uns [50][51] sont une généralisation des algorithmes déjà existants concernant la 1-exclusion mutuelle ou construisent des topologies logiques en dessus de la structure physique du réseau [58][59], d'autres [52][53][54][55][56] ont adopté une nouvelle stratégie basée sur les K-coteries qui est elle-même une extension de celle de coterie.

Les algorithmes basés sur les K-coteries se différencient par la manière de construire les K-coteries. Plusieurs stratégies sont offertes telles que les structures de cohorts [55], les K-coteries non dominées (non dominated K-coteries) [53],…etc. Certains algorithmes [54] utilisent des structures logiques pour former les quorums telles que la structure d'arbre binaire de quorums.

Les algorithmes basés sur les K-coteries sont résistants aux pannes des nœuds ou des liens de communication, au partitionnement du réseau et offrent des coûts en terme de nombre de messages relativement bas.

Chapitre 2 La K-exclusion mutuelle dans les réseaux mobiles Ad Hoc

A. L'exclusion mutuelle dans les réseaux mobiles Ad Hoc

I. Introduction

L'exclusion mutuelle est un problème fondamental dans les systèmes distribués. Il consiste à allouer en exclusivité une ressource critique à un et un seul processus à la fois. Un grand nombre d'auteurs ont contribué à la résolution de ce problème. Les solutions proposées s'appuient sur divers outils et concepts développés afin de construire des algorithmes qui répondent au mieux aux exigences de vivacité, de sûreté et de tolérance aux pannes. Deux approches fondamentales ont été dégagées: Dans les algorithmes d'exclusion mutuelle basés sur le jeton, la possession du jeton représente un droit d'accès; Dans les algorithmes basés sur la permission, le nœud demandeur doit obtenir la permission de tous (ou un sous ensemble) les nœuds du réseau.

Les caractéristiques des réseaux mobiles ad hoc (interfaces de communication sans fil, topologie arbitraire fréquemment modifiée à cause de la mobilité des nœuds et la source d'énergie limitée des équipements mobiles) représentent un véritable défi quant à la manière de concevoir les protocoles à implémenter dessus.

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

Les premières solutions au problème d'exclusion mutuelle pour l'environnement mobile ad hoc sont dues à Walter [43] et Baldonni [44] elles s'appuient sur les solutions bâties pour les réseaux fixes moyennant une adaptation qui tient compte des paramètres de la mobilité. En outre, il s'avère que les solutions basées sur le jeton sont les plus appropriées pour cet environnement.

Dans cette section, nous étudions ces solutions en dégageant les principes de chacune.

II. L'exclusion mutuelle dans les réseaux mobiles ad hoc

Un algorithme d'exclusion mutuelle dans un environnement mobile ad hoc est un algorithme distribué, il possède les mêmes propriétés; C'est à dire, qu'il doit garantir l'exclusion mutuelle sans interblocage ni famine. En outre, il doit tenir compte des caractéristiques des réseaux mobiles ad hoc qui représentent un véritable défi dans la construction de ces algorithmes compte tenu des phénomènes aléatoires qui le caractérisent.

Les performances d'un algorithme d'exclusion mutuelle dans un environnement mobile ad hoc se mesurent en tenant compte des paramètres suivants:

- ✓ ***Le nombre de messages*** échangés par accès en section critique;
- ✓ ***Le délais de synchronisation***: c'est le temps écoulé entre deux entrées successives en section critique, il est exprimé par le nombre maximum de messages séquentiels nécessaires après qu'un nœud i quitte sa section critique et avant qu'un nœud j entre dans sa section critique;
- ✓ ***La complexité de routage***: le nombre de messages de contrôle générés par la couche réseau ;

Les algorithmes distribués d'exclusion mutuelle construits pour l'environnement mobile ad hoc sont obtenus à partir d'une adaptation des algorithmes développés pour les réseaux fixes; On peut envisager deux architectures pour l'implémentation de ces algorithmes:
La première architecture suppose que l'algorithme d'exclusion mutuelle est implémenté au dessus du protocole de routage (*figure 3*).

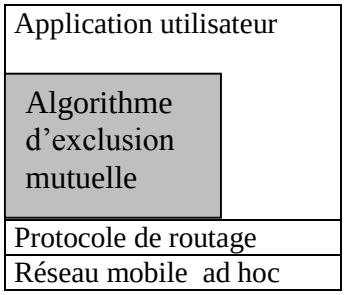


Figure 3. Première approche pour l'implémentation de l'algorithme d'exclusion mutuelle.

Dans ce cas, l'algorithme ne tient pas compte de la mobilité, cette tâche est assurée par le protocole de routage au niveau de la couche réseau. Ce fait est illustré par la *figure 4* : Le système est composé de neuf nœuds; Dans le cas statique, si le nœud D envoie une requête, alors elle va traverser le chemin D-A-C-F (Le nœud F étant le détenteur du jeton).
Considérons le scénario suivant présenté dans la *figure 4*, où les nœuds A et D se déplacent de telle sorte que le lien physique entre A et C est rompu, et qu'un nouveau lien entre A et F soit formé. Pour l'algorithme d'exclusion mutuelle, le lien logique entre A et C existe. Si D envoie une requête, le protocole de routage est invoqué pour localiser le nœud C. La requête va donc traverser le chemin D-A-F-C-F qui contient un cycle, ce qui augmente le coût en terme de nombre de messages et d'énergie consommée.

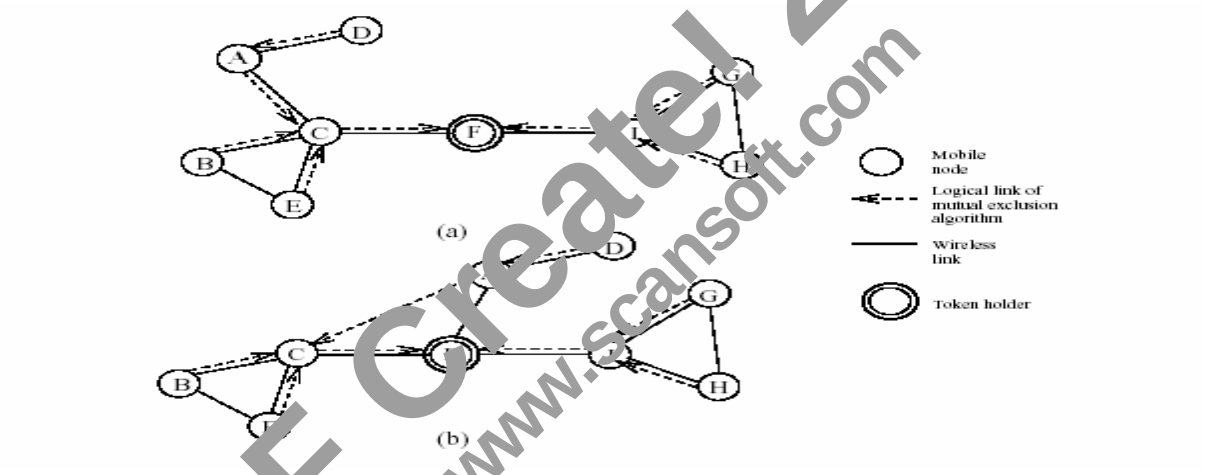


Figure 4. Influence de la mobilité sur le chemin emprunté par les messages

Cette approche ne convient plus à un environnement de réseaux mobiles ad hoc, où le mouvement des nœuds peut engendrer la rupture de liens existants ou la création de nouveaux. En outre, dans cet algorithme, les liens logiques ne correspondent pas nécessairement aux liens physiques, ce qui implique que les messages peuvent traverser des chemins contenant des cycles. Ceci nous conduit à une autre architecture qui implémente l'algorithme d'exclusion mutuelle au niveau de la couche réseau (voir figure 5).

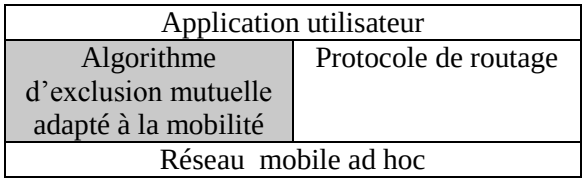


Figure 5. Deuxième approche pour l'implémentation de l'algorithme d'exclusion mutuelle.

1. Algorithme Reverse Link (RL)

Cet algorithme [43] est basé sur le jeton. La technique '*partial reversal*' est utilisée pour maintenir une structure de DAG (Direct Acyclic Graph) orientée vers un nœud spécial: le nœud privilégié ou le propriétaire du jeton. Chaque nœud du réseau maintient une file de requêtes contenant les identificateurs des nœuds voisins de qui une requête a été reçue.

a. Hypothèses

Les hypothèses sur lesquels est construit cet algorithme sont :

- Chaque nœud a un identificateur unique;
- La défaillance des liens ne peut pas se produire;
- Les liens de communication sont bidirectionnels et FIFO;
- La couche liaison assure que chaque nœud connaît l'ensemble de ses voisins en fournissant un indicateur de défaillances ou de formations de liens;
- Le partitionnement du réseau ne peut pas se produire.

b. Description du système

Le système est composé de N nœuds mobiles et indépendants identifiés de 0 à N-1 et communiquant par envoi de messages à travers des liens sans fil. Les nœuds du réseau sont totalement ordonnés par l'algorithme grâce à un triplet (α, β, γ) représentant le poids accordé à chaque nœud dynamiquement. Le nœud de poids le plus faible est le nœud privilégié à qui toutes les requêtes sont destinées. Ainsi, il représente le nœud racine '*sink node*' du DAG. Chaque nœud choisi dynamiquement parmi ses voisins directs le nœud de poids le plus faible pour être sur le chemin qui le mène vers le jeton.

A chaque nœud sont implémentés deux processus, un processus d'application et un processus d'exclusion mutuelle pouvant communiquer et définissant trois états: *REMAINDER* (n'est pas intéressé par la section critique), *WAITING* (en attente de la section critique) et *CRITICAL* (en section critique).

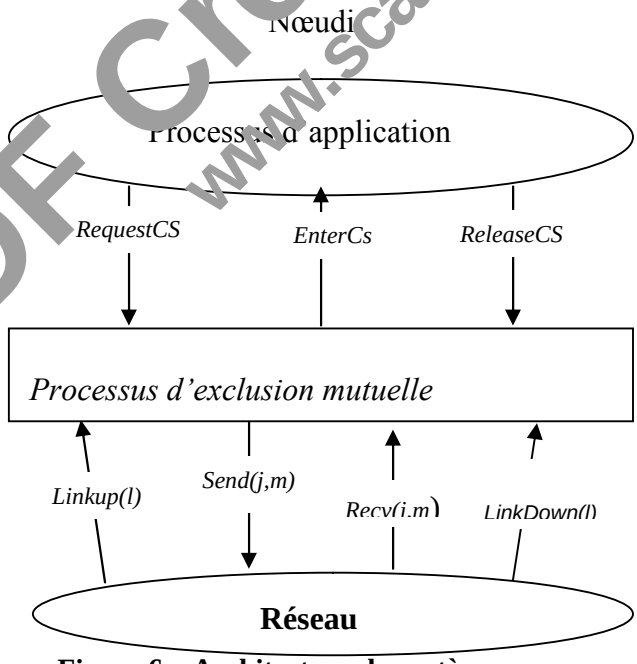


Figure 6. Architecture du système

Comme c'est indiqué par la figure 6, le processus d'exclusion mutuelle au niveau d'un nœud i est déclenché par un ensemble d'évènements d'entrée:

RequestCs: Le processus d'application dans le nœud i demande d'accéder à la section critique et l'algorithme entre dans l'état *WAITING*;
ReleaseCs: Le processus d'application dans le nœud i sort de sa section critique et l'algorithme entre dans l'état *REMAINDER*;
Recv(j,m): Le nœud i reçoit un message m du nœud j;
LinkUp(l): Le nœud i reçoit une notification indiquant qu'un lien l est formé;
LinkDown(l): Le nœud i reçoit une notification indiquant qu'un lien l est rompu.

En fonction des évènements d'entrée et de l'état du nœud, les événements de sortie suivants sont produits:
EnterCs: Le processus d'exclusion mutuelle dans le nœud i informe le processus d'application qu'il peut entrer dans l'état *CRITICAL* signifiant l'accès à la section critique;
Send(j,m): Le nœud i envoie le message m au nœud j.

Les évènements *RequestCs, EnterCs, ReleaseCs* sont appelés '*évènements d'application*' quant aux événements *Recv(j,m), Send(j,m), LinkUp(l), LinkDown(l)* sont appelés '*évènements réseau*'.

Les structures de données : Chaque nœud maintient à son niveau les informations suivantes :
- *status*: indique l'état du processus: *WAITING*, *REMAINDER*, *CRITICAL*; son état initial étant *REMAINDER ;*
- *Ni* : l'ensemble des voisins du nœud i;
- *myheight* : c'est un triplet (h1, h2, i) représentant le poids ou la taille d'un nœud i. Les nœuds sont liés du nœud ayant le poids le plus fort au nœud ayant le poids le plus faible. Par exemple, si $myheight_1$ = (2,3,1) et $myheight_2$ = (2,2,2) alors $myheight_1 > myheight_2$ et le lien est dirigé du nœud 1 au nœud 2. Initialement, au nœud 0, $myheight_0$ = (0,0,0) et pour tout i ≠0, $myHeight_i$ est initialisé de sorte que l'orientation du DAG soit dirigée vers le nœud 0;
- *height[j]*: un tableau qui contient les tailles des nœuds voisins vu du nœud i, si j est un voisin de i alors le lien entre i et j est entrant si *height[j] > height[i]* et sortant si *height[j] < height[i]* ;
- *tokenHolder*: un booléen qui vaut TRUE si le nœud détient le jeton, FALSE sinon; initialement, *tokenHolder* = TRUE si i = 0 sinon *tokenHolder* = FALSE;
- *next* : si le nœud i détient le jeton, *next* = i sinon *next* désigne le nœud sur le lien sortant;
- *Q*: file des requêtes de nœuds voisins ;
- *receivedLI[j]*: tableau booléen initialisé à TRUE pour tout j ∈Ni, *receivedLI[j]* est à FALSE lorsque j est le nœud auquel le jeton est récemment acheminé;
- *forming[j]*: tableau de booléens initialisé à TRUE quand le lien vers le nœud j est détecté, FALSE sinon ;
- *formHeight[j]*: tableau contenant des triplets pour mémoriser les poids des nœuds dès leur première détection. Initialement *formHeight[j] = myHeight* pour tout j∈Ni ;

Les procédures : L'exécution de l'algorithme RL est déclenchée par des évènements; un évènement au nœud i consiste en la réception d'un message d'un nœud j (j ≠ i), ou une indication de rupture ou de formation d'un lien. Certaines procédures sont déclenchées par l'application à travers une requête *RequestCs* ou un message *ReleaseCs*. Chaque message envoyé est accompagné par la valeur *myHeight*. Les modules sont exécutés automatiquement et se présentent comme suit :

Demande et sortie de la section critique : Quand un nœud i désire entrer en section critique, il met sa propre identification dans la file Q et affecte à *status Waiting*. Si le nœud i n'est pas le nœud privilégié et possède un seul élément dans sa file (lui même), il appelle *ForwardRequest*() pour envoyer un message de requête. Si le nœud i possède le jeton et il est en tête de file, il met sont *status* à *CRITICAL* et entre en section critique. Quand i quitte sa section critique, il appelle *GiveTokenToNext*() pour envoyer le message de jeton au nœud en tête de file si la file n'est pas vide et met son *status* à *REMAINDER*.

Réception d'un message de requête : A la réception d'un message de requête d'un nœud voisin j, i ignore la requête si *receivedLI*[j] est FALSE. Sinon, i met à jour *height*[j], met j en queue de file. Si la

file n'est pas vide et *status=REMAINDER* et i détient le jeton, i appelle la procédure *GiveTokenToNext*(). Dans le cas où le nœud i n'a pas de liens sortants, il appelle la procédure *RaiseHeight*() pour mettre à jour ses liens.

***Remarque*:** À sa première version [41], l'algorithme de Walter adoptait une autre politique quant aux deux procédures précédentes; Si un nœud i désire entrer en section critique, il n'enfile sa requête que si sa file de requêtes n'est pas vide ou qu'il ne possède pas le jeton. Si le nœud i possède le jeton, il entre en section critique à condition que sa file ne soit pas vide auquel cas (Qi >1) le nœud i attend l'arrivée du jeton (car il aurait déjà soumis une requête). Si le jeton n'atteint pas le prochain nœud privilégié en un temps limité, le détenteur courant du jeton lance un message *Find*(*pathi*,*i*,θi) à tous les nœuds voisins dont les liens sont entrants (où *pathi* est la destination du jeton, *i* la source et θi est la valeur de LCi: Local Clocki). Ce message est propagé sur tous les liens entrants jusqu'au nœud *pathi*. Ce nœud va répondre par un message *Reply* qui va traverser le chemin inverse au chemin du message *Find* reçu. Ce dernier chemin sera utilisé pour envoyer le message de jeton. Le jeton est toujours accompagné d'une information sur la taille de la file de l'émetteur afin d'assurer son retour au cas où elle n'est pas vide.

Réception d'un message de jeton : Quand un nœud i reçoit un message de jeton d'un nœud voisin j, il met sa variable t*okenHolder* à TRUE, il met à jour son poids tel qu'il devienne plus petit que celui de j, et envoie un message *LinkInfo* pour informer ses voisins (qui ont un lien sortant de i) qu'il est le nœud privilégié. Ensuite, le nœud i appelle la procédure *GiveTokenToNext*(). Le nœud i informe aussi le nœud j de son nouveau poids en considérant ce dernier message comme un accusé de réception.

La mise à jour du poids (αi, βi, i) du nœud i se fait en appliquant la formule suivante:

$$\alpha_i=\min\{\alpha_k/\ k \in Ni\} \quad \text{et } \beta i= \begin{cases} \text{Min}\{\beta_k/\ k\in Ni \wedge (\alpha_i=\alpha_k \wedge \beta_i \geq \beta_k)\}-1 \\ \beta_i \text{ sinon} \end{cases}$$

Formule 1 Diminution du poids des nœuds

Le message LinkInfo : Si *receivedLI*[j] est à TRUE à la réception du message *LinkInfo* du nœud j, le poids de j est sauvegardé dans *heightL*[j] sinon, i vérifie si le poids reçu du nœud j est le même que celui connu lors du dernier message *Token* envoyé auquel cas i met *receivedLI*[j] à TRUE (il s'agit d'un accusé de réception du message de jeton de la part du nœud j).

Si *forming*[j] est à TRUE (c'est un message LinkInfo envoyé par j après sa détection d'un nouveau lien avec i), la valeur courante de *myHeight* est comparée à *formHeight*[j] (ce dernier est le poids du nœud i à la détection du lien avec le nœud j). Si elles sont différentes, alors un message *LinkInfo* est envoyé à j, l'identificateur de j est ajouté à l'ensemble Ni et *forming*[j] est mis à FALSE.

Si le nœud j est un élément de Q et j représente un lien sortant de i, j est supprimé de la file d'attente Q.

Si le nœud i n'a pas de liens sortants et n'est pas le nœud privilégié, i appelle *RaiseHeight*() pour qu'un nouveau lien sortant vers le nœud privilégié soit formé.

Si Q n'est pas vide et le lien vers le nœud *next* est inversé alors i appelle *ForwardRequest*() pour envoyer la prochaine requête.

Rupture de lien : Quand le nœud i détecte une rupture de lien vers un voisin j, il enlève j de *Ni*, met *receivedLI*[j] à TRUE et si j est un élément de Q, il est supprimé de la file. Ensuite, si i n'est pas le nœud privilégié et ne possède pas de liens sortants, il appelle *RaiseHeight*(). Si i n'est pas le nœud privilégié, Q est non vide et le lien avec le nœuds *next* est rompu alors i appelle la procédure *ForwardRequest()*.

Formation de lien :A la détection de formation d'un nouveau lien d'un nœud j vers le nœud i, i envoie un message *LinkInfo* à j accompagné de *myHeight*, met *forming*[j] à TRUE et affecte *myHeight* à *formHeight*[j].

Procédure ForwardRequest : Sélectionne le nœud dont le poids est le plus petit parmi les voisins de i pour être dans *Nexti* et envoie un message de requête à celui ci.
Procédure d'envoi de jeton (GiveTokenToNext) : Le nœud retire de la file l'élément de tête, l'affecte à *next*. Si i=*next* alors i entre en section critique. Sinon, i diminue le poids de *next* de sorte que toutes les requêtes qui arrivent à i soient acheminées vers *next*, met *tokenHolder* à FALSE, *receivedLI*[*next*] à FALSE et envoie le message de jeton à *next*. Si Q n'est pas vide un message de requête est envoyé à *next* donc le jeton peut éventuellement retourner à i.
Procédure RaiseHeight : Appelée quand le nœud i n'a pas de jeton et qu'il a perdu son dernier lien sortant. Le nœud i élève son poids selon la *formule 2* et informe tous ses voisins de son nouveau poids avec le message *LinkInfo*.

$$\alpha_i = \min\{\alpha_k / k \in Ni\} + 1 \quad \text{et } \beta_i = \begin{cases} \text{Min}\{\beta_k / k \in N_i \wedge (\alpha_i = \alpha_k)\} - 1 \\ \beta_i \text{ sinon} \end{cases}$$

Formule 2 Augmentation du poids des nœuds

Toutes les requêtes issues des nœuds voisins ayant un lien sortant du nœud i (poids plus faible) sont supprimées. Si Q n'est toujours pas vide la procédure *ForwardRequest*() est appelée.

c. Exemple de déroulement de l'algorithme RL dans un réseau statique

Nous illustrons à travers la *figure 7* un déroulement de l'algorithme d'exclusion mutuelle dans le cas statique c'est à dire que les nœuds ne sont pas en mouvement.

Les liens directs entre les nœuds sont indiqués par des flèches en pointillés. Les liens de chaque nœud avec son nœud *next* sont indiqués par une flèche simple. La file de requêtes est indiquée par un rectangle devant chaque nœud.

Dans la *figure 7.a* les nœuds 2 et 3 ont demandé l'accès à la section critique au nœud 0. Il est à noter que les nœuds 2 et 3 ont enfilé leur propre requête. Plus tard (*figure 7.b*), le nœud 1 demande l'accès en section critique à son *next* (le nœud 3). Celui-ci enfile la requête. A sa sortie de la section critique (*figure 7.c*), le nœud 0 envoie le jeton au nœud 3, suivi d'un message de requête pour le compte du nœud 2. Observons le changement du lien *next* entre les nœuds 0 et 3 et la diminution du poids du nœud 3. La *figure 7.d* montre l'envoi du jeton du nœud 3 vers le nœud 1, suivi d'une requête pour le compte du nœud 0. Enfin le nœud 2 reçoit le jeton qui, partant du nœud 1, traverse les nœuds 3 ensuite 0 avant d'atteindre sa destination. Notons la diminution du poids à chaque passage du jeton par un nœud de sorte que le poids le plus faible soit celui du nœud privilégié.

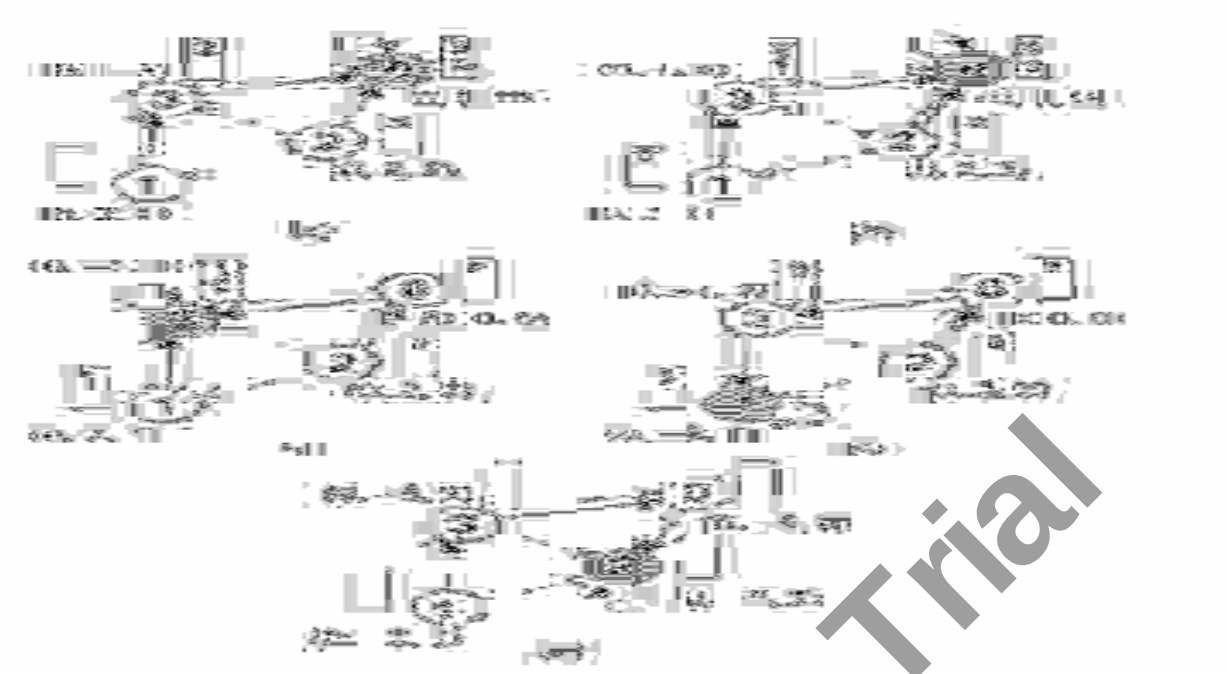


Figure 7. Déroulement de l'algorithme RL dans un réseau statique

d. Exemple de déroulement dans un réseau dynamique

Maintenant, nous allons considérer le déroulement dans le cas dynamique où les formations de liens et déconnexions sont prises en compte:

Comme pour le cas statique, la *figure 8.a* montre les deux requêtes issues des nœuds 2 et 3 vers le nœud 0 qui est détenteur du jeton. La *figure 8.b* montre la rupture des liens 0-3 et 2-3 après mouvement du nœud 3. En conséquence, le nœud 2 a supprimé la requête du nœud 3 de sa file. Dans la *figure 8.c*, le nœud 3 a augmenté son poids et a envoyé une requête à son nouveau *next* (le nœud 1). Le nœud 1 n'ayant pas de lien sortant (*figure 8.d*), a augmenté son poids (voir les lien de la figure 8.e) et a envoyé une requête au nœud 2 qui l'a enfilé sans envoyer de requête au nœud 0 car il est en attente du jeton.

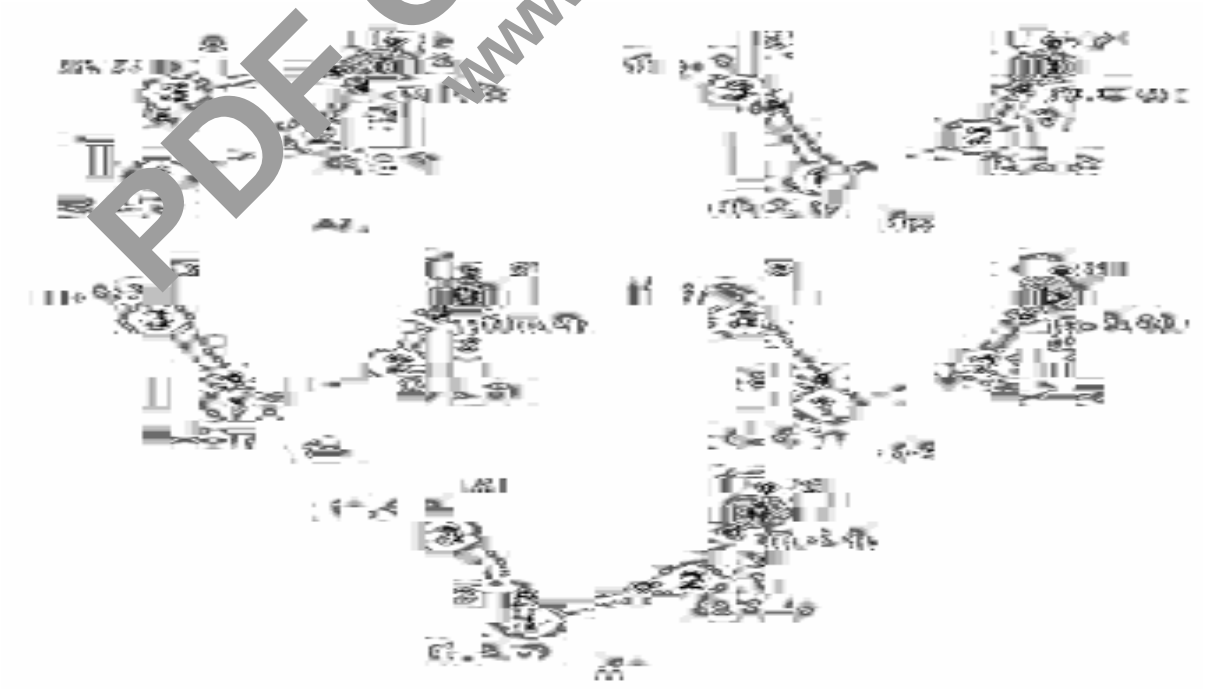


Figure 8. Déroulement dans un réseau dynamique

e. Les avantages de l'algorithme RL

Les points forts relevés pour l'algorithme **RL** sont :

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

f. Les inconvénients de l'algorithme RL

Les inconvénients de cet algorithme sont:

- Quelques hypothèses ne sont pas réalistes par exemple '*Le partitionnement du réseau ne peut pas se produire*'**;**
- L'algorithme prend en charge le cas de défaillance et formation de liens, ce qui représente un travail supplémentaire en terme de communication et de consommation d'énergie;
- Un nœud qui a envoyé une requête et perd son lien avec son voisin *next,* renvoie la requête vers un autre voisin, ce qui d'une part augmente le temps d'attente pour un nœud et peut produire une situation de famine si les changements de la topologie sont très fréquents; d'autre part, si le voisin *next* n'est pas le nœud privilégié, la requête peut figurer sur plusieurs chemins;
- Un nœud privilégié qui n'est pas en tête de sa propre file envoie le jeton pour le demander une nouvelle fois;
- Des nœuds qui n'ont pas demandé l'accès en section critique participent à l'exécution de l'algorithme.

2. Algorithme de R.Baldoni-A.virgillito

Cet algorithme [44] est basé sur deux principes: '*demande de jeton*' (*token asking*) et '*circulation de jeton*' (*circulating token*) afin d'assurer un nombre optimal de jetons échangés pour chaque accès en section critique.

La structure en anneau logique des processus est réalisée de manière dynamique: chaque processus recevant le jeton de son prédécesseur décide selon une politique donnée P quel sera le successeur parmi les nœuds qui n'ont pas encore reçu le jeton. La politique P peut être choisie en fonction de paramètres spécifiques par exemple: le jeton est envoyé au processus le plus proche en terme de nombre de sauts.

Chaque tour de jeton est caractérisé par un coordinateur (on désigne par C_k le processus coordinateur pour le tour k). Ce dernier représente le processus auquel toutes les requêtes sont destinées. La circulation en anneau du jeton à travers les processus est déclenchée par le processus coordinateur afin d'informer tous les autres processus de l'identité du coordinateur courant et de permettre au processus qui le reçoit d'entrer en section critique s'il le désire.

a. Hypothèses

Le système est composé d'un ensemble de N≥1 processus $\{P_1,\ldots, P_n\}$ se déroulant chacun sur un nœud mobile, les hypothèses sur lesquelles repose cet algorithme sont:

- Les processus ne tombent pas en panne ;
- Les liens de communication sont fiables ;
- Le réseau n'est pas l'objet d'un partitionnement permanent ; si un partitionnement se produit, chaque paire de processus pourra après un temps fini rétablir la communication.

b. Circulation du jeton

A l'initialisation, le coordinateur C0 correspond au processus P1. Tous les processus connaissent l'identité de C0.

Un tour de jeton k commence à partir du coordinateur C_k et n'inclus pas C_{k-1}. Chaque processus maintient un vecteur booléen de taille n appelé *receivedToken*. Initialement le coordinateur C_{k-1}, soit

P_i, affecte la valeur *FALSE* à chaque entrée du vecteur *receivedToken* sauf *receivedToken[i]* qui est initialisée à *TRUE* avant d'envoyer le jeton accompagné du vecteur *receivedToken* au nouveau coordinateur C_k. Quand un processus P_i reçoit le jeton durant le tour de jeton coordiné par Ck, il affecte la valeur *TRUE* à *receivedToken[i]* et défini un ensemble de successeurs $possSucc_i$ comme suit:

$$possSucci(\mathcal{P})=\{P_l/\ (receivedToken[l]=FALSE)\wedge(l\neq i)\ \} \quad (1)$$

Si $possSucc_i$ est vide le jeton est envoyé au coordinateur C_k (le tour est accompli); Sinon, la politique P est appliquée pour déterminer le successeur du processus P_i (qui est par hypothèses toujours accessible en un temps fini). Le tour est ainsi construit de manière dynamique et la politique P peut aisément être substituée sans aucun impact sur le reste de l'algorithme.

Après un tour k de jeton, le coordinateur C_k envoie le jeton au premier processus dans sa pile de requêtes appelée *pendingRequest*(qui est ordonnée selon une politique donnée : le processus de plus petit identificateur en premier, FIFO, …etc), soit P_j. P_j devient le coordinateur et qui doit initier le tour k+1qui peut être différent du tour k du fait de la mobilité des nœuds (voir Figure 9).

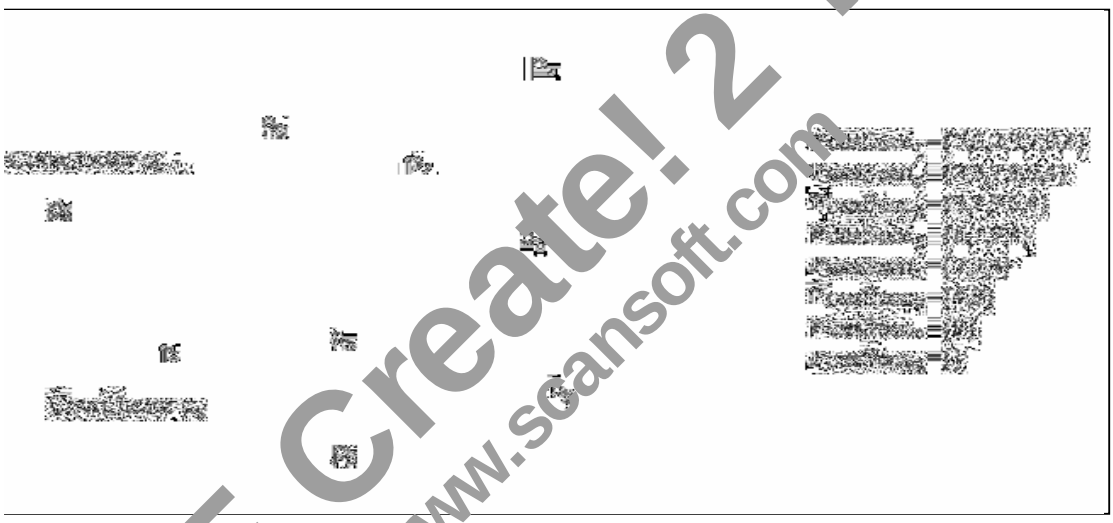


Figure 9. Exemple d'un anneau logique

c. Description de l'algorithme

La description de l'algorithme est donnée en trois parties : la première décrit les structures de données maintenues au niveau de chaque processus, la seconde présente la liste des messages échangés et la troisième partie explique le comportement de chaque processus

Structures de données: Chaque processus p_i maintient l'ensemble des données locales suivant :

- *coordinator*: c'est l'identificateur du nœud coordinateur le plus récemment connu par p_i ; initialement, *coordinator* = p_1 pour chaque processus ;
- *stat*: un bit qui peut être dans l'état IDLE ou NOT-IDLE. L'état IDLE signifie que p_i est le coordinateur et que le jeton est en sa possession. Autrement *stat* est à l'état NOT-IDLE ;
- *tokenHolder*: un booléen qui peut être TRUE si p_i a reçu le jeton et peut donc entrer en section critique. Initialement *tokenHolder* est à TRUE dans p_1 et FALSE dans tous les autres processus
- *requestCs*: un booléen initialement FALSE ; Si p_i possède une requête alors *requestCs* reçoit TRUE ;

- *receivedToken*: un vecteur de type booléen dont toutes les entrées sont initialement à FALSE (sauf pour le processus p_1 où *receivedToken* = TRUE); il est envoyé avec le jeton pour donner l'état de la circulation du jeton et sert à calculer l'ensemble *possSucc(P)*;
- *pendingRequest(*P*)*:une file organisée selon la politique P ; initialement elle est vide ;
- *possSucc(*P*)*: structure de données comprenant tous les successeurs de p_i selon la formule donnnée en (1).

Messages échangés: Un processus communique avec les autres en envoyant l'un des messages suivants:

- *Token*(*type,rt,coordinator*): consiste en l'envoi du message de jeton. Trois paramètres sont passés : *rt* qui contient le vecteur *receivedToken* de l'émetteur, *coordinator* qui contient l'identité du coordinateur et le paramètre *type* qui peut prendre deux valeurs: *COORDINATOR-CHANGE* ou *NEW-COORDINATOR*. Si *type=NEW-COORDINATOR*, ceci signifie que l'émetteur du message est un coordinateur et qu'il vient de désigner le prochain coordinateur (qui est le récepteur du message). Si *type=COORDINATOR-CHANGE*, ceci signifie que le jeton est en cours de sa tournée.
- *Request*: Le processus émetteur de ce message lance une requête à destination du coordinateur.

Comportement des processus: Lorsqu'un processus désire entrer en section critique, il affecte TRUE à *requestCS* et envoie le message *Request* au coordinateur. L'envoi est omis si p_i est lui-même coordinateur. Lorsque le jeton arrive, p_i entre dans sa section critique. A sa sortie, le processus p_i remet *requestCs* à FALSE et, si le paramètre *type* est à *COORDINATOR-CHANGE*, le processus p_i envoie le jeton à son successeur dans l'anneau logique avec les nouvelles données; Si p_i est coordinateur et sa file *pendingRequest* est vide, il garde le jeton, dans le cas où la file n'est pas vide, le coordinateur envoie le jeton au premier processus de *pendingRequest* qui devient le nouveau coordinateur et se reconnaît comme tel; p_i vide alors *pendingRequest*.

A l'arrivée d'une requête au processus p_i, si p_i est le coordinateur courant la requête est insérée dans *pendingRequest*($\mathcal{P}$) sinon elle est ignorée.

d. Les avantages de l'algorithme

Les points fort de cet algorithme sont listés comme suit :

- L'anneau est construit dynamiquement par l'envoi du jeton, au processus qui demande le moindre effort en termes de ressources du réseau et d'énergie ;
- La politique $\mathcal{P}$ présente une stratégie heuristique, qui tient compte de la mobilité des nœuds avec la supposition que l'algorithme utilise les informations fournies par le protocole de routage ;
- La politique $\mathcal{P}$ peut être substituée sans qu'il n'y ait un impact sur le reste de l'algorithme.

e. Les inconvénients de l'algorithme

Les inconvénients relevés sont :

- Pour une seule requête, le jeton doit traverser tout l'anneau logique, ce qui représente une charge supplémentaire en terme de communication et de consommation d'énergie ;
- L'algorithme défini un ordre total sur tous les processus du système (chaque nœud connaît les informations de routage de tous les nœuds du réseau), ce qui implique que le protocole de routage exécuté par les nœuds est *proactif* ;
- Des nœuds qui n'ont pas demandé l'accès en section critique participent à l'exécution de l'algorithme;
- L'envoi du jeton est accompagné d'un vecteur de type booléen ce qui augmente la taille du message de jeton ;
- L'évaluation des performances de l'algorithme ne tiens pas compte des messages de routage;
- Le calcul des successeurs à chaque réception de jeton peut être coûteux si le nombre de nœuds est grand;

- Certain messages envoyés sont inutiles; par exemple: envoi d'un message de requête au coordinateur alors que le jeton est en cours de circulation; sachant en plus que ces messages seront ignorés;
- L'utilisation de la file *pendingRequest* ne sert qu'à sélectionner le prochain coordinateur: il n'y a pas de gestion des requêtes au niveau de celui-ci.
- Les hypothèses ne sont pas réalistes car le partitionnement du réseau peut durer un temps aléatoire et qui peut ne pas être limité; deux dangers peuvent causer une détérioration des performances (partitionnement pour une durée limitée) ou un état de famine (partitionnement pour une durée illimitée):
  1-le coordinateur quitte le réseau qu'il soit détenteur du jeton ou pas car c'est lui qui décide de l'identité du prochain coordinateur ;
  2-un nœud quelconque peut recevoir le jeton sans pouvoir le retransmettre à cause d'un partitionnement subit;

III. Conclusion

Nous avons étudié deux algorithmes d'exclusion mutuelle dans les réseaux mobiles ad hoc. Ces algorithmes proposent l'implémentation de topologies logiques dynamiquement modifiables (DAG, Anneau) par l'évolution même de l'exécution de l'algorithme. A chacun de ces algorithmes, un détail du fonctionnement ainsi que les avantages et inconvénients ont été présentés. La prochaine section fera l'objet d'une étude de la K-exclusion mutuelle dans les réseaux mobiles ad hoc.

B. La k-exclusion mutuelle en environnement mobile ad hoc

I. Introduction

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

Plusieurs applications peuvent émerger de la k-exclusion mutuelle notamment dans le domaine de la robotique, l'aéronautique,…etc. Malheureusement, la k-exclusion mutuelle dans les réseaux mobiles ad hoc reste encore un domaine à développer vu le manque de travaux traitant ce sujet.

Nous présentons dans cette section le seul algorithme connu de K-exclusion mutuelle dans les réseaux ad hoc existant dans la littérature [60] et qui a été modifié dans [61] en vue d'obtenir de meilleures performances en temps de réponse aux requêtes en attente. Cet algorithme est une généralisation de l'algorithme RL [43] présenté dans la section précédente.

II. Algorithme KRL

L'algorithme que nous présentons ici [60] est une généralisation de l'algorithme RL présenté dans la section précédente. Il permet de maintenir K jetons dans le système; quand K=1 (1-exclusion mutuelle), le nœud dont l'identificateur est le plus faible détient le jeton et quand K>1, plusieurs nœuds détiennent un jeton ou plus et sont la destination éventuelle des requêtes. Initialement, les nœuds dont l'identification appartient à l'intervalle [0, K-1] sont les détenteurs du jeton.

Un DAG est maintenu sur les liens à travers l'exécution de l'algorithme. Les nœuds sont dirigés du nœud de plus grand poids vers le nœud de poids le plus petit. Cet algorithme assure que tous les nœuds non détenteurs de jeton ont toujours un chemin vers au moins l'un des nœuds privilégiés. Chaque nœud choisit dynamiquement le voisin de poids le plus faible pour être son lien vers le détenteur de jeton.

Le poids de chaque nœud accompagne les messages envoyés par ce nœud. Toutes les requêtes ayant atteint le nœud privilégié sont traitées symétriquement avec la construction automatique du DAG; Les requêtes bloquées seront reroutées vers le prochain nœud privilégié.

Dans cet algorithme, il est possible pour un nœud d'être en section critique et de recevoir des requêtes. Dans ce cas, si le nœud détient plusieurs jetons, il satisfait immédiatement la requête reçue. Cependant, un nœud détenteur de jeton peut enfiler plusieurs requêtes en attente d'être satisfaites alors qu'un autre nœud demeure des jetons libres. Cette situation a été solutionnée en proposant de faire circuler les jetons libres dans le système, ce qui a donné naissance à une nouvelle variante de l'algorithme KRL, il s'agit de l'algorithme KRLF (K Reverse Link with token Forwarding). Dans ce denier, si un nœud a quitté sa section critique avec une file de requêtes vide, il envoie le jeton à l'un de ses voisins.

II.1. Hypothèses

Les hypothèses sur lesquelles est construit cet algorithme [60] sont:

- Les nœuds possèdent une identification unique dans l'intervalle [0..N-1];
- Les liens de communication sont bidirectionnelles et FIFO;
- La couche liaison assure que chaque nœud connaît ses voisins qui sont dans sa portée de communication et fournit des informations sur la formation et la rupture de liens;
- Les nœuds possédant au moins un jeton, ne tombent pas en panne;
- Le partitionnement du réseau est permis, tel qu'une portion du réseau doit contenir au moins un nœud qui possède un jeton et qui va continuer d'exécuter l'algorithme avec le sous ensemble de jetons dans la partition;
- Les déconnexions ne sont pas fréquentes.

II.2. Description de l'algorithme KRL

Structures de données: Chaque nœud maintient à son niveau les mêmes variables décrites dans l'algorithme RL présenté précédemment. Quelques variables ont été ajoutées ou adaptées telles que:
-*tokenHolder*: une variable booléenne indiquant si le nœud détient le jeton, initialement *tokenHolder* = TRUE pour tout i < k ;
- *totalTokens*: C'est le nombre de jetons possibles dans le système;
- *numTokens*: Compteur de jetons détenu par le processus; initialement, *numTokens* = 0 si i ≥ *totalTokens* sinon *numTokens* = 1;
- *next*: Indique la localisation du jeton c'est à dire le nœud voisin sur le chemin du jeton. Dans cet algorithme, la variable *next* est initialisée et mise à jour différemment: si le nœud i détient le jeton, *next*=i sinon, *next* vaut l'identificateur d'un nœud voisin ayant un lien sortant de i. Initialement, *next* = i si $0 \leq i < k$.

Déroulement des processus: Lorsque le processus d'application du nœud i formule sa requête pour entrer en section critique, le processus d'exclusion mutuelle insère le nœud i dans sa propre file de requêtes. A la réception d'une requête d'un nœud voisin de plus grand poids, le processus i insère cette requête en queue de file. Si i ne détient pas le jeton, il transmet la requête au voisin de plus petit poids. Quand i reçoit le message de jeton, il vérifie s'il se trouve en tête de file auquel cas, il permet à son processus d'application d'exécuter sa section critique sinon il envoie le jeton au processus en le supprimant du sommet.

Chaque nœud i modifie la valeur de son poids conformément à la *formule 1* (donnée dans la section précédente) à chaque réception de jeton de sorte que son poids devienne plus petit que le poids du nœud émetteur du jeton (pour notifier qu'il est privilégié).

Les nœuds non privilégiés doivent s'assurer qu'ils ont au moins un voisin dont le poids est plus petit. Si un nœud i n'a aucun chemin vers le nœud privilégié, il augmente son poids selon la *formule 2* (donnée dans la section précédente) de sorte que cette propriété soit vérifiée. Chaque fois qu'un nœud modifie son poids, il envoie un message *LinkInfo* à tous ses voisins.

Quand un lien d'un nœud i vers un nœud j est rompu ou inversé (changement de poids) le nœud i supprime la requête du nœud j de la file d'attente. Les requêtes ne sont pas perdues suite à cette opération pour une raison simple aucun nœud ne supprime sa propre requête de sa file, ainsi toutes les requêtes ont une chance d'être satisfaites.

Un nœud privilégié i doit avoir à tout moment au moins un voisin de poids plus fort, dans le cas contraire le nœud i diminue son poids de sorte qu'au moins un nœud voisin ait son poids plus fort que le sien.

II.3. Exemple de déroulement dans un réseau statique

Nous illustrons à travers la *figure 10* un déroulement de l'algorithme de K-exclusion mutuelle dans le cas statique c'est à dire que les nœuds ne sont pas en mouvement.

Dans la *figure 10.a*, les nœuds 2,3 et 4 ont envoyé une requête d'accès en section critique après avoir enfilé leurs propres requêtes. Le nœud 4 a envoyé sa requête au nœud 1 et les nœuds 2 et 3 ont envoyé leurs requêtes au nœud 0. La *figure 10.b* montre qu'après avoir envoyé le jeton au nœud 4, le nœud 1 est lui même demandeur d'accès en section critique. Le nœud 0 envoie aussi le jeton au nœud 3 avec un message de requête pour le compte de la requête qui lui reste dans sa file correspondant au nœud 2. Notons l'inversement des liens logiques entre les nœud 1 et 4 et les nœuds 0 et 3 avec diminution des poids relatifs aux nœuds 4 et 3 suite à la réception du jeton. La *figure 10.c* montre l'état du réseau après la sortie du nœud 4 de sa section critique, ce dernier envoie le jeton au nœud 1. Le nœud 3 après sa sortie de sa section critique envoie le jeton au nœud 0 qui le transmet au nœud 2.
La *figure 10.d* montre une nouvelle requête en provenance du nœud 4 qui a choisit le nœud voisin de poids le plus faible (nœud 2) à qui sa requête sera adressée; Un message *LinkInfo* est aussi adressé au nœud 1. La *figure 10.e* montre la réception du jeton au niveau du nœud 4 qui diminue son poids. Dans

la *figure 10.f*, le nœud 1 diminue son poids pour être le plus petit par rapport à tous ses voisins et pour assurer que les requêtes vont lui parvenir.

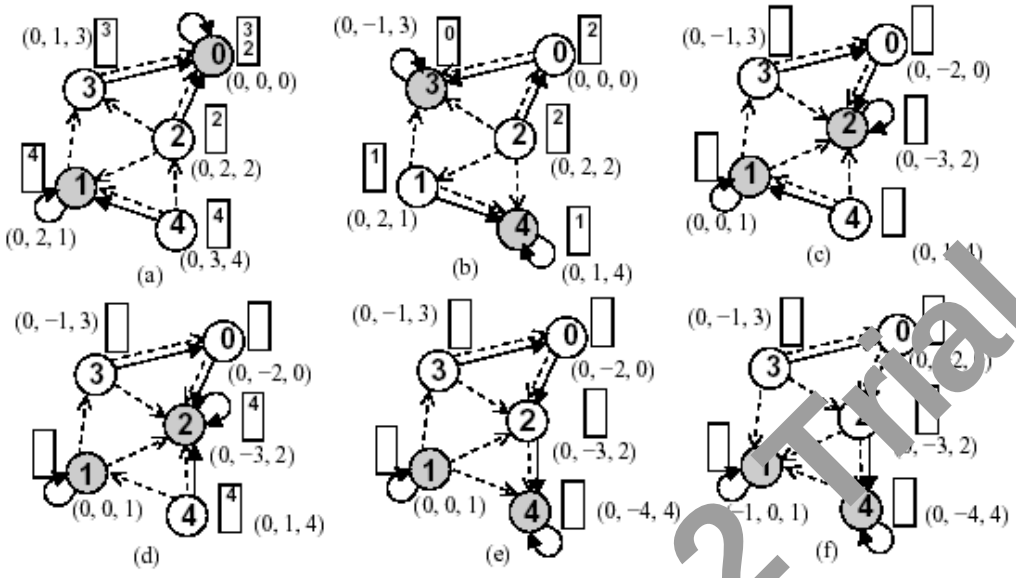


Figure 10 Exemple de déroulement dans un réseau statique (cas où K=2)

II.4. Exemple de déroulement dans un réseau dynamique

Maintenant, nous allons considérer le déroulement dans le cas dynamique où les formations de liens et déconnexions sont prises en compte.
La *figure 11.a* montre le même état que la *figure 10.a*. La *figure 11.b* montre la rupture du lien direct entre les nœuds 3 et 0 et les nœuds 3 et 2 après mouvement du nœud 3. Le nœud 3 a perdu son chemin vers le nœud privilégié et le nœud 0 a constaté la perte du lien avec le nœud 3. Le nœud 0 a supprimé le nœud 3 de sa file de requête, le nœud 2 n'a pas réagit puisqu'il n'a pas perdu son lien vers $next_2$ qui est le nœud 0. Le nœud 3 augmente son poids et envoie une requête au nœud 1 (*Figure 11.c*). La *figure 11.d* montre le système après l'envoi du jeton du nœud 0 au nœud 2 et l'envoi du jeton du nœud 1 au nœud 4 avec un message de requête pour le profit du nœud 3. Après avoir quitté sa section critique, le nœud 4 envoie le jeton au nœud 1 qui le transmet au nœud 3 (*Figure 11.e*).

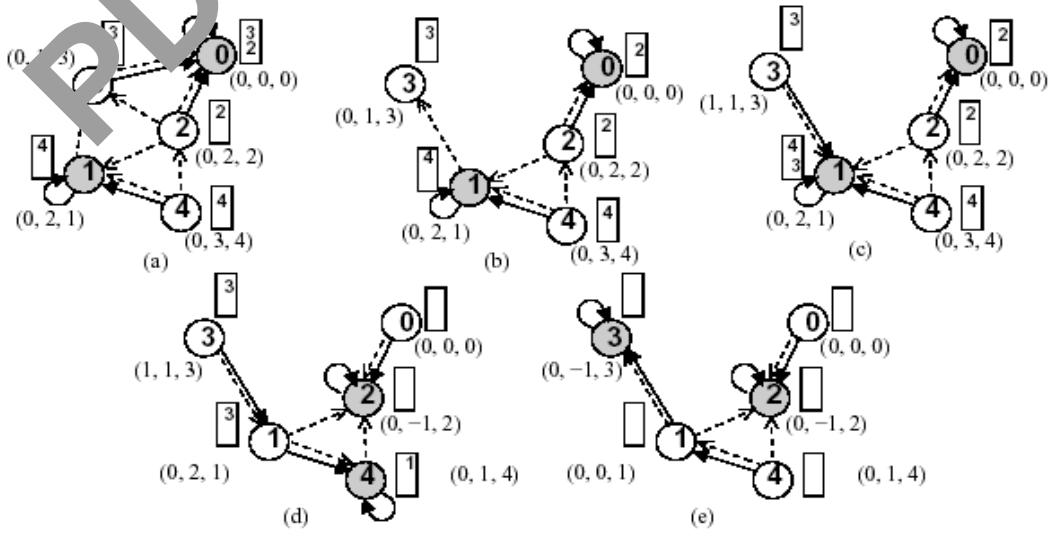


Figure 11 Exemple de déroulement dans un réseau dynamique (cas où K=2)

II.5. l'algorithme KRL avec envoi de jeton (KRLF) [61]

La *figure 12* montre l'exécution de l'algorithme dans lequel il y a deux nœuds qui possèdent le jeton (3 et 5). Les nœuds 4 et 6 soumettent leurs requêtes au nœud 5, le processus d'application de ce nœud est dans sa section critique et a inséré les requêtes des nœuds 4 et 6 dans sa file des requêtes. Le processus d'application au niveau du nœud 3 quitte la section critique, le nœud 3 n'envoie le jeton que lorsqu'il reçoit un message de requête, mais les nœuds 4 et 6 doivent attendre leur tour pour avoir le jeton utilisé par le nœud 5.

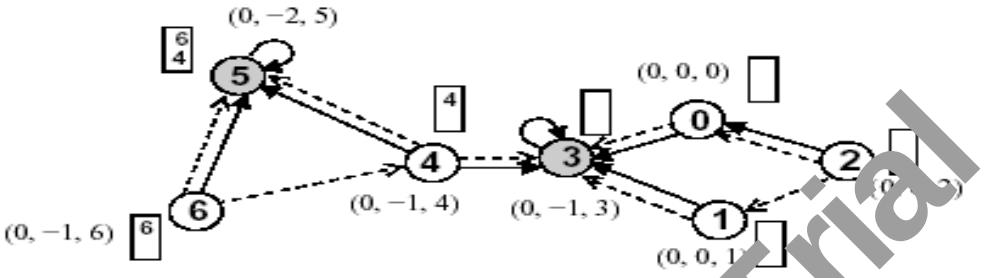


Figure 12 Problème de détention de jeton dans KRL

Pour remédier à cette situation, le nœud qui détient le jeton doit l'envoyer à d'autres nœuds du réseau s'il n'a aucune requête en attente. Pour cela, il choisit le nœud voisin dont le poids est le plus petit. Les nœuds gardent une trace de leurs voisins qui ont envoyé le jeton et qui ont une file de requêtes vide en marquant le lien comme visité.

La *figure 13* illustre ces modifications durant l'exécution de l'algorithme. Dans la *figure 13.a,* le nœud 3 a un jeton mais aucun voisin ne veut accéder en section critique. La *figure 13.b* montre l'état du système après que le nœud 3 ait quitté sa section critique et le jeton est envoyé à travers les nœuds 0, 2, 1 et 3 vers la partie gauche du réseau. La lettre V sur chaque lien signifie que celui-ci est marqué comme visité par le nœud émetteur et le nœud récepteur du jeton. Quand le nœud 3 reçoit le jeton et il a une file de requêtes vide, il marque tous ses liens comme non visités et recommence le processus d'envoi du jeton.

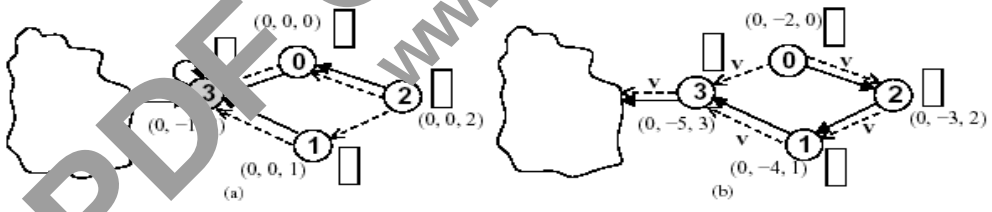


Figure 13 Opération d'envoi de jeton (Token Forwarding)

III. Conclusion

Nous avons présenté l'algorithme [60] de k-exclusion mutuelle dans les réseaux mobiles ad hoc. Cet algorithme est une généralisation de l'algorithme [43] qui, lui même est une adaptation d'une solution déjà existante dans les systèmes répartis car la structure logique de DAG a été déjà exploitée dans ces systèmes [5].

Sous les hypothèses posées au départ, cet algorithme assure l'exclusion mutuelle entre k+1 processus demandeur d'un exemplaire d'une ressource existant en K exemplaires, de sorte qu'au plus k processus peuvent exécuter leurs sections critiques.

Les résultats obtenus par la simulation [60][61] montrent que l'algorithme KRLF possède une meilleure performance en terme de temps de réponse par entrée en section critique en particulier

lorsque la charge (les demandes d'accès) augmente. Par contre, lorsque le système est sous une charge réduite L'algorithme KRLF est plus coûteux en terme de nombre de messages générés.

Toutes les requêtes sont satisfaites si les déconnexions ne se poursuivent pas car pour un déroulement défavorable où un nœud se déconnecte successivement de ses nœuds *next* après chaque formation et renvoi de requête, ce nœud risque d'être en état de famine (*starvation*).

En général, les solutions basées sur les jetons semblent être les plus appropriées pour le problème de K-exclusion mutuelle en environnement mobile ad hoc. En outre, l'utilisation des K-coteries est à écarter vu le coût de leur construction et maintien (jusqu'à preuve du contraire) qui s'ajoute au coût du service de k-exclusion mutuelle lui-même.

Chapitre 1 Conception de l'algorithme

I. Introduction

L'environnement mobile ad hoc possède des caractéristiques qui sont autant de paramètres supplémentaires à prendre en considération dans la construction des algorithmes distribués destinés à cet environnement. Les algorithmes d'exclusion mutuelle et de K-exclusion mutuelle existants s'inspirent des algorithmes d'exclusion mutuelle distribués, ils en sont soit une adaptation soit une amélioration ou bien une généralisation. Contrairement à ses prédécesseurs, notre algorithme est construit directement pour l'environnement mobile ad hoc.

Le seul algorithme connu [60] pour le problème fondamental de la K exclusion mutuelle dans l'environnement mobile ad hoc est celui présenté par Walter en 1998. Il est la généralisation d'un algorithme [43] du même auteur traitant l'exclusion mutuelle dans les réseaux mobiles ad hoc et qui s'inspire lui-même de la solution proposée par K.Raymond [5] pour l'exclusion mutuelle distribuée dans les réseaux statiques.

Dans son algorithme [60], Walter structure le réseau en '**D**irect **A**cyclic **G**raph' (DAG) dynamique (structure logique) maintenu grâce au poids affecté à chaque nœud tel que les nœuds soient dirigés du nœud de plus grand poids vers le nœud de poids le plus petit. Chaque nœud choisit dynamiquement le voisin de poids plus faible pour être son lien vers le nœud détenteur de jetons. Il existe K jetons dans le système répartis, initialement sur les K premiers nœuds du réseau. Une requête est acheminée du nœud demandeur vers le nœud privilégié dans le sens décroissant des poids des nœuds traversés. En réponse à la requête, le nœud détenteur de jetons envoie un seul jeton dans le chemin inverse; d'où le nom 'Revese Link'. Les évènements de formation et de rupture de liens sont pris en charge par la mise à jour des structures de données locales, notamment les poids des nœuds de telle sorte que la structure de DAG soit maintenue.

L'algorithme que nous proposons adopte une approche complètement différente de celle de Walter [60]. En effet, aucune structure logique n'est imposée sur le réseau, le système est composé de nœuds n'ayant aucune connaissance de la grandeur du réseau qui peut être variable. La couche liaison fournit à chaque nœud des informations sur la formation et rupture de liens. Il existe K jetons ($K > 0$) dans le système, initialement détenus par un seul nœud (le nœud d'identité 0). Notre algorithme est orienté recherche de jeton, d'où son nom 'Research'. Une requête est acheminée sur le réseau de la même manière qu'une exploration d'arbre, sa trace est sauvegardée de manière distribuée afin de pouvoir reconstituer son chemin qui va servir à véhiculer le jeton dans le sens inverse. Deux types de jetons ont été définis, Un jeton de type '*Single*' et un jeton de type '*Collector*'. Un jeton de type '*Single*' est un message classique de jeton véhiculé pour satisfaire une et une seule requête en attente. Un jeton de type '*Collector*' est un message qui comprend au moins un jeton. Il est envoyé par un nœud dont la file n'est pas vide dans le but de le faire revenir en collectant tous les jetons libres se trouvant sur son chemin. Seuls les nœuds entrain d'exécuter leurs sections critiques ou qui sont demandeurs d'accès sont habilités à construire une file d'attente.

Pour mettre en œuvre l'ensemble des principes sur lesquels est bâti notre algorithme, il est nécessaire de passer par deux outils de base qui consistent en l'utilisation:

- des **heuristiques**: Les heuristiques sont utilisées lorsqu'un nœud se trouve en face d'une situation de prise de décision. Ainsi, en fonction des informations collectées, de l'état local du nœud et face à une situation particulière, le nœud est capable d'entreprendre une action. Par exemple: A l'envoi d'une requête, il est préférable de l'acheminer vers un nœud qui vient d'être servi récemment (qui vient de recevoir un jeton). Ceci mettra la requête sur les traces du jeton, ce qui augmente ses 'chances' d'atteindre celui-ci le plus rapidement possible en moins de sauts. Il est donc nécessaire d 'enregistrer localement toutes les informations relatives aux envois de jetons à savoir l'heure d'envoi, le voisin destinataire, etc… Les informations maintenues localement ne nécessitent aucun message supplémentaire outre les messages de service de K exclusion mutuelle. Les heuristiques ne sont pas seulement utilisées pour le choix du chemin à prendre, mais peuvent servir à d'autres niveaux; par exemple: lorsqu'une requête est reçue par un nœud demandeur d'accès ou en section critique, en générale, la requête est enfilée. Néanmoins, l'algorithme se veut plus dynamique: S'il existe dans le système un grand nombre de jetons et que la durée de vie de la requête est jusque là relativement

petite, il serait préférable d'acheminer la requête au lieu de l'enfiler. D'une part, la requête est plus active car au lieu de rester dans une file d'attente au risque d'être détruite au bout d'un délai préfixé, elle va à la recherche d'un autre jeton, d'autre part, tous les jetons du système ont la même chance d'être sollicités.

- des **nombres premiers**: Les nombres premiers sont utilisés comme outil permettant de sauvegarder en une seule valeur la trace d'un chemin traversé par un message (ici le message de requête). En effet, chaque nœud possède un attribut propre unique sous la forme d'un nombre premier. Ainsi, l'algorithme calcule de manière distribuée le produit des attributs de tous les nœuds traversés par la requête. Le nombre obtenu à la fin du parcours admet une décomposition unique en produit de nombres premiers, ce qui permet d'avoir la liste de tous les nœuds ayant participé à l'acheminement de la requête. Ceci permet d'un coté d'éviter les chemins cycliques, de l'autre, aide à déterminer le chemin du jeton. Les nombres premiers ont déjà été exploités par M.Raynal [35] qui a démontré leur puissance quant à la détermination de l'état global d'un système distribué à l'instar des horloges logiques.

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

- Après la présentation des principes généraux de l'algorithme et des hypothèses, nous décrivons le système afin de dégager une vue globale de la stratégie adoptée et des choix effectués;
- L'algorithme sera ensuite présenté dans le détail en précisant les structures de données implémentées au niveau local ainsi que les actions entreprises à chaque évènement;
- Dans la section suivante, un déroulement de l'algorithme sera effectué tant pour le cas statique que pour le cas dynamique;
- Enfin, une discussion sur la stratégie globale et les choix adoptés permettra de préciser certains éléments de l'algorithme et d'en éclaircir d'autres en mettant en valeur les points forts et en indiquant les points faibles ainsi que l'apport de l'algorithme par rapport à ses prédécesseurs;
- Nous terminons par une conclusion.

II. Principes généraux de l'algorithme

Nous présentons un nouveau protocole de K exclusion mutuelle conçu pour l'environnement mobile ad hoc. Ce protocole tente de contracter un comportement intelligent qui essaie d'interagir avec le comportement dynamique du réseau. En effet, face à la même situation et sous les mêmes conditions locales l'action à entreprendre peut ne pas être toujours la même, cela dépend aussi des informations collectées. Par exemple: A la réception d'une requête d'un nœud j, si le nœud i se trouve en section critique ou est demandeur d'accès (ie en voie d'obtenir un jeton dans un futur proche), la politique générale de l'algorithme est telle que la requête est enfilée (ie mise en attente). Cependant, compte tenu de certaines informations, l'algorithme peut en décider autrement: la requête peut continuer son chemin en requête d'un autre jeton. Prenons à titre d'exemple les possibilités suivantes où l'algorithme peut réagir de plusieurs manières face à une situation précise telle que la réception d'une requête:

- Prise de compte d'informations locales:
  - Si le nœud i est demandeur d'accès sachant qu'il a effectué un certain nombre de relances pour la demande actuelle; ici plusieurs possibilités sont réalisables:
    - Ignorer ce fait (ie, enfiler la requête);
    - Supposer qu'à partir d'un certain nombre de relances, les chances d'acquérir le jeton diminuent et donc, il serait préférable de ne pas compromettre une autre requête (ie, la requête sera acheminée au lieu d'être enfilée);
  - Si la file contient déjà des requêtes en attente, plusieurs possibilités :
    - L'algorithme enfile la requête sans tenir compte de l'état de la file;
    - L'utilisateur peut définir un seuil limite en terme de nombre de requêtes pouvant être enfilées. Exemples:
      - Une seule requête est autorisée dans la file, les autres seront réacheminées; A la sortie de la section critique, le jeton sera envoyé au propriétaire de la requête enfilée;
      - Deux requêtes sont autorisées; À la sortie de la section critique, le jeton sera envoyé au propriétaire de la requête en sommet de file accompagné d'une indication de retour afin de pouvoir satisfaire l'autre requête;

- Plusieurs requêtes sont autorisées (avec un nombre limite fixé); À la sortie de la section critique, le jeton sera envoyé au propriétaire de la requête en sommet de file accompagné d'une indication de retour et d'une qualification supplémentaire, celle du 'C*ollector*'. Le jeton va non seulement revenir, mais aussi collecter tous les jetons qui peuvent éventuellement se trouver sur son parcours en nombre suffisant pour pouvoir satisfaire les requêtes restantes;

- Prise en compte des informations collectées:

- Le nombre de sauts effectués par la requête jusqu'au nœud i: il est possible de proposer ce qui 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 section critique, la requête doit être enfilée sinon réacheminée. Le nombre de messages moyen par entrée en section critique peut être une valeur fixée par l'utilisateur en fonction de sa connaissance 'personnelle' du réseau ou calculée localement (par le nœud i) en utilisant des informations relatives à d'anciennes requêtes satisfaites du nœud lui même;
- Le nombre de refus de traitement subit par la requête: C'est le nombre de nœuds en section critique ou demandeurs d'accès ayant refusé d'enfiler la requête. Ici, il est possible de définir un seuil de refus maximum à ne pas dépasser. Ce seuil peut être 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.

On constate ainsi que l'algorithme est paramétré dès qu'on a défini des fonctions qui délivrent un résultat (une décision) en faisant abstraction de son contenu qui peut être modélisé de plusieurs manières différentes.

Nous détaillons dans ce qui suit les principaux aspects qui ont permis le paramétrage de notre algorithme, ainsi que les axes sur lesquels il est bâti:

II.1. Utilisation des heuristiques

Dans cet algorithme, les heuristiques ont un rôle très important quant à la recherche du meilleur chemin et à la prise de décision et nécessitent une collecte d'informations fiables. En effet, chaque nœud du système ne connaissant que ses voisins directs, maintient des informations sur son historique, relatives à son activité locale et à sa relation avec son environnement. Par exemple:

- l'historique des envois de jetons, cette information peut servir dans deux situations différentes:
  - lors de la demande d'accès en section critique: la requête est envoyée vers le voisin qui a le plus récemment reçu un jeton du nœud i ;
  - lorsque le nœud i vient de transiter un jeton vers un voisin j après avoir acheminé une requête sur une certaine voie (autre que j sachant que j a déjà été exploré par la requête auparavant); si la requête revient vers le nœud i, celui-ci l'envoi vers le nœud j même si cette voie a déjà été explorée;
- le nombre de nœuds traversés par une requête: il peut servir à déterminer si un nœud en section critique ou demandeur d'accès est appelé à enfiler une requête reçue ou non (voir exemple de début du paragraphe II);
- le nombre de fois qu'un jeton est exploité avant d'atteindre son destinataire: il est possible pour un nœud i ayant reçu un jeton dont il n'est pas le destinataire de l'exploiter avant de le réacheminer vers son destinataire. Afin de ne pas risquer de mettre le destinataire en situation de famine, il est possible de définir des conditions d'utilisation d'un jeton; par exemple:
  - le nombre d'utilisations ne doit pas dépasser un seuil définit au départ,
  - le nœud i se trouve dans un état de demandeur d'accès avec un nombre R de relances effectuées (R>0, R tend vers un seuil de relance préfixé ou bien R > 30% du seuil de relance, etc…);
- Les informations de type '*LinkInfo*'; Si le nœud j (qui vient de se lier au nœud i) se trouve sur le même chemin que le nœud i d'une ou plusieurs requêtes, et si j se trouve à une position posj sur le chemin telle que: posj ≤ posi (posi et posj sont respectivement les positions relatives des nœuds i et j sur le chemin traversée par la 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;

- Le nombre de voisins; cette information peut servir dans le cas où il ne reste à un nœud i qu'un seul voisin, c'est son lien avec le réseau; Si le nœud i détient un ou plusieurs jetons, ceux-ci sont immédiatement envoyés vers le voisin afin d'assurer leur activité en cas d'isolement du nœud i;
- etc…

Ces informations sont collectées et/ou calculées de manière distribuée durant l'acheminement des messages de service de K exclusion mutuelle, elles ne nécessitent *aucune investigation* particulière. En outre, il n'existe pas nécessairement une structure de données pour chaque information, la plus part des informations sont *aussitôt obtenues aussitôt exploitées*. Ainsi, un nœud peut devenir autonome et prendre des décisions appropriées.

II.2. Utilisation des nombres premiers

Grâce à l'utilisation des nombres premiers, l'exploration du réseau à la recherche d'un jeton se fait de la même manière que l'exploration d'un arbre. En effet, l'attribut (valeur numérique en nombre premier) affecté à chaque nœud est utilisé pour calculer une variable T au fur et à mesure qu'une requête effectue des sauts. T est le produit des attributs de tous les nœuds traversés par la requête. Ainsi si l'attribut d'un voisin divise T, cela signifie que ce voisin a déjà été visité ce qui évite de le solliciter une nouvelle fois (chemin acyclique).

La valeur de la variable distribuée T admet une décomposition unique en produit de nombres premiers, ce qui permet de reconstituer la liste de tous les nœuds traversés par une requête. Cette liste est insuffisante pour déterminer le chemin exact de la requête et ainsi inverser ce chemin pour conduire le jeton. En effet, chaque nœud connaît sa position sur le chemin de la requête mais aussi la position de son prédécesseur. La connaissance des deux dernières informations (valeur de T et les deux positions sur le chemin) permet de tracer de manière distribuée le chemin du jeton dont la longueur est au plus, égale à celle du chemin de la requête.

II.3. Trois types de messages de jetons

Le système comprend K jetons initialement détenus par le nœud d'identité 0. Nous définissons trois types de messages de jetons: le type '*Single*' et le type '*Collector*' et un troisième type qui est un message de cession de jetons; Un jeton de type '*Single*' 'voyage' seul sur le réseau et n'est pas appelé à revenir au nœud source; il peut être exploité en cours de route par des nœuds intermédiaires (qui sont des demandeurs d'accès et n'ayant pas encore reçu leur jeton). Nous définissons ainsi un seuil d'utilisation pour ne pas léser le nœud destinataire de ce jeton. Dans certaines situations, principalement lorsqu'un nœud sort de sa section critique en ayant une file d'attente non vide, le nœud peut demander le retour de son jeton 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. Ainsi plusieurs jetons peuvent être véhiculés par un même message; c'est le type de jeton '*Collector*'. Un jeton collecteur permet au nœud source 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. 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 en état d'attente '*Need*'. Un jeton '*Collector*' doit effectuer deux passes. En effet, un nœud i qui doit envoyer un jeton en sa possession vérifie si sa file contient plus d'une requête. Dans le cas où le nombre de requêtes dans la file est égal à un le jeton prend la mention '*Single*' et est envoyé au propriétaire de la requête. Sinon, il envoie le jeton avec la mention '*Collector*' en précisant le nombre de jetons nécessaires à la satisfaction de toutes les requêtes se trouvant dans la file. Ainsi, Ce message de jeton a plusieurs missions: la première consiste à satisfaire le nœud dont la requête se trouve au sommet de la file, la deuxième est de revenir vers le nœuds i après avoir collecté tous les jetons stationnaires se trouvant sur son parcours afin de lui permettre de satisfaire les requêtes restantes. En réalisant ces deux passes aller et retour, le message de

jeton a aussi contribué à faire circuler les jetons stationnaires vers des 'endroits' où ils sont les plus sollicités.

Le troisième type de messages de jetons est un message généré par un nœud qui 'craint' son isolement dans un futur proche en ayant dans sa possession des jetons libres. Ce nœud étant lié au réseau par un seul lien, envoi ses jetons vers son unique voisin afin de permettre non seulement de les préserver d'une perte même momentanée, mais aussi leur circulation. Comme le message de jeton '*Collector*', un message de jeton de cession peut transporter plusieurs jetons simultanément.

Notre algorithme est orienté recherche de jeton et non circulation de jeton, un message de requête précède toujours un message de jeton sauf dans le cas où un nœud détenteur de jeton se trouve lié au réseau par un seul lien, dans ce cas les jetons seront transférés sans type au seul voisin (ni '*Collector*' ni '*Single*').

II.4. Les files d'attente

Seuls les nœuds demandeurs d'accès '*Need*' ou en section critique ('*CS*') ont habilités à construire une file d'attente et sont donc des cibles pour les requêtes: Les nœuds se trouvant à l'état '*Need*' sont probablement de futurs acquéreurs de jetons; Ceux qui sont en section critique vont dans un futur proche libérer la section critique et donc posséder un jeton libre; les autres ne sont pas intéressés par la section critiques, donc leur intervention n'est pas de figurer. La file est triée par ordre croissant des numéros de séquence des requêtes reçues '*NS*'.

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, nombre de sauts effectués par la requête, etc…) classées 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 temporairement, dans l'espoir que ce chemin soit renoué dans un futur proche (grâce à une formation de lien).

II.5. La cession de jetons

Cette opération vise à satisfaire un besoin spécifique et à rendre actifs les jetons stationnaires sans aucun coût supplémentaire.

Dans cet algorithme, 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 d'accès; 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 non encore atteint; 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.

II.6. La trace

C'est une structure de données maintenue au niveau de chaque nœud et ayant deux rôles: Le premier consiste à maintenir pour chaque requête le chemin traversé par celle-ci, le deuxième est de permettre l'optimisation de ce chemin en sauvegardant en plus tous les chemins possibles afin de choisir, lors de l'acheminement du jeton correspondant à la requête, le chemin minimal.

II.7. Les relances

Dans certaines situations extrêmes, un nœud ayant émis une requête peut attendre longtemps l'arrivée d'un jeton, mais celui-ci n'arrive pas. Ceci peut se produire dans deux cas possibles:

- La requête n'a pas abouti et ce pour plusieurs raisons possibles:
  - Cas d'isolement temporaire;
  - Cas de partitionnement temporaire;
  - La requête est insérée dans une file contenant un nombre considérable de requêtes tel que sa satisfaction soit impossible par manque de jetons collectés;
  - La requête tourne dans le réseau mais en vain car les ruptures et formations de liens se font tel que le jeton n'est jamais atteint;
- Le jeton a perdu le chemin du retour: Tous les liens sur les chemins possibles du jeton sont perdus suite à une ou plusieurs ruptures de liens.

Dans les deux cas précédents, il est possible de relancer la requête après un certain '*timing*', avec le même numéro de séquence pour garder son ancienneté dans le système. Nous pouvons définir un seuil de relance à partir duquel un '*échec*' est observé.

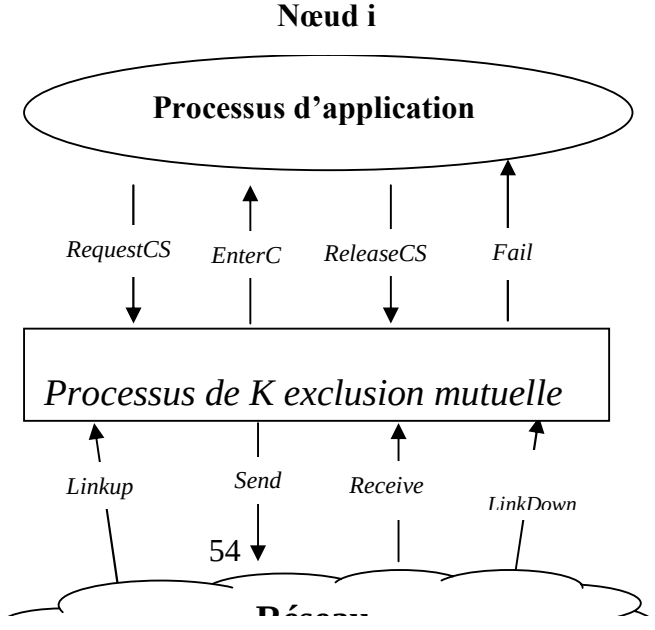
III. Hypothèses

On pose les hypothèses suivantes:

- Le système est composé d'un nombre variable de nœuds;
- Chaque nœud ne connaît que ses voisins directs et ne connaît pas la taille du réseau;
- Il existe K jetons (K valeur connue par tous les nœuds et K>0) dans le système;
- Un nœud ne peut pas lancer plus d'une requête à la fois;
- Aucun algorithme de routage n'est utilisé;
- La couche liaison fournit les informations sur la formation et la rupture de liens;
- Les canaux de communication sont bidirectionnels et FIFO et fiables (ie sans perte de messages);
- Les nœuds du réseau forment une topologie connexe;
- Pas de pannes de nœuds ni de perte de jetons ;
- Tolérer un partitionnement de durée limitée.

IV. Description du système

Le système est composé de nœuds identifiés chacun par un numéro unique séquentiel (0,1,2,3, etc…) et communiquant par échanges de messages à travers des liens sans fils. A chaque nœud i est affecté un attribut unique fixe qui représente le $(i+2)^{ième}$ nombre premier; Deux processus y sont implémentés (voir la figure1): un processus d'application et un processus de K exclusion mutuelle. La communication des deux processus détermine l'état '*Status*' du nœud aussi bien que la communication avec l'environnement. Celui-ci peut être dans l'état '*Free*' (n'est pas intéressé par la section critique), '*CS*' (en train d'exécuter sa section critique, '*Need*' (demandeur de jeton) ou bien '*Unable*' (demandeur mais isolé).



Le processus de K exclusion mutuelle au niveau d'un nœud i est déclenché par un ensemble d'évènements 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; Ces événements sont totalement ordonnés grâce à l'utilisation du mécanisme d'estampillage de L.Lamport[23].

Figure 1 Architecture du système

Les événements en entrée sont:
- ***RequestCS***: Le processus d'application demande d'accéder à la section critique; le statut du nœuds i se trouve ainsi à l'état '*CS*' s'il détient au moins un jeton, '*Need*' s'il ne détient aucun jeton (sa requête est acheminée) ou '*Unable*' s'il est isolé du reste du réseau;
- ***ReleaseCS***: Le processus d'application a fini l'exécution de sa section critique ce qui mettra le '*Status*' du nœud i à '*Free*';
- ***Receive***: Le processus i reçoit un message du processus j;
- ***Link_Up***: Notification de formation d'un lien, avec l'identification du nœud nouvellement lié. En fonction de son état, le processus i déclenche une suite d'actions;
- ***Link_Down***: Notification de rupture d'un lien; l'identification du nœud dont le lien est rompu est reçue.

Les évènements en sortie sont:
- ***Send***: Envoi de message; Trois types de messages peuvent être envoyés:
  - ***RequestCS***: Message de requête, il est de la forme RequestCs(Source,NS,Time_transfert,T, pos) où,

  - *Source* est l'identité du nœud source;
  - *NS* est le numéro de séquence de la requête, il représente l'heure locale à laquelle la requête est générée par le nœud source;
  - *Time_transfert* est l'estampille enregistrée chez l'émetteur de la requête en cours de transit;
  - *T* est le produit de tous les attributs des nœuds visités par la requête;
  - *Pos* est la position du nœud sur le chemin de la requête.
  - ***Token***: Message de jeton, il est de la forme Token (Source,Target, Ns,Type,Nb_token_need, Nb_token_grant, Pos, Pass, Util,Time,T) où

  - *Source*: est l'identité du nœud source, cette information est utile pour le cas de retour du jeton (le nœud source devient destinataire);
  - *Target*: est l'identité du nœud destinataire;
  - *Ns*: est le numéro de séquence de la requête pour laquelle ce jeton est envoyé;
  - *Type*: sert à qualifier le jeton '*Single*' ou '*Collector*';
  - *Nb_token_need*: est le nombre de jetons demandés à travers ce message, ce paramètre est utilisé pour le cas d'un jeton '*Collector*';
  - *Nb_token_grant*: est le nombre de jetons collectés par le message;
  - *Pos*: est la position du nœud sur le chemin du jeton, cette information est utile pour la mise à jour du chemin de retour du jeton;
  - *Pass*: détermine si le jeton est entrain de faire 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. Pass est mise à 0 s'il s'agit d'un jeton '*Single*';
  - *Util*: calcule de manière distribuée le nombre de fois qu'un jeton '*Single*' en transit est utilisé par les nœuds intermédiaires. Cette information est utile car elle permet de déterminer si le seuil est atteint auquel cas le jeton sera verrouillé;
  - *Time*: heure locale à laquelle le jeton est transféré, sous la forme d'un nombre entier;
  - *T*: est le produit des attributs de tous les nœuds par lesquels est passé un jeton. Cette information est utile seulement pour le cas où le jeton est de type '*Collector*' en première passe.

- ***Link_Info***: Envoyé à la formation d'un lien afin de déterminer si ce lien pourrait représenter un chemin optimisé pour l'acheminement d'un jeton; il est de la forme Link_Info(Tracei) où trace représente la structure de trace enregistrée au niveau de i ;

- ***Enter_CS***: Le processus d'exclusion mutuelle accorde la permission au processus d'application d'accéder à sa section critique; Avec cette primitive, seront sauvegardées selon le besoin toutes les informations relatives au jeton à exploiter telles que les circonstances d'obtention (détenu, cédé, exploité en cour de route, …), les informations véhiculées avec le message de jeton, etc…;
- ***Fail***: Le processus d'exclusion mutuelle informe le processus d'application de son incapacité à satisfaire sa demande.

V. Présentation de l'algorithme

1. Les structures de données

Chaque nœud i maintient à son niveau les structures de données suivantes:

a. Les constantes

- ***K***: Nombre total de jetons dans le système;
- A_i : attribut en nombre premier du nœud i;
- ***Max_autorised*** : Nombre maximum de jetons autorisés à 'voyager' ensemble.

b. Les variables

- Way_i:un entier indiquant l'identité du nœud sur le chemin du jeton. Il s'agit du nœud auquel le nœud i a envoyé un jeton le plus *récemment*; si $way_i = i$ cela signifie que le nœud i ignore le chemin du jeton;
- $status_i$: indique l'état du nœud i; il peut être '*Free*' (non demandeur), '*Need*' (demandeur), '*CS*' (en section critique), '*Unable*' (demandeur isolé);
- $Nb\_free\_token_i$: un entier >=0 indiquant le nombre de jetons libres détenus par le nœud i;
- Ns_i: Numéro de séquence de la dernière requête envoyée par le nœud i;
- $Time_i$: heure locale gérée selon L.Lamport[23];
- $File_i$: Une file de requêtes empilant dans l'ordre croissant des numéros de séquence les requêtes; Les informations suivantes sont sauvegardées avec chaque requête: *Source* est l'identifiant du nœud source; *Ns* est le numéro de séquence de la requête; *Time* est l'heure logique d'arrivée de la requête; *T* est la valeur de T reçue (c'est le produit des attributs de tous les nœuds traversés par la requête jusque là); *Type* est le type de jeton demandé et *Nb_token_need* est le nombre de jetons sollicités par la requête ; ce dernier champs ne sert que lorsque le nœud i insère dans sa file un message de jeton '*Collector*' qui a collectionné aucun jeton en cours de sa 'route' (voir paragraphe II.3);
- $File\_tmp_i$: Une file temporaire de requêtes, utilisée lors du traitement de requêtes dont le chemin du jeton est inconnu (impossible d'envoyer le jeton). Ces requêtes sont traitée lors de la formation d'un nouveau lien. A l'expiration du délai fixé ('*Valid_Timer*'), les requêtes seront supprimée.
- N_i: Une liste dont la structure d'enregistrement est composée de trois champs donnant des informations sur les voisins directs a savoir: *Idf* est l'identification du voisin; *A* est l'attribut du voisin et *Stamp* est l'estampille (sous la forme d'un entier) du dernier jeton envoyé à ce voisin.
- $Trace_i$: une liste maintenue par i et manipulée lors de l'acheminent des requêtes, chaque entrée de cette structure comporte les informations suivantes: *Source* est l'identificateur du nœud demandeur d'accès; *Idf* est l'identifiant du nœud émetteur de la requête; *Ns* est le numéro de séquence de la requête; *T* est le produit des attributs des nœuds déjà traversés par la requête; *Pos* est la position du nœud sur le chemin de la requête et *Prod*, 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 liste liée à '$Trace_i$' et qui 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 da '*Tracei*' (qui représente une requête) correspond à plusieurs entrées de '*S_Tracei'* dont

chacune représente un éventuel chemin qui servirait à acheminer le jeton de la requête correspondante. Cette structure est composée des champs suivants: *Idf* qui représente l'identificateur du nœud voisin sur le chemin optimal et *pos* est la position de ce nœud voisin sur le même chemin;
- R_i: est le nombre de relances de la dernière requête du nœud i.

2. Les procédures

- ***Init()***: procédure lancée à l'initialisation du système;
- ***Gen_Prem(i)*** : fonction permettant de générer le (i+3)ième nombre premier ;
- ***Delay()***: Fonction capable (selon l'état historique du réseau, le temps de satisfaction d'anciennes requêtes, etc…) de déterminer le temps nécessaire qu'un nœud ayant émis une requête, doit attendre pour considérer que sa requête (ou son jeton) a perdu son chemin et ainsi décider de relancer la requête, dans l'algorithme la fonction delay() est représentée par le paramètre '*Request-timer*';
- ***Valid_Timer(s)***: Fonction dont le résultat dépend de celui de la fonction '*Delay()*' et est capable de déterminer le moment à partir duquel le nœud i doit purger ses structures de données de la requête s;
- ***Seuil_File()***: fonction booléenne permettant à un nœud de 'décider' d'enfiler ou nécessaire acheminer la requête en fonction de plusieurs paramètres: la file est vide ou non, le nombre de nœuds traversés par la requête, le nombre de jetons dans le système et le nombre de nœuds traversés détenant le jeton, etc…;
- ***Seuil_Util()***: Fonction permettant de calculer le seuil d'utilisation d'un jeton en cours de route, il est déterminé en fonction du nombre d'utilisations autorisées, etc…;
- ***Seuil_R_File()***: Fonction booléenne qui permet à un nœud de 'décider' s'il doit enfiler une requête dans la file temporaire ou de la détruire en fonction de plusieurs paramètres (la durée de vie de la requête dans le système en termes de son heure d'enregistrement et le nombre de nœuds qu'elle a traversé, le nombre de fois où elle a été réenfilée, etc…);
- ***Seuil_abandon()***: Fonction booléenne qui permet de vérifier si le seuil de relances est atteint afin de déclarer au processus d'application un échec. Le seuil est atteint lorsque le nombre de relances maximal est atteint;
- ***Add_Queue(****Source,Ns,Time,T,Type,Nb_token_need****)***: procédure permettant d'insérer des requêtes dans la file d'attente dans l'ordre croissant des estampilles;
- ***Add_File_Tmp(****Source,Ns,Time,T,Type,Nb_token_need****)***: procédure permettant d'insérer des requêtes dans la file d'attente temporaire dans l'ordre croissant des estampilles;
- ***Add_Trace(****Source,Emetteur,Ns,T,Pos,Nb_token_need****)***: procédure permettant d'insérer la trace d'une nouvelle requête afin d'assurer l'acheminement du jeton sur cette trace;
- ***Add_Sub_Trace(****Emetteur,Pos****)***: procédure permettant d'insérer un nouvel élément dans la structure Stracei afin d'assurer l'existence d'un chemin supplémentaire et optimal pour le jeton;
- ***Add_Neighbor(j)***: A la formation d'un lien, cette procédure permet d'insérer le nouveau voisin;
- ***Change_Trace(****Source, p****)***: procédure permettant de modifier un ou plusieurs paramètres de la structure de trace en fonction de la valeur de p. exemple: Si p vaut 'All' toute l'entrée est modifiée;
- ***Purge_Queue(****Source****)***: procédure ayant pour rôle de supprimer une requête de la file d'attente;
- ***Purge_File_Tmp(****Source****)***: procédure ayant pour rôle de supprimer une requête de la file temporaire;
- ***Purge_Trace(****Source****)***: procédure ayant pour rôle de supprimer une requête de la trace;
- ***Purge_Neighbor(j)***:procédure appelée à la rupture d'un lien afin de supprimer le nœud éloigné de la liste des voisins;
- ***Transit_Request(***Source,NS,Time_transfert,T,pos***)***: Procédure ayant pour rôle d'acheminer une requête reçue par le chemin qui a le plus d'habilité à la mener vers un jeton; Ce chemin est en premier lieu un chemin qui a le plus récemment acheminé un jeton ou un chemin déjà visité en vain, mais qui vient juste d'acheminer un jeton;
-***Transit_Token(***Source,Target,Ns,Type,Nb_token_need,Nb_token_grant,Pos,pass,util,time,T***)***: Cette procédure permet, grâce au chemin de la requête de transmettre le jeton sur un chemin optimal en choisissant parmi les voisins diviseurs de T celui dont la position sur le chemin est la plus petite. Si ce chemin est perdu (suite à une rupture de lien) la requête est soit réenfilée, soit détruite: le choix est effectué grâce à une fonction Seuil_R_File() qui utilise des informations locales pour décider;

- ***Set-Request_Wait***: Cette procédure permet la création d'une structure temporaire dans laquelle une requête est placée en attente d'un jeton, elle est sollicitée en général lorsqu'un jeton reçu par un nœud i est appelé à être exploité par celui-ci, les données véhiculées par le jeton seront sauvegardées momentanément grâce à cette procédure.

3. Déroulement du processus de K exclusion mutuelle

Les traitements liés au processus K exclusion mutuelle sont déclenchés soit par un évènement externe tel que l'arrivée d'un message ou la formation/rupture de liens ou par un évènement interne tel que l'expiration d'un délai, demande d'un service de K exclusion mutuelle (la section critique), etc…, les évènements du système sont synchronisés par un mécanisme global basé sur l'horloge logique de Lamport [23]. Le déroulement du processus de K exclusion mutuelle s'effectue de la manière suivante:

- Initialisation du système

A l'initialisation du système, le nœud 0 génère localement K jetons dont il restera détenteur tant qu'il n'a pas été sollicité ou qu'il n'est pas en voie de perdre son dernier lien avec le réseau; auquel cas, il va léguer ses jetons à son voisin;

Chaque nœud i du système génère localement une constante Ai dont la valeur correspond au $(i+2)^{ième}$ nombre premier grâce à la fonction *Gen_Prem(i)*. La constante Ai servira d'attribut au nœud i; Un nœud i maintient localement la liste de tous ses voisins dans la structure *Neighbori* où il enregistre l'attribut Aj de chaque voisin; Cet attribut est calculé localement grâce à la fonction *Gen_Prem(j)*. Toutes les variables locales telles que $Status_i$, Way_i, $Nb\_free\_token_i$, etc… sont initialisées à ce niveau.

- Demande d'accès en section critique 'Request_CS'

Lorsqu'un nœud i désire entrer en section critique, s'il possède au moins un jeton libre ($Nb\_free\_token_i \geq 1$), il peut accéder à sa section critique. Dans le cas contraire, il peut se '*souvenir*' du dernier nœud qu'il a lui même servi précemment (grâce au champ *stamp* de la structure *Neighbori*) et lui demander le jeton. Rappelons que le champ *'stamp'* de la structure '*Neighbori(j)'* est mis à jour chaque fois que le nœud i achemine un jeton vers le voisin j. Si le nœud i n'a jamais acheminé de jetons précédemment, il envoie sa requête vers un voisin quelconque. Ce voisin peut être choisi selon un critère bien défini. Par exemple: '*le voisin dont l'identité est la plus petite*' ou bien sélectionné grâce à une fonction de randomisation parmi d'autres. Ainsi, en suivant le même principe, chaque nœud recevant la requête va l'acheminer jusqu'au nœud détenteur de jetons libres.

Si le nœud i a perdu tous ses liens avec le réseau (il est isolé, '*Unable*'), il attend la formation d'un nouveau lien pour relancer sa requête avec la même valeur de R_i (nombre de relances effectuées).

A l'envoi d'une requête, le processus de K exclusion mutuelle initialise un *timer* appelé *'Request_timer'* afin de déterminer le moment où la requête doit être relancée. Une fonction *Delay()* permet de calculer le délai d'attente avant de supposer une perte de chemin (de la requête ou du jeton) ou blocage de la requête dans la file d'attente, du à une rupture de lien. Auquel cas une nouvelle tentative de relance de la requête est effectuée. Un seuil de relance est aussi défini afin que la requête ne soit pas relancée à l'infini. Ayant atteint ce seuil, un échec *'Fail'* peut être observé pour le service de la requête.

- Expiration du délai Request_Timer 'Delay()'

A l'expiration du délai '*Request_timer*', le processus de K exclusion mutuelle vérifie si le seuil de relance est atteint; tant qu'il n'est pas atteint la requête sera relancée (avec incrémentation du nombre de relances effectuées R_i) avec le même numéro de séquence jusqu'à ce qu'elle soit satisfaite et ce, à condition que le nœud ne soit pas isolé. Dans ce dernier cas, un état '*Unable*' est marqué. Si le seuil de relance est atteint, une réponse '*Fail*' est communiquée au processus d'application.

- Réception d'une requête 'Request_Received'

Si un nœud i reçoit une requête, i peut être lui-même le propriétaire de la requête. S'il est toujours à l'état demandeur '*Need*', il tente de la réacheminer sur une autre voie. Si tous les voisins sont explorés, le nœud i attend l'expiration du délai '*Request_Timer*' relatif à la requête afin de pouvoir la relancer.

Si le nœud i n'est pas le propriétaire de la requête et détient au moins un jeton libre, il le transmet au nœud source (c'est-à-dire le propriétaire) de la requête. Si le nœud i se trouve en section critique ('*CS*') ou en voie de détenir un jeton ('*Need*') sans détenir de jetons libres et reçoit une requête, il détermine grâce à une fonction *Seuil_File()* s'il doit l'enfiler ou l'acheminer. La fonction *Seuil_file()* peut, grâce à un ensemble d'informations relatives à cette requête (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 i pour sa propre requête en cours (en cas de '*Need*'), etc…), déterminer si elle a plus de 'chances' d'être satisfaite en étant mise dans la file ou en allant chercher un autre jeton. Si la requête est enfilée, le *timer* '*Valid_Timer(s)'* est initialisé, s étant l'identité du nœud source, émetteur de la requête. Si la requête est à acheminer, une variable T calculée de manière distribuée est utilisée. Au niveau du nœud source (le nœud qui a généré la requête), T vaut la valeur de l'attribut affecté à ce nœud; Ensuite, T est multiplié par l'attribut de chaque nœud visité par 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; 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). On peut aisément voir à ce niveau, que la recherche du jeton se fait de manière arborescente.

Si un voisin est le destinataire d'un jeton à partir du nœud i après l'envoi d'une requête, si celle-ci retourne elle peut être acheminée vers ce voisin même s'il a déjà été exploré.

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.

Chaque requête qui passe un délai initialisé '*valid_timer(s)*' (s étant le numéro du nœud propriétaire de la requête) dans la trace, dans la file d'attente ou dans la file d'attente temporaire est automatiquement effacée afin d'assurer une mise à jour régulière des structures de données de tous les nœuds du réseau. Néanmoins, une ancienne requête peut être retournée après un certain nombre de sauts (c'est le cas d'une requête effacée des structures de données pendant qu'elle se trouvait en cours d'acheminement sur le canal): Celle-ci est ignorée.

- Expiration du délai Valid_Timer(s)

'*Valid_Timer(s)*' est le temps de séjour d'une requête relative au nœud s dans la file d'attente d'un nœud i. Expiré ce délai, toute trace de la requête est détruite: dans la structure *Trace*, dans la *file d'attente* et dans la *file d'attente temporaire*; Cette opération a un rôle important dans la mise à jour des structures de données locales aux nœuds. Il est à noter que le calcul de '*Valid_Timer*' doit tenir compte de celui de '*Request_Timer'*.

- Réception d'un jeton 'Token_Received'

A la réception d'un jeton, un nœud i doit distinguer plusieurs situations à savoir si le jeton lui est destiné, si le jeton est de type '*single*' ou '*Collector*' ou bien cédé par un nœud qui '*craint*' son isolement (c'est-à-dire qu'il est relié au réseau par un seul lien: son lien avec le nœud i) dans un futur proche, s'il est appelé à l'exploiter (en entrant en section critique ou en traitant son éventuelle file d'attente) ou bien s'il doit l'acheminer vers son destinataire ou 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 section critique et éventuellement traiter sa file dès sa sortie de la section critique. 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 section critique 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 grâce à la fonction *Seuil_Util()* 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 dernier cas, à sa sortie de la section critique, le nœud i doit réaacheminer 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*' (voir explications dans le paragraphe II.3), il faut déterminer s'il se trouve dans sa première ou sa deuxième passe (c'est-à-dire sur son chemin d'aller ou 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 reçu, s'il est à l'état '*Need*' et si le jeton est de type '*Collector*', si ce dernier 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 '*Trace*' la trace du passage de cette requête et notamment, la valeur de T à ce niveau ainsi que sa position. Le jeton est destiné à suivre le chemin inverse à celui emprunté par la requête auparavant, grâce à la valeur T. En premier lieu, le nœud détenteur du jeton, avant de l'envoyer, détermine si le nœud demandeur '*Target'* 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 (cela permet d'optimiser le chemin du jeton); et lui envoi le jeton après avoir divisé T par son propre attribut. 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. 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 (l'identité du nœud, sa position sur le chemin de la requête, etc…) sont enregistrées localement dans la structure '*S_trace*' 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 *'valid_Timer'*.

Si le jeton est de type '*Collector*' le même principe est appliqué pour l'aller et le retour. Sauf que dans l'allée, le chemin doit être inversé pour assurer le retour.

- Sortie de la section critique 'Release_CS'

A la sortie de sa section critique, le nœud i 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 elle 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 non seulement a 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 (mise à jour du paramètre '*util*' qui sera véhiculé par le 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 section critique, les évènements de formation et de rupture de liens peuvent se produire et être traités. Dans ce cas, à sa sortie de la section critique, 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 ses files d'attente.

- A la formation d'un lien 'Link_Up'

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, s'il se trouve sur le chemin d'au moins une requête, envoie sa structure de donnée *'Trace$_i$'* relative aux requêtes communes au nœud j. 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 ce nœud se trouve en 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 sa file temporaire et envoie le jeton.

Si ce nœud se trouve en état isolé et détient des jetons libres, ceux-ci sont acheminés vers le nouveau lien.

- A la réception d'un message 'Link_info'

Un message *'Link_Info'* est échangé à chaque formation de lien entre deux nœuds (i et j) s'ils se trouvent sur le même chemin d'au moins une requête. Un nœud i recevant ce message, procède à la mise à jour de sa trace ('*Trace*' et '*S_trace*') de sorte que si le nœud correspondant se trouve sur le même chemin d'une requête avec une position plus petite par rapport à ce qui existe déjà dans la trace de i, ce nœud y est rajouté. Il est à noter qu'un message *'Link_info'* contient des informations (identité du nœud source de la requête, position du nœud j sur le chemin de la requête, etc…) sur les requêtes communes seulement. Cette opération aide à minimiser le chemin du jeton.

- Rupture d'un lien 'Link_Down'

Dès qu'un nœud i '*constate*' 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. Le nœud i procède alors à la mise a jour de sa variable Way.

4. L'algorithme

L'algorithme de K-exclusion mutuelle que nous avons proposé sera présenté dans cette section sous la forme d'un ensemble de procédures déclenchées soit par appel ou par l'arrivée d'un évènement. Tous les traitements sont sans interruption. Cependant, lors de l'exécution de la section critique, un évènement peut survenir.

- Initialisation du système

```
Procédure init()
{
        Waiting_i = File_i = File_tmp_i = trace_i = nil          //structures de données internes
        R_i = 0
        A_i = Gen-Prem(i)   // génère le (i+1)ième nombre premier
        way = i
        status_i = 'Free'
        time_i = 0
        to_send_i = False
        If i = 0 then
                If  (| Ni | = 1) then
                        inc (Timei)
                        Send TOKEN(i,Ni.idf,0,K,0,'',0,0,Timei, A_i) to Ni.idf
                        Nb_free_tokeni = 0
                        Wayi = Ni.idf
                else
                        Nb_free_token_i = K
```

```
…Suite
        Pour tout j ∈ Ni
        {
                Ni.attribut = Gen-Prem(j)
                If j = 0 then
                        Wayi = j
                FI
        }
}
```

- Lors de la demande d'accès en section critique'Request_CS'

```
{
If  Nb_free_tokeni >= 1 then
        Statusi = 'CS'
        Dec(Nb_free_tokenii )   //décrémente le nombre de jetons
        Enter_CS
else
        If  ({∃ Ni.idf / Ni.idf = wayi or Ni.Stamp is max}≠ Φ) then  //On envoi vers way ou un voisin dont le stamp est maximum
                Statusi = 'Need'
                Inc(timei)
                Nsi = timei
                Timer_on(Valid_Timer(i))
                Add_trace(i,i,timei,Ai,0, 1)
                timer-On(Request-timer)
                Send REQUEST(i, Nsi, timei, Ai, 0) to Ni.idf
        Else
                //le nœud est isolé et attend la formation d'un lien
                timer-On(Request-timer)
                statusi = 'Unable'
        FI
FI
}
```

- A l'expiration du délai Request_timer ' Delay()'

```
{
If seuil_abandon() then  //seuil à partir duquel on constate un échec
        Purge_trace(i)
        Statusi='free'
        wayi=-1
        Fai
Else    //tentative d'une nouvelle relance
        If  (∃ j € {Ni.idf / Ni.idf = wayi or Ni.stamp max}  then
                Inc(timei)
                Inc(Ri)
                Change_Trace(i,Ai)  //ici on ne change que la valeur de T
                timer-On(Request_timer)
                send REQUEST(i,Nsi, timei,Ai, 0) to j
        Else
                //le noeud est isolé et attend la formation d'un lien
                timer-On(Request-timer)
                statusi = 'unable'
        FI
FI
}
```

- Lors de la réception d'une requête 'Request_received'* d'un nœud j *de la forme:
RequestCs(Source,NS,Time_transfert,T,pos)

```
{
Timei = max(Timei, Time_transfert) + 1
If (source = i)          //retour de la requête
        If Ns = NSi then
                If  Statusi = 'Need' then    //Requête non encore satisfaite; réacheminée sur une autre voie
                        Change_Trace(Source,T)  //ici on ne change que la valeur de T
                        Transit_Request()
                FI    //Else cette requête a été satisfaite ou en CS grâce à un jeton en transit
        FI     // Else il s'agit d'une ancienne requête
Else
        Treat = True          //initialement la requête n'est pas à ignorer
        If (Source € tracei ) then
                If (T mod Ai ≠ 0) then
                        If (Ns = tracei(Source).Ns) then     //c'est une relance
                                Change_trace(Source,All)
                        Else                    //nouvelle requête
                                If (Ns >  Tracei(Source).Ns)  then //nouvelle requête, l'ancienne n'étant pas purgée
                                        Purge_Queue(Source)
                                        Change_Trace(Source,All)
                                FI
                        FI
                Else
                        If (Ns <Tracei(Source).Ns) then  //ancienne requête
                                Treat = False          //la requête doit être ignorée
                        Else     //retour d'une requête en cours
                                Change_Trace(Source,T)  //On ne change que la valeur de T
                        FI
                FI
        Else
                If (T mod Ai ≠ 0) then          //1ère arrivée de la requête
                        Timer_on(Valid_Timer(source))
                        Add_Trace(Source,Ns,T*Ai,Pos+1)
                Else     //Else retour d'une requête purgée
                        Treat = False          //la requête doit être ignorée
                FI
        FI
        If Treat then          //la requête doit être traitée et non ignorée
                If (Nb_free_token(i) = 1) then
                Transit_Token(final,i,Source,Ns,1,1,0,'Single',0,0,timei,T)
                Else
                to_sendi = False
                If (statusi = 'CS' or statusi = 'Need') then
                        If (seuil_file())  //voir s'il y a possibilité d'enfiler
                                Timer_on(Requestsource_timer)
                                Add_Queue(source,Ns,Timei,T,'Single',1)
                        Else
                                //la requête est à acheminer
                                to_sendi = True
                        FI
                Else     //la requête est à acheminer
                        to_sendi = True
                FI
                If (to_sendi) then  //la requête n'est pas enfilée mais à acheminer
                        to_sendi = False
                        Transit_Request()
                FI
                FI
        FI
FI
}
```

- A l'expiration du délai 'Valid_Timer(s)'

```
{
        Purge_trace(s)
        Purge_queue(s)
        Purge_File_Tmp(s)
}
```

***- Lors de la réception d'un jeton 'Token_Received' d'un noeud j de la forme*:**
Token(Source,Target,Ns,Nb_token_need, Nb_token_grant,Pos,Type, pass, util,Time ,T)

```
{
timei = max(timei,Time) + 1
Nb_free_token = Nb_free_token + Nb_token_grant
If (Target = i) then
        If ((type = 'collector') and (pass= 1)) then  //c'est un collector appelé à être renvoyé
                If (Nb_token_grant>1) then
                        If ((Statusi='Need') or (filei≠Φ) or (waitingi≠nil))  then    //un jeton sera exploité les autres retournés
                                Nb_free_tokeni = Nb_free_tokeni - Nb_token_grant
                                Transit_Token(filei,i,Source,Nb_token_need,Nb_token_grant-1,0,'Collector',2,util,timei,Ai)
                        Else
                                Transit_Token(filei,i,Source,Nb_token_need,Nb_free_token,0,'Collector',2,util,timei,Ai)
                                Nb_free_token = 0
                        FI
                Else   // Nb_token_grant = 1
                        Add_Queue(Source,Ns,Time,T,'Collector',Nb_token_need)
                FI
        FI
        If ((Statusi = 'Need') or (typei='' and Statusi='Unable')) then
                timer _off(Request_timer)
                Statusi = 'CS'
                Purge_Trace(i)
                Enter_CS
        Else
                If (waiting ≠ nil)      //acheminer le jeton vers le nœud en attente
                        If (waiting.Type) = 'Collector' then
                            Transit_Token (filei,i,Waiting.idf, Waiting.Ns,Waiting.Nb_token_need,1,0,'Collector',2,0,Timei,T)
                        Else
                            Transit_Token (filei, Waiting.idf, Waiting.Ns,1,1,0,'Single',0,util+1,Timei,T)
                        FI
                Else
                        If (filei ≠ Φ) then
                                If (filei.Type = 'Single') then  //il s'agit d'une requête
                                        Transit_Token(filei,i,filei.idf, 1, 1, 0,'Single',0,0,Timei,filei.T)
                                Else
                                        Transit_Token (filei,i,filei.idf,1,1,0, 'Collector',2,0,Timei,filei.T)
                                FI
                        Else
                                1
                        FI
        FI
Suite …
```

```
… Suite
Else   //i n'est pas le destinataire du jeton, va t-il l'exploiter?
        If (Statusi = 'Need') then
                If (Nb_token_need < Nb_token grant) then  // il s'agit d'un 'collector' en voie de céder un de ses jetons
                        Nb_free_tokeni = Nb_free_token – Nb_token_grant
                        If (pass = 1)           //maj de la trace pour assurer le retour
                                Change_trace(Source,Pos+1,T)   //Trace.Idf = Source
                                Pos = pos+1
                                T = T * Ai
                        FI
                        Transit_Token(filei,Source,Target,Ns,Nb_token_need,Nb_token_grant-1,Type,pass,util,Timei,T)
                        timer _off(Request_timer)
                        Statusi = 'CS'
                        Purge_Trace(i)
                        Enter_CS
                Else
                        If (Type = 'Collector')  //tous les jetons seront retransmis
                                Nb_free_tokeni = Nb_free_tokeni – Nb_token_granti
                                If (pass = 1)           //maj de la trace pour assurer le retour
                                        Change_trace(Source,Pos+1,T)   //Trace.Idf = Source
                                        Pos = pos+1
                                        T = T * Ai
                                FI
                                Transit_Token(filei,Source,Target,Ns,Nb_token_need, Nb_token_grant,Type, pass, util,Timei,T)
                        Else
                                If (Seuil_Util(util)) then
                                        Set-Request_Wait   //mise en attente du destinataire dans la structure Waiting
                                        timer _off
                                        Dec(Nb_free _tokeni)
                                        Statusi = 'CS'
                                        Purge_Trace(i)
                                        Enter_CS
                                Else  //transmission du jeton sans exploitation
                                    Transit_Token(filei,Source,Target,Ns,Nb_token_need,Nb_token_grant,Type,Pass,util,
                                Timei,T)
                                FI
                        FI
                FI
        Else    //pas de requête locale en cours, doit acheminer le jeton; i n'est qu'un transitaire
                If (type = 'collector') then
                        Nb_token_grant = Nb_free_tokeni
                        Nb_free_tokeni = 0
                        If (pass = 1) then
                                Change_trace(Source,Pos+1,T)   //Trace.Idf = Source
                                Pos = pos+1
                                T = T * Ai
                        FI
                FI
                Transit_Token(filei,Source,Target,Ns,Nb_token_need,Nb_token_grant,Pos,Type,Pass,util,Time,T)
        FI
FI
}
```

- A la sortie de la section critique ' Release_CS':

```
{
        Status = 'Free'
        Inc(Nb_free_i_tokeni)
        Ri=0
        #déterminer le type de jeton exploité ainsi que les informations qui l'accompagnent (destinataire, type de cession, …)
        If (Target ≠ i) then    // ce nœud vient d'exploiter un jeton dont il n'est pas le destinataire,les informations relatives au jeton
(Source,Target,Ns,Nb_token_need,…) étant sauvegardées dans des structures temportaires à l'entrée de la section critique
                If (waiting_i ≠ Nil) then          //acheminer le jeton vers l'élément en attente
                        Transit_Token(filei,Waiting.Source,Waiting.Target,Waiting.Ns,Waiting.Nb_token_need,
                Waiting.Nb_token_grant,Waiting.Pos,Waiting.Type,Waiting.pass,Waiting.util+1,Waiting.Time,Waiting.T)
                Else    //acheminer le jeton vers l'élément de sommet de file
                        if (filei ≠ Φ) then
                                If (Filei.type ≠ 'Collector') then      //demander un 'Collector'
                              Transit_Token(filei,i,filei.idf,filei.Ns,min(|filei|,Max_autorised),1,0,'Collector',1,0,Timei,Ai)
                                Else  //on envoi le jeton collector demandé
                                        Dec(Nb_free_tokeni)
                                        Transit_Token(filei,i,filei.idf,filei.Ns,filei.Nb_token_need,0,'Collector',2,0,timei,
                                filei.T)
                                FI
                        Else
                                If  ((| N_i | = 1) and (Nb_free_token > 0)) then //c'est le cas ou i étant en CS, il ya rupture de lien
                                        Send TOKEN(i,N_i.idf,0,0,Nb_free_tokeni,0,'',0,0,Timei_i,Ai) to N_i.idf
                                        Nb_free_token_i = 0
                                        Way = Ni.Idf
                                FI        //Else le jeton est gardé
                        FI
                FI
        Else
                If (filei ≠ Φ) then
                        If (filei.type = 'Collector') then //mettre la requête dans la structure Waiting
                                Set-Request_Wait
                        FI
                        Transit_Token(filei,i,filei.idf,filei.Ns,min(| filei |, Max_autorised),0,0,'Collector',1,0, Timei,Ai)
                Else
                        If  ((| N_i | = 1) and (Nb_free_token > 0)) then //c'est le cas ou i etant en CS, il ya rupture de lien
                                Send TOKEN(i,N_i.idf,0,0,Nb_free_tokeni,0,'', 0,0,Time_i,Ai) to N_i.idf
                                Nb_free_token_i = 0
                                Wayi = Ni.Idf
                        FI        //Else le jeton est gardé

                FI
        Fi
}
```

- A la formation d'un lien avec le nœud j 'Link_Up'

```
{
        Add_Neighbor(j)
        If (Trace_i ≠ Nil and {∃ Tracei.source/ Tracei.source=j}) then
                Send Link_Info(Trace_i)          //On envoi l'entrée correspondante
        FI
        If (Status_i = 'Unable') then   //cas d'un noeud demandeur mais isolé
                timer _off(Request_timer)
                Start_request()       //appel à la procédure de demande d'entrée en section critique
        FI
        If ((Nb_free_token_i >0) and (File_tmp_i ≠ Φ)) then //traitement des requêtes de la file temporaire
                If (File_tmp_i(1).Type = 'Collector') then
                        Transit_Token(File_tmp_i,i, File_tmp_i(1).Idf,file.Ns,file.Nb_token_Need,1,'Collector',2,0,Timei,Ai)
                Else
                        Transit_Token(File_tmp_i,i, File_tmp_i(1).Idf,file.Ns,1,1,'Single',0,0,timei,Ai)
                FI
        FI
        If  ((| N_i | = 1) and (Nb_free _tokeni > 0)) then    //le noeud était isolé avant ce lien, il cède ses jetons à son nouveau lien
                Send TOKEN(i,j,0,0,Nb_free_token,0,'', 0,0,Time_ii,Ai)  to N_i.idf
                Nb_free_token_i = 0
                Wayi=j
        FI
}
```

- Réception d'un message 'Link_info(trace)'

```
{
Pour toute entrée de Tracei
Faire
        Pour toute entrée de Trace
        Faire
                If (Tracei.Source = Trace.Source ) then
                        If  ((Trace.Idf € Ni) and (tracei.pos > trace.pos) then
                                Add_Sub_Trace(Trace.Idf,Trace.Pos)
                        FI
                        Exit
                FI
        Fait
Fait
}
```

- A la rupture d'un lien avec un nœud j 'Link_Down'

```
{
        Purge_Neighbor(j)
        If  ((| Ni | = 1) and (Nb_free_tokeni > 0)) then
                Send TOKEN(i,Ni.idf,0,0,Nb_free_token,0,'',0, 0,Timei,Ai)  to Ni.idf
                Nb_free_tokeni = 0
                Wayi = Ni.Idf
                Ni.stamp=Timei
        FI
        If (Wayi = j) then
                If (il existe h € {Ni.Idf/ Ni.Stamp = max and Ni.Stamp ≠ 0} then
                        Wayi = h
                Else
                        Wayi = i
                FI
        FI
}
```

- Procedure Transit_Request(source, Ns,Time, T, pos)

```
{
        If (T mod Ai ≠ 0)  then    //première arrivée de la requête
                T = Ai*Tre
        Fi
        If (∃j € {Ni.idf / Ni(j).Idf=source ou Ni(j).A divise T et Ni(j).A divise Tracei(j).prod ou Ni(j).Stamp est max}) then
                Inc(Timei)
                Send REQUEST(source, Ns,Timei, T, pos+1) to j
        Else
                If (source ≠ i) and (Tracei(source).Idf € Ni)) then   //Retour de la requête au prédécesseur
                        Inc(Timei)
                        Send REQUEST(Source,Ns,Timei,T/Ai, 0) to Tracei(source).Idf
                FI    // Else attendre l'expiration du délai pour relancer la requête
                Purge_trace(source)
        Fi
```

- Procedure Transit_Token(file,Source,Target,Ns,Ntn,Ntg,Pos,Type,Pass,Util,Time,T) to j

```
{
etiq :    If (∃j ∈ {Ni.idf/ Ni.Aj divise tracei(j).T and tracei(j).pos est min}) then
              If (Target = waiting.idf) then waitingi = nil FI
              inc(Timei)
              Ni(j).stamp = Timei
              Wayi = Ni(j).idf
              Dec(Nb_free_tokeni)
              Send TOKEN(Source,target,Ns,Nb_token_need,Nb_token_grant,Pos,Type, Pass,util,Timei,T/Ai) to j
              If (pass ≠ 1) then
                    Purge_trace(Target)
                    Purge_Queue(Target)
              FI
              If (Tracei ≠ Nil) then   // maj de prod pour  un acheminement optimal des requêtes suivantes
                    Pour toute entrée e de tracei
                    Faire
                          Tracei(e).Prod = Tracei(e).Prod*Ni(j).A
                    Fait
              FI
        Else   //impossible d'acheminer le jeton
              If (seuil_R_File()) then
                    Add_File_Tmp(Source,Ns,Timei,T,Type,Ntn)
              Else       //la requête est détruite
                    Purge_trace(Target)
                    Purge_Queue(Target)
              FI
        FI
        If  ((Nb_free_tokeni >0) and (file ≠ Φ)) alors
                    # soit idf l'élément de waiting ou de sommet de filei, définir les paramètres d'envoi
                    Aller à etiq     //pour traiter waiting ou file idf
              FI
        FI
}
```


VI. Exemple de déroulement de l'algorithme dans un cas de réseau statique

Dans cette section, un exemple illustratif du fonctionnement de notre protocole dans un cas de réseau statique est décrit. Le scénario permet de clarifier le comportement général de l'algorithme et 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… Supposons une topologie avec 8 nœuds que nous considérons sans mouvement. Le nœud 0 détient 3 jetons (figure 2.a) et la technique de recherche appliquée dans le cas où la valeur de Way (prochain nœud sur le chemin du jeton) n'est pas positionnée sur un nœud voisin, est d'adresser le voisin dont l'identité est la plus petite (cette politique est choisie à titre d'exemple, une autre politique aurait pu être utilisée sans problèmes).

La figure 2.b montre le nœud 3 et le nœud 7 formuler une requête qui, n'ayant pas de chemin préliminaire, elles sont respectivement adressées au nœud 1. Le nœud 1 choisi parmi ses voisins celui qui a la plus petite identité et lui adresse les deux requêtes dans l'ordre de leur arrivée c'est à dire 3 ensuite 7. Le nœud 3 ne dispose d'aucun jeton, il retourne les deux requêtes dans l'ordre au nœud 1. D'après la figure 2.c le nœud 1 achemine la requête de 3 sur le chemin 1,5,4,0 et la requête du nœud 7 vers le nœud 3 car il a la plus petite identité parmi les voisins non visités par cette requête. Entre temps le nœud 1 désire entrer en section critique et formule une requête qu'il adresse au nœud 2. Cette requête lui revient et l'adresse au nœud 3. Le nœud 3 (figure 2.d) étant à l'état '*Need*', enfile d'abord la requête du nœud 7 ensuite celle du nœud 1. La figure 2.e montre en bleu 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. Mais le nœud 4 est le voisin direct du nœud 3, c'est ainsi que le chemin 0,4,3 est préféré. Le nœud 3 entre en section critique. La figure 2.f montre le nœud 3 après sa sortie de la section critique. Il envoie le jeton au nœud se trouvant au sommet de sa file (ie 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 section critique. Entre temps le nœud 6 désire entrer en section critique, envoie sa requête au nœud indiqué par sa variable Way (valeur positionnée à l'initialisation du système). La figure 2.g montre le nœud 7 après sa sortie de la section critique, il 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 section critique. Le nœud 6 reçoit aussi son jeton et entre en section critique.

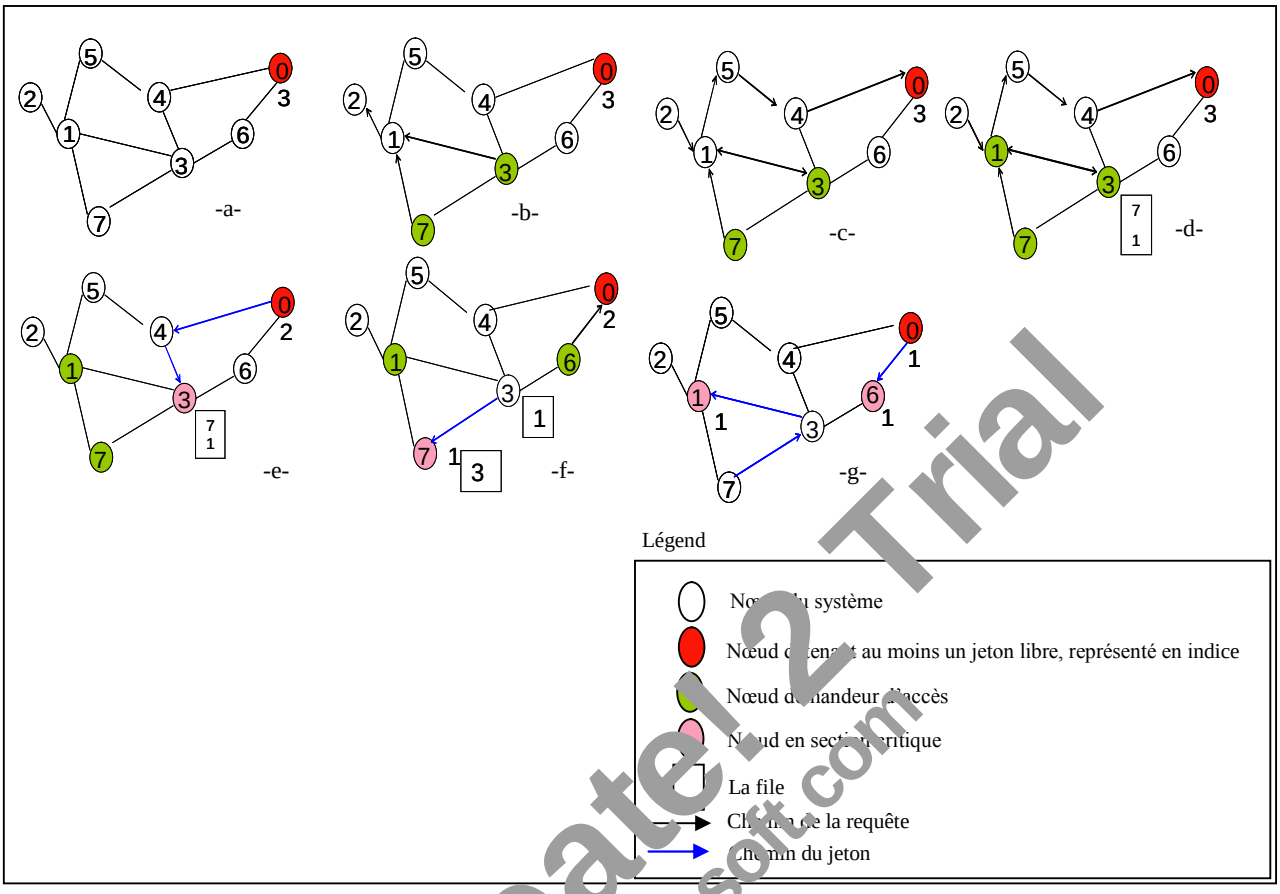


Figure 2 Illustration d'un déroulement de l'algorithme Ksearch (cas statique avec 3 jetons)

VII. Exemple de déroulement de l'algorithme dans un cas de réseau dynamique

A l'état initial (figure 3.a), nous allons considérer le même réseau que pour l'exemple précédent et un scénario de demandes d'accès à la section critique identique avec une certaine mobilité des nœuds afin de voir l'influence de la mobilité sur le cours des opérations.

La figure 3.a montre plusieurs évènements qui se sont succédés: le nœud 3 et le nœud 7 formulent respectivement une requête qui, n'ayant pas de chemin préliminaire, elles sont adressées au nœud 1. Le nœud 1 choisi parmi ses voisins celui qui a la plus petite identité (le nœud 2) et lui adresse les deux requêtes dans l'ordre de leur arrivée c'est à dire 3 ensuite 7. Entre temps, le lien entre le nœud 3 et le nœud 4 est rompu suite à l'éloignement du nœud 4. Le nœud 2 n'ayant pas de jetons libre renvoie la requête de 3 ensuite celle de 7 vers le nœud 1. Le nœud 1 achemine la requête du nœud 3 sur la voie 1,5,4,0 et celle du nœud 7 vers le nœud 3 qui, étant demandeur d'accès insère la requête dans sa file d'attente. Dans la figure 3.c, le nœud 1 désire entrer en section critique, il formule une requête qu'il adresse au nœud 2. Celui-ci, la renvoi faute de jetons. Le nœud 1 décide alors d'envoyer sa requête vers le nœud 3. La figure 3.d montre le nœud 0 envoyant vers le nœud 3 un jeton qui sera intercepté par le nœud 1 et sera exploité par celui-ci puisqu'il est demandeur d'accès (en considérant que le jeton peut être utilisé au moins une fois en cours de route). Le nœud 1 entre en section critique. Dans la figure 3.e le nœud 6 s'éloigne du réseau en perdant tous ses liens. Le nœud 0 va céder ses jetons au nœud 4 puisqu'il est son seul lien avec le réseau. Le nœud 6 désire entrer en section critique mais il a perdu tous ses liens, il se met à l'état '*Unable*' avec l'espoir qu'un lien vienne se former. En effet, dans la figure 3.f le nœud 6 va se rapprocher du nœud 4 et lui formule sa requête. A sa sortie de la section critique, le nœud 1 achemine le jeton sur la voie qui lui

était destinée au départ c'est-à-dire en direction du nœud 3. Le nœud 3 entre en section critique. Dans la figure 3.g, le nœud 3 sort de sa section critique. Ayant 2 requêtes dans sa file d'attente, il envoie un jeton '*Collector*' au nœud 7. Le nœud 7 entre en section critique en insérant dans sa file la requête du nœud 3 (c'est le jeton qui est considéré comme tel). Le nœud 6 ayant reçu un jeton du nœud 4 entre en section critique. A sa sortie de la section critique (figure 3.h), le nœud 7 retourne le jeton vers le nœud 3 dans sa deuxième passe, le nœud 3 n'étant pas demandeur, va tenter de satisfaire le nœud se trouvant au sommet de sa file 'sans se douter qu'il a déjà été satisfait par un jeton en transit' et envoie le jeton avec la mention '*Single*' cette fois puisque la file ne contient qu'un seul élément. Le nœud 1 étant déjà satisfait et n'ayant pas de requêtes dans sa file garde le jeton à son niveau. Le nœud 6 (figure 3.h) à sa sortie de la section critique 'constate' qu'il est lié au réseau par un seul lien, il cède son jeton libre au noeud 4.

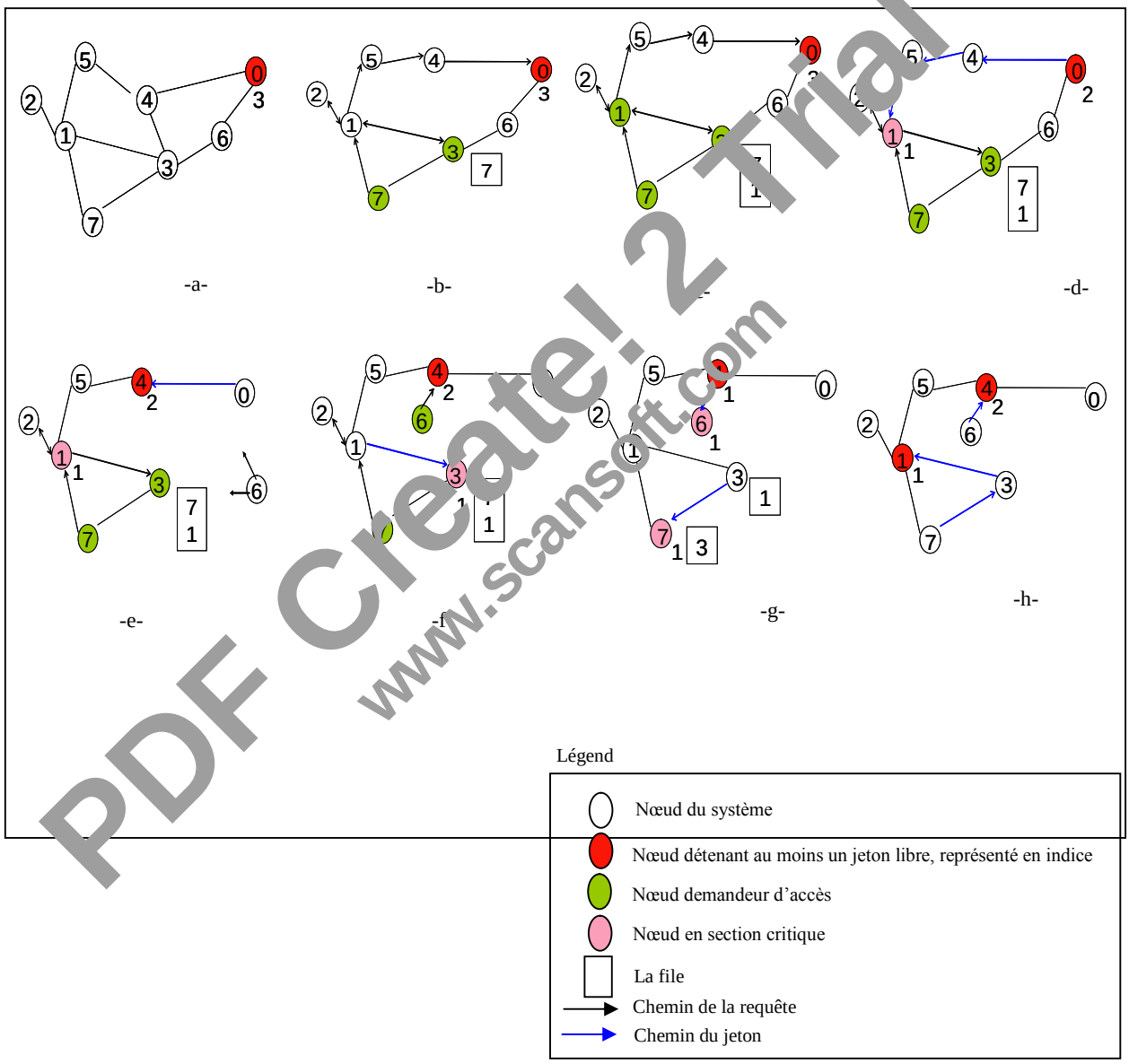


Figure 3 Illustration d'un déroulement de l'algorithme Ksearch (cas dynamique avec 3 jetons)

VIII. Discussion et conclusion

L'algorithme que nous avons proposé est orienté recherche de jeton et non circulation de jeton. La recherche du jeton est principalement basée sur une heuristique que nous énonçons comme suit : 'Le chemin qui conduira une requête le plus «*sûrement*» vers un jeton est celui qui vient de recevoir un jeton le plus récemment'. Cette heuristique n'est pas suffisante pour éviter les chemins cycliques. Nous avons besoin d'un outil qui permet de sauvegarder le chemin traversé, d'où l'utilisation des nombres premiers.

Nous essayons dans ce qui suit, de voir ces deux derniers principes de notre algorithme sous un œil critique et dégager certains aspects de force ou de faiblesse des outils utilisés:

- Utilisation des heuristiques

'Une heuristique est définie comme une technique qui aide à la découverte de solutions aux problèmes bien qu'il n'y ait aucune garantie qu'elle ne mène jamais dans la mauvaise direction.' [70]. Ce type de techniques est un recours là où les autres s'avèrent inefficaces. 'Le but d'une fonction heuristique est de guider le processus de recherche dans la direction la plus profitable, en suggérant quel chemin suivre en premier lorsqu'il y en a plus d'un disponibles. Plus l'estimation de la fonction heuristique sur les vrais mérites de chaque nœud dans l'arbre (le graphe) de recherche est précise, plus direct sera le processus de solution. A l'extrême, la fonction heuristique serait si bonne qu'aucune recherche ne serait vraiment nécessaire… La stratégie utilisée pour contrôler une telle recherche est souvent d'une importance capitale pour déterminer l'efficacité de cette recherche' [70].

Dans notre algorithme, nous avons utilisé les heuristiques à deux niveaux:

- Recherche de jeton: Depuis le départ, nous avons voulu faire abstraction de la topologie du réseau et appliquer une recherche arborescente puisque cette recherche assure l'existence d'un chemin entre deux nœuds différents du graphe. 'Sans les heuristiques nous serions irrémédiablement pris au piège de l'explosion combinatoire' [70].
- Prise de décision quant au comportement à adopter: L'algorithme peut être fixe à ce niveau, mais nous avons voulu paramétrer cet aspect afin de pouvoir vérifier l'efficacité des heuristiques à rendre un nœud plus intelligent et plus interactif. En outre, il serait intéressant d'évaluer le coût de cette solution.

- Utilisation des nombres premiers:

M.Raynal [35] a déjà exploité les nombres premiers dans le problème d'exclusion mutuelle dans les systèmes distribués, mais tous les calculs qu'il propose sont effectués de manière isolée localement. Dans notre algorithme, 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 de tenu de leur propriétés [72]:

- Il existe une infinité de nombres premiers;
- La factorisation d'un nombre en facteurs premiers est unique.

Néanmoins, nous sommes confrontés à deux inconvénients [72]:

- Lorsque le nombre de nœuds visités augmente le produit de leurs attributs augmente très rapidement au risque de causer un dépassement de capacité. Pour cela, nous pouvons faire des

propositions afin de palier ce problème; par exemple: utiliser plusieurs variables pour stocker dans chacune, une partie du produit;

- Il n'existe pas de formule algébrique pour représenter un nombre premier. Peu de formules existent, mais elles sont incomplètes (ne donne pas toute la suite des nombres premiers) et exigent un temps de calcul très élevé.

Tout le long de la conception de l'algorithme, notre souci était d'abord de solutionner le problème de K-exclusion mutuelle en prenant en considération l'aspect dynamique et aléatoire de notre environnement tout en proposant des optimisations qui permettraient de gagner du temps par rapport à tous les évènements qui peuvent se produire et palier ainsi toutes les conséquences (c'est l'idéal) du au changements fréquents de la topologie du réseau. Parmi les techniques proposées nous pouvons citer: l'utilisation d'une file temporaire, la méthode de gestion des jetons, les relances, la méthode d'optimisation du chemin des jetons, etc**…**, afin de favoriser la vérification des propriétés de '**vivacité**' et de '**sûreté**'.

Chapitre 2 Etude de simulation

I. Introduction

La preuve de correction d'un algorithme et l'évaluation de ses performances devrait se faire avec des outils formels tels que les outils mathématiques, les réseaux de pétri, les chaînes de Markov, etc... La simulation ne donne pas une preuve de correction formelle mais, elle permet non seulement de tester un algorithme sans aucun coût de nouvelles technologies et de nouveaux protocoles, mais aussi d'anticiper les problèmes qui pourraient se poser dans le futur; elle renseigne sur le comportement de l'algorithme devant des scénarios prédéfinis qui mettent en jeu un ensemble de paramètres tels que la mobilité dans le sens de la variation du déplacement et de la vitesse des nœuds, la charge des demandes, la connectivité, etc... La variation de ces paramètres permet suite aux différentes exécutions du programme une prise des mesures de performance. Ces mesures peuvent être déployées dans une analyse qui permet de tirer des conclusions sur le fonctionnement et l'efficacité de l'algorithme ainsi que les schémas favorables et défavorables quant aux performances calculées.

NS (Network Simulator) est un outil logiciel de simulation de réseaux informatiques. C'est un simulateur à évènements discrets et supporte plusieurs protocoles à différentes couches (session, application, transport, liaison et mac). Il fournit un environnement complet à plusieurs plates formes telles que les réseaux filaires, les réseaux mobiles, etc... NS est principalement bâti avec les idées de la conception par objets, de réutilisabilité du code et de modularité. Il utilise deux langages C++ et OTCL (Object Tools Command Language). Ce dernier est utilisé pour définir les conditions de la simulation. L'utilisateur peut décrire d'une part la topologie du réseau, les caractéristiques des liens physiques, les protocoles utilisés, les communications qui ont lieu, etc... d'autre part, il crée ses propres procédures.

Le simulateur de réseaux NS2 sous Linux est disponible en version 'AllInOne' incluant ainsi plusieurs outils tels que Tcl, Otcl, Xgrapgh et Nam. Les liens entre ces outils sont automatiquement établis rendant l'environnement de simulation plus facile.

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

Ce chapitre présente les différentes simulations réalisées ainsi que les résultats obtenus en passant par la présentation générale du simulateur NS2. A la suite, les différentes structures et classes qui constituent le programme seront mises en évidence. Dans les paragraphes qui suivent, les paramètres pris en considération dans ces simulations tels que la mobilité, la connectivité, etc... sont présentés. Ensuite, une définition de chaque critère de performance à évaluer sera donnée. Les résultats des différentes simulations seront présentés et interprétés selon les données initiales et les scénarios d'exécution générés. Nous terminons ce chapitre par une discussion et conclusion qui permettront de faire la synthèse du comportement de l'algorithme et des résultats obtenus.

II. Présentation du simulateur NS2 [72]

NS, acronyme de Network Simulator, fait parti du projet VINT qui est un effort commun mené par l'université de Berkeley, l'USC/ISI, le LBL et Xerox PARC, il est supporté par le DARPA (Defense Advanced Research Projects Agency). NS est un simulateur de réseaux orienté objet, écrit en C++, il utilise comme interface utilisateur un interpréteur OTcl. L'utilisateur peut créer de nouveaux objets en C++ et les utiliser ensuite dans NS en les instanciant avec OTcl, les deux langages ont une hiérarchie très proche l'une de l'autre.

NS a une utilisation assez souple grâce à l'utilisation d'OTcl, il existe de nombreux objets prédéfinis qui peuvent être utilisés pour simuler un grand nombre de scénarios aisément. On respecte

pour cela la syntaxe OTcl qui permet de manipuler facilement ces objets ainsi que leurs paramètres. Pour les simulations nécessitant un comportement particulier ou des protocoles spécifiques, on utilise le langage C++ pour programmer un agent adapté.

OTcl est une extension orientée objet de l'interpréteur Tcl (Tool Command Language). Tcl est un langage de commande comme le Shell UNIX mais qui sert à contrôler les applications. Tcl offre des structures de programmation telles que les boucles, les procédures ou les notions de variables. En pratique, l'interpréteur Tcl se présente sous la forme d'une bibliothèque de procédures C qui peuvent être facilement incorporées dans une application.

II.1. Nœuds mobiles

Le modèle de réseaux sans fil consiste essentiellement en l'objet 'MobileNode' dérivé de la classe 'Node'. L'objet 'MobileNode' possède les fonctionnalités des unités mobiles sans fil telles que la possibilité de déplacement sans aucune topologie donnée et la possibilité de transmettre et de recevoir des signaux vers/ou de canaux sans fil. La configuration d'un nœud mobile permet de lui attribuer le protocole de routage à implémenter (les 4 protocoles de routage DSDV, DSR, TORA et AODV peuvent être utilisés), le type de canal, le modèle de propagation, etc...

II.2. Modèle de mobilité

Il existe deux mécanismes pour introduire le mouvement des nœuds:
- Le premier permet au concepteur de définir explicitement la position de départ d'un nœud, sa destination, le moment où le déplacement doit commencer ainsi que la vitesse de déplacement.
- Le deuxième mécanisme consiste à utiliser le générateur de scénarios qui permet de générer un scénario de mobilité aléatoire en tenant compte de certains paramètres à passer tels que la superficie maximale pour le déplacement, la vitesse des nœuds, le temps de pause, etc...

En réalité, un nœud mobile peut se mouvoir dans les trois dimensions, cependant la troisième dimension n'est pas supportée par NS.

II.3. Modèle d'énergie

Le modèle d'énergie est défini par la classe 'EnergyModel'. Il existe une seule variable de cette classe qui permet de donner le niveau d'énergie d'un nœud en un temps donné. Il est nécessaire de passer en paramètre le niveau initial d'énergie ensuite, à chaque émission ou réception de paquets ce niveau est décrémenté jusqu'à ce qu'il soit nul; À ce stade aucune émission/réception n'est possible.

III. Implémentation du protocole

Le protocole de k exclusion mutuelle a été implémenté sous l'environnement NS version 2.26 avec le système d'exploitation Linux Mandrake version 9.0. Le langage OTCL a principalement été utilisé en tant que langage de programmation et a permis la création d'un ensemble de classes dont chacune couvre un aspect de la simulation; ceci n'exclue pas l'utilisation de classes prédéfinies. Les classes développées dans cette simulation sont :
- La classe NodeInfo: contient les variables de description d'un nœud mobile (par exemple: le nombre de jetons détenus, le status, la liste des voisins, etc...) et les méthodes permettant d'attribuer, de modifier ou de consulter ces variables (par exemple: la méthode PurgeTrace, PurgeQueue, AddTrace, etc...);
- La classe Node: contient l'ensemble des méthodes relatives aux mouvements des nœuds en termes de formation et rupture de lien et en termes de son activité locale tel que l'envoi d'une requête, sortie de la section critique, insertion d'un nouveau voisin, etc...;
- La classe RequestAgent: définit un agent UDP qui permet la réception des paquets ayant le format d'une requête au niveau de la couche transport;
- La classe TokenAgent: définit un agent UDP qui permet la réception des paquets ayant le format d'un jeton au niveau de la couche transport;

- La classe LinkAgent: définit un agent UDP qui permet la réception des paquets ayant le format d'un message LinkInfo au niveau de la couche transport;
- La classe Timer: Cette classe est essentiellement créée pour la gestion des timings tels que les délais de relance, etc...

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

Plusieurs procédures ont été créées avec TCL notamment celles qui réalisent des calculs ou des comparaisons entre entités. Par exemple une procédure divise(x,y) permet de déterminer si y est divisible par x.

IV. Paramètres et mesures

Dans cette simulation nous nous sommes inspirés des conditions de simulation du protocole 'KRL' de k exclusion mutuelle dans les réseaux mobiles ad hoc [60] de Walter. En effet, nous avons sélectionné les mêmes paramètres afin d'offrir des éléments de comparaison dans la mesure du possible. Néanmoins, des scénarios de simulation propres à notre algorithme ont été réalisés afin d'observer son comportement et évaluer ses performances face à des situations préfixées.

IV.1. Paramètres de mesure

Dans ce qui suit, nous présentons l'ensemble des paramètres pris en compte lors de cette simulation:

- La charge: D'après la définition de [60], La charge correspond au délai entre le moment où un nœud quitte sa section critique et formule une nouvelle demande. Les requêtes d'entrée en section critique sont déterminées par un processus de poisson avec un taux d'arrivée λ_{req}. Par conséquent, la charge correspond à une variable aléatoire exponentielle avec une moyenne de $1/\lambda_{req}$ unités de temps. Les valeurs prises pour λ_{req} dans [60] sont $\lambda_{req}=1$ (forte charge), $\lambda_{req}=10^{-1}$, $\lambda_{req}=10^{-2}$ et $\lambda_{req}=10^{-3}$ (charge faible). Dans notre simulation et vu le grand écart existant entre les valeurs précédentes de λ_{req}, nous avons pris des valeurs intermédiaires telles que $\lambda_{req}=1/5$, $\lambda_{req}=1/50$ et $\lambda_{req}=1/500$.

- La connectivité: La connectivité est calculée en termes de pourcentage des liens existant par rapport au nombre de liens possibles présents dans le graphe. Les valeurs de connectivité considérées par Walter dans [60] sont: 10%, 20%, 30%, 40%, 60%, 80% et 100%. Vu le temps considérable consommé par les différentes simulations, nous avons considéré seulement les connectivités de 20% et 80% pour les cas de topologie générale.

- La mobilité: Dans Walter [60], Les changements de liens (formation ou rupture) forment un processus de poisson avec un taux d'arrivée λ_{mob}. Par conséquent, la durée de temps entre deux changements consécutifs du graphe du réseau est une variable aléatoire exponentielle avec une moyenne de $1/\lambda_{mob}$ unités de temps. Les valeurs prises pour λ_{mob} sont $\lambda_{mob}=10^{-1}$ (forte mobilité) et $\lambda_{mob}=10^{-2}$ (faible mobilité) sans négliger le cas statique (qui correspond à la mobilité nulle). Cela signifie que pour le cas de forte mobilité, nous avons 1 seul changement de lien toutes les 10 unités de temps et 1 changement de lien toutes les 100 unités de temps pour le cas de faible mobilité. Nous considérons que ces valeurs sont faibles (tendent vers une mobilité nulle) et ne permettent pas de mettre en évidence les fonctionnalités de l'algorithme par rapport à la mobilité.

Le modèle de mobilité retenu est celui proposé par Tony Larson et Nicholas Hedman [70] et défini comme suit: 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. Le calcul de cette mobilité s'effectue en suivant les étapes suivantes:

1- Calcul de la distance moyenne d'un nœud i par rapport à tous les nœuds en des intervalles de temps t=0, t=0+X, t= 0+2X, … , t= temps de simulation. Cette valeur est donnée par:

$$Ai(t) = \frac{\sum_{j=1}^{n} dist(ni,nj)}{n-1} \quad (1)$$

2- Calcul de la mobilité moyenne d'un nœud i:

$$Mi = \frac{\sum_{t=0}^{T-\Delta t} |Ai(t)-Ai(t+\Delta t)|}{T-\Delta t} \quad (2)$$

3-Calcul de la moyenne des mobilités de tous les nœuds:

$$Mob = \frac{\sum_{i=1}^{n} Mi}{n} \quad (3)$$

où,
dist(ni,nj)t : est la distance entre un nœud i et un nœud j en un temps t,
n : est le nombre de nœuds,
T : est le temps de simulation,
Δt : est le pas avec lequel sont effectués les calculs. Dans notre simulation, le pas est égal à 1.

Ce modèle de mobilité a été testé (sur une durée de simulation de 250 unités de temps) afin de déterminer son influence sur le changement de la topologie du système. Nous pouvons constater d'après le graphe de la figure 4, que les changements de liens sont proportionnels au facteur de mobilité. Un changement de lien signifie un passage de l'état UP (lien établi) à l'état DOWN (lien rompu) ou inversement.

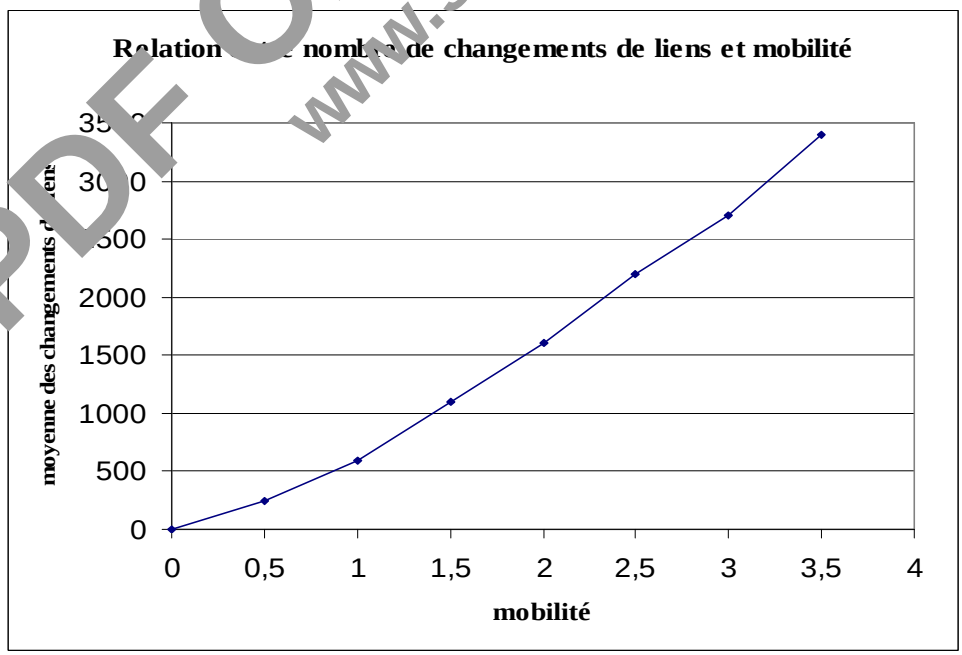


Figure 4.

ainsi, les valeurs de mobilité retenues sont données par:

- mobilité nulle: signifie que Mob = 0; Dans ce cas, 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. Plusieurs topologies on été prises en compte à savoir: un graphe complet, un arbre (binaire, à trois branches et à 5 branches) et une topologie quelconque dont nous avons pris plusieurs structures;
- Faible mobilité: signifie que $1 \leq Mob \leq 3$; ce qui donne un nombre de changements de liens compris entre 600 et 2700 sur toute la durée de simulation (soit 250 unités de temps), c'est-à-dire un nombre qui varie entre 2 et 11 changement de liens par unité de temps;
- Forte mobilité: signifie que $8 \leq Mob \leq 10$; ici, le nombre de changements de liens est supérieur à 30 changements part unité de temps.

IV.2. Paramètres généraux

Ce sont des paramètres nécessaires à fixer avant chaque simulation:
- Nombre de nœuds [60]: En général, le nombre de nœuds utilisés pour la simulation des réseaux mobiles ad hoc varie entre 10 et 50. Vu le temps considérable que prend une simulation le nombre de nœuds considérés est égal à 30;
- Rayons de communication: 250 m;
- Surface de simulation: La surface de simulation varie entre (300X300)m et (1000X1000)m en fonction de la vitesse de déplacement des nœuds. En effet, l'outil offert par NS pour la génération des scénarios de mobilité (setdest) permet d'obtenir des fichiers de mobilité avec la valeur de mobilité et de connectivité désirés en variant les deux paramètres surface de simulation et vitesse de déplacement;
- Nombre de jetons: Walter dans [60] a réalisé des simulations avec 1, 3 et 5 jetons. Nous avons en plus pris le cas de 8 jetons afin de pouvoir déterminer l'influence de ce paramètre sur les performances;
- Délai de relance: Le délai de relance correspond au temps nécessaire qu'un nœud ayant émis une requête, doit attendre pour considérer que sa requête comme étant non satisfaite. Ce délai est calculé grâce à la fonction *Delay()* décrite dans le chapitre précédent. Lorsque le délai de relance est atteint la demande d'accès à la section critique est renouvelée à moins que le seuil de relances maximum ne soit atteint (Nombre de relances).

Au lancement des premières simulations, nous n'avions pas une idée sur le comportement de l'algorithme à son exécution. Dès les premières tentatives, le délai de relance a été fixé à 70 unités de temps, valeur qui d'après les résultats obtenus s'est avérée très grande. Nous avons ensuite établis une moyenne qui varie entre 20 et 30 unités de temps.
- Nombre de relances: 5
- La durée de simulation: Ce paramètre varie lui aussi en fonction de la charge afin de permettre à chaque nœud de lancer un nombre significatif de requêtes (soit 10 au moins) sans léser le temps (réel) global de la simulation. Voici un tableau récapitulatif:

Charge	Durée de simulation (unités de temps)
1, 5 et 10	200
50 et 100	1000
500 et 1000	10 000

IV.3. 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:
- Lancement du système: Les premières requêtes lancées par chacun des nœuds représentent un amorçage de tout le système car à partir de la première sortie de la section critique correspondant à chaque nœud les autres requêtes du même nœud vont se succéder avec un taux $1/\lambda_{req}$ dont la valeur est prédéfinie. Nous avons plusieurs choix de lancement, par exemple:
- faire en sorte que toutes les requêtes soient lancées en même temps,
- lancer les requêtes aléatoirement avec un délai fixe ou variable séparant deux lancements,

- lancer les requêtes aléatoirement pendant un intervalle de temps qui pourrait varier en fonction de la charge.

Dans notre simulation, nous avons lancé les requêtes aléatoirement pendant les 30 premières unités de temps à partir du commencement de la simulation;

- Durée d'exécution de la section critique: Dans [60], une entrée en section critique dure 1 unité de temps. Ce qui semble raisonnable;

- Durée de transfert de messages: Dans [60], la durée de transmission d'un message est 1 unité de temps, valeur que nous avons conservé.

- Gestion de la file d'attente: Dans cette simulation, nous avons exécuté trois variantes de notre algorithme selon la manière de gérer la file d'attente:

- Dans la première variante, nous avons proposé de n'accepter qu'une seule requête en attente dans la file. Cette proposition a été appliquée dans le cas où le nombre de jetons est suffisamment grand (ie 8 et 5).
- Dans la deuxième, nous avons proposé d'accepter un nombre fixe de requêtes en attente dans la file. Cette proposition 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 section critique ou demandeur d'accès sont admises dans la file.

- Structure des fonctions paramètres: Rappelons que l'algorithme que nous avons conçu est modulaire et paramétré; pour cela, plusieurs fonctions ont été définies sans donner leurs structures. Toutefois, afin que la simulation soit réalisable, il est nécessaire de leur en donner une.

Dans cette simulation, nous avons choisi la structure la plus simple pour chaque fonction. Par exemple:

- la fonction *Seuil_File()*: délivre vraie si le nombre de jetons est égal à 1 ou bien s'il est égal à 3 en ayant au plus un seul élément dans la file d'attente ou bien s'il est égal à 5 ou 8 en ayant une file d'attente vide; c'est cette fonction qui a permis la gestion de la file d'attente telle qu'elle est décrite précédemment;
- la fonction *Seuil_R_File()*: délivre toujours vraie: Ceci permet de maintenir les requêtes pour lesquelles il est impossible d'acheminer un jeton, dans la file d'attente temporaire jusqu'à leur satisfaction ou destruction à la fin du délai qui leur est attribué;
- la fonction *Seuil_Util()*: délivre toujours vraie afin de permettre au jeton d'être exploité en cours de route chaque fois qu'il transite par un nœud se trouvant à l'état 'Need';
- la fonction D*elay():*Délivre une valeur fixe (variant entre 20 et 30 unités de temps);
- la fonction *Seuil_abandon()*: délivre vraie si le nombre de relances est fixé à 5; or, pour cette simulation, c'est le cas. Cette valeur est fixée à partir de l'observation des résultats des simulations dans lesquels le nombre de relances réellement effectuées ne dépasse jamais 5 par nœud.

V. Critères de performance

Ce sont les métriques que nous avons évaluées en tenant compte des paramètres déjà cités:

- Temps moyen d'attente par entrée en section critique: C'est le nombre moyen d'unités de temps écoulé entre la demande d'accès en section critique et l'accès lui-même. Cette métrique dépend du nombre de messages générés après une demande d'entrer en section critique, de la durée de séjour dans une file d'attente permanente et temporaire, ainsi que du délai de relance; Cette métrique est calculée par:

$$\frac{\sum_{i=0}^{i=N-1} \left(\sum_{r=1}^{r=NRi} \left| t_{2ri} - t_{1ri} \right| \right) / NR_i}{N}$$

Où,

t_{1r} est le temps de lancement d'une requête r,

t_{2r} est le temps où le message de jeton correspondant à la requête est reçu,
NR est le nombre total de requêtes émises par un nœud lors d'une simulation et
N est le nombre total de nœuds dans le système.

- Nombre moyen de messages générés par entrée en section critique: C'est le nombre moyen de sauts effectués par le message de requête et le message de jeton entre la demande d'accès en section critique et l'acquisition du jeton. Dans le cas mobile, nous avons inclus les messages LinkInfo et de cession. Cette métrique est calculée par :

$$\frac{\sum_{i=0}^{i=N-1} ((\sum_{r=1}^{r=NRi} Nmr_i) + NL_i)/NR_i}{N}$$

Où,
Nmr est le nombre de messages relatifs à une requête r,
NL est le nombre de messages LinkInfo émis par un nœud pendant toute la durée de simulation,
NR est le nombre total de requêtes émises par un nœud lors d'une simulation et
N est le nombre total de nœuds dans le système.

- Délai de synchronisation: C'est le temps moyen écoulé en termes de nombre moyen de messages générés entre deux entrées successives en section critique. Cette métrique permet de renseigner sur la manière dont sont gérés les jetons: plus sa valeur est réduite, meilleure est l'exploitation et la répartition des jetons dans le système. Cette métrique est calculée comme suit :

$$\frac{\sum_{i=1}^{i=Ne} | t_{ei}-t_{ei-1} |}{N_e}$$

Où,
T_{ei} est le temps simulé d'une entrée i en section critique,
N_e est le nombre total d'entrées en sections critiques lors d'une simulation entière.

- Nombre de relances: C'est le nombre moyen de renouvellement d'une demande d'acquisition d'un jeton relativement à une même requête. Cette métrique peut nous renseigner sur les cas d'échecs et d'évaluer leur ampleur. Cette métrique est calculée par :

$$\frac{\sum_{i=0}^{i=N-1} ((\sum_{r=1}^{r=NRi} NRelr)/NRi}{N}$$

Où,
NRel est le nombre de relances relatif à chaque requête émise par un noeud,
NR est le nombre total de requêtes émises par un nœud lors d'une simulation et
N est le nombre total de nœuds dans le système.

Remarque: Le temps d'intervention du système est considéré comme nul.

VI. Résultats et interprétation

Dans ce qui suit, nous allons présenter les résultats obtenus des différentes simulations en considérant séparément chacun des cas de mobilité (cas statique, faible et forte mobilité). Nous intégrons dans cette étude, les résultats obtenus par l'algorithme KRL afin de permettre une

comparaison même approximative des deux algorithmes. Il est à noter que vu la mobilité telle qu'elle est définie par Walter dans [60] et les valeurs de mobilité considérées par notre algorithme, les mobilités définies pour l'algorithme KRL sont très faibles. Pour cela, les résultats issus des simulations de l'algorithme 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 de l'algorithme KRL est ignoré. Le cas statique ne pose aucun problème. En outre, dans les simulations de l'algorithme KRL, seuls les deux premiers critères de performances (a savoir le temps d'attente et le nombre de messages pour une entrée en section critique) sont pris en considération. Donc, la comparaison ne tiendra compte que de ceux là.

Pour chacun des critères de performance suscités, nous présentons le graphe des courbes correspondantes aux mesures effectuées (on peut trouver le détail des mesures en annexe 2). L'interprétation de ces mesures permettra ensuite d'expliquer ou de justifier les valeurs obtenues et de montrer l'influence de la variation de certains paramètres sur ces mesures tout en se basant sur le comportement de l'algorithme.

V.1. Influence de la variation de la charge sur les performances

Les différentes simulations ont permis de dégager un comportement général de l'algorithme en fonction de la charge:

Lorsque la charge est forte, un nœud i, à sa sortie de sa section critique, formule une nouvelle requête après une très courte durée de temps (qui correspond à la charge) à destination du dernier nœud déjà servi par celui-ci (soit j). Si le nœud j ne peut pas satisfaire sa requête, celle-ci va explorer les autres nœuds, conformément à la méthode de recherche appliquée dans l'algorithme, à la recherche d'un autre jeton. Si au contraire, j peut servir la requête de i immédiatement ou la mettre dans la file d'attente, le nœud i peut être servi dans un futur proche (si la charge est égale à 1, ce scénario peut se répéter k fois dans le système chaque unité de temps (k étant le nombre de jetons dans le système). Cela signifie, qu'à tout instant, k processus sont en section critique et N-k autres en attente.

Les faibles charges (100, 500 et 1000) permettent une activité intense du système pendant un intervalle de temps limité dans lequel une seule requête au plus est en cours de traitement pour chaque nœud, le reste du temps le système est oisif et attend le prochain lancement. Par exemple: à la charge 100, l'activité du système peut être représentée par le schéma suivant:

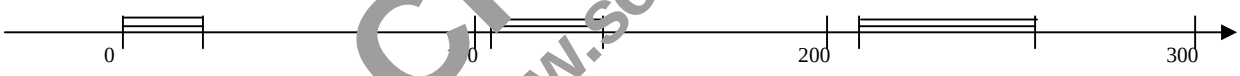


Figure 5. Comportement global en cas de faible charge

Où,

═══ Représente une phase d'activité (chaque nœud a au plus une requête en cours),

——— Représente une zone d'oisiveté.

Nous déduisons que dans chaque intervalle de 100 unités de temps (respectivement de 500 et 1000 unités de temps), le nombre de requêtes circulant dans le système est toujours inférieur ou égal à N-K (si l'on considère que K ressources sont occupées, N étant le nombre de nœuds dans le système).

Dans ce qui suit, seuls les échantillons relevés pour un nombre de jetons égal à 3 sont pris en considération dans les tracés des courbes.

V.2.1. Cas de réseau statique

Dans le cas statique, les courbes issues des simulations de l'algorithme 'Ksearch' et celles issues des simulations de l'algorithmes 'KRL' sont représentées dans un même repère. Dans nos simulations, deux topologies sont considérées: une topologie en arbre (binaire, à 3 branches et à 5 branches) et une topologie quelconque avec une connectivité de 20% et 80 %.

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

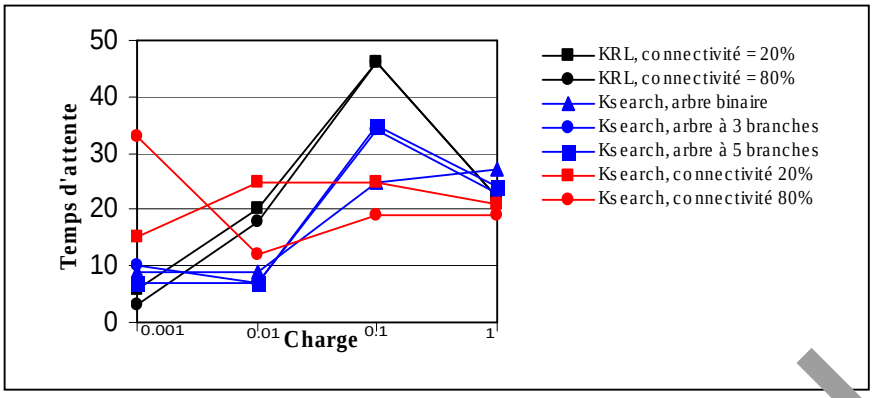


Figure 6. Temps moyen d'attente pour une entrée en section critique (cas statique, 3 jetons)

A la première vue de ces graphes, il est important de souligner l'existence de trois faisceaux différents de courbes. Le premier (en noir) correspond à l'algorithme 'KRL' de Walter [60] et les deux autres correspondent à notre algorithme et qui représentent les deux types de topologies: topologie en arbre (en bleu) et topologie générale ou quelconque (en rouge). Le comportement de notre algorithme en terme de temps d'attente par entrée en section critique dans le cas de structures arborescentes est remarquablement identique à celui de l'algorithme 'KRL': les deux faisceaux de courbes ont la même allure.

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

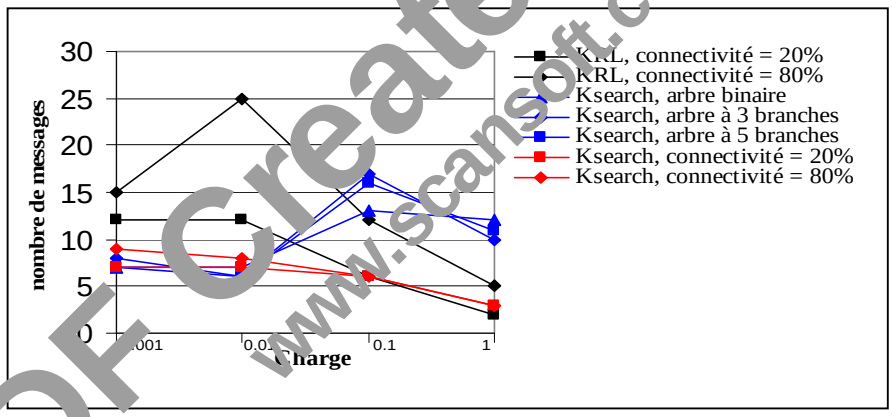


Figure 7. Nombre moyen de messages pour une entrée en section critique (cas statique, 3 jetons)

Les performances en terme de nombre de messages par entrée en section critique offertes par notre algorithme semblent être meilleures que celle de l'algorithme 'KRL' dans le cas de faible charge. Dans le cas de structures quelconques, le faisceau de courbes obtenu (en rouge) est inférieur à celui de l'algorithme 'KRL', ce qui marque de meilleures performances. L'allure générale des courbes issues des topologies en arbre est croissante et celle des courbes issues des topologies quelconques est décroissante.

- **Délais de synchronisation**

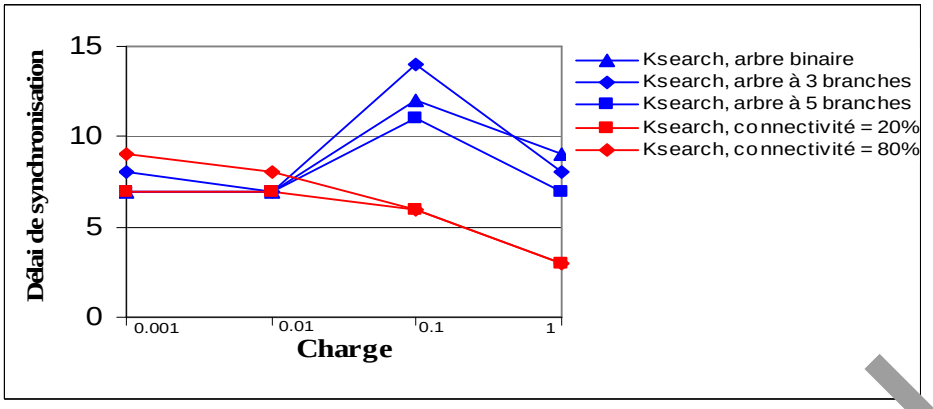
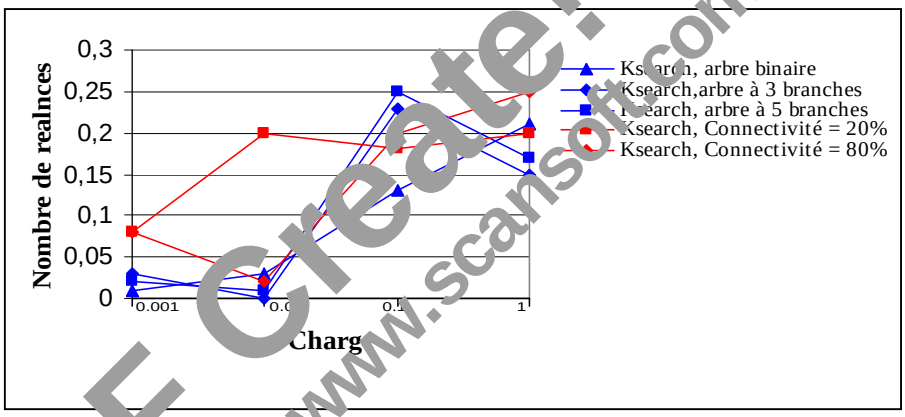


Figure 8. Délais de synchronisation (Cas statique, 3 jetons)

Les délais de synchronisation enregistrés dans le cas de faible charge sont très proches quelques soit la topologie et la connectivité. Cependant, les courbes issues de la topologie arborescente s'éloignent dans le sens croissant ensuite décroissant pour marquer de plus faibles performances dans le cas de forte charge. Parallèlement, les courbes issues des topologies quelconques sont décroissantes marquant ainsi de meilleures délais de synchronisation dans le cas de forte charge.

- **Nombre de relances**



gure 9. Nombre moyen de relances (cas statique, 3 jetons)

Si l'on considère les courbes tracées en bleu, il est possible de remarquer que la croissance de ces dernières a la même allure que celle des courbes correspondantes relatives aux différentes métriques représentées dans les graphes précédents; ce qui montre une forte corrélation entre les métriques essentiellement dans les cas de topologies en arbre.

- **Analyse des résultats**

En général, nous constatons que les courbes issues de la même topologie ont la même allure pour toutes les métriques relevées, ce qui exprime la corrélation existant entre ces métriques.

Les graphes obtenus à partir des mesures effectuées sur des topologies en arbre ont une allure plutôt croissante ce qui exprime que plus la charge augmente plus les performances diminuent. Il est à constater que l'écart entre les courbes n'est pas très grand sauf le cas binaire qui a l'air de s'écarter légèrement lorsque la charge est de 10; A cette même charge les trois courbes montrent un pic, mais en réalité les valeurs des mesures au charges 10 et 1 sont relativement proches (voir Annexe 2).

Le comportement général de l'algorithme montre que dans le cas de forte charge de demandes d'accès en section critique, K processus exactement sont en section critique et N-K processus sont soit en attente soit entrain de formuler de nouvelles relances et ce durant toute la simulation; ce qui

exprime une forte activité du système et peut expliquer les valeurs de performances obtenues. Dans les cas de faible charge (voir figure 5), 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. Ajouté à ce comportement, la circulation des messages de requêtes et jetons est limitée à deux sens de circulation: le premier va depuis la racine jusqu'aux feuilles et l'autre inversement. A la recherche d'un jeton, un message de requête est appelé à explorer entièrement le sous arbre en cours avant de passer à un autre sous arbre. Dans les meilleures circonstances, le jeton peut être présent et servi dans le premier sous arbre exploré sinon, il serait au niveau de la dernière branche à visiter. Les performances dans ces cas, dépendent du parcours de la requête par rapport à la position du jeton. En outre, dans une structure arborescente le chemin du jeton n'est jamais optimisé du fait même de la topologie.

Contrairement aux graphes obtenus des mesures effectuées sur des structures arborescentes, ceux issus des topologies quelconques sont en général décroissants; ce qui exprime que les meilleures performances sont obtenues dans les conditions de forte charge. Ceci exprime aussi que les solutions d'optimisation telles que: exploitation des jetons en cours de route, introduction des nœuds de type '*Collector*' et optimisation du chemin des jetons sont plus fonctionnelles dans les topologies générales que dans les topologies en arborescence, ce qui engendre une diminution considérable du nombre de messages. Or, les performances en terme de durées d'attente par entrée en section critique ne semblent pas être améliorées dans tous les cas, ceci peut s'expliquer par une mauvaise estimation des temps des relances qui influencent considérablement les temps d'attente.

Les résultats obtenus pour les délais de synchronisation montrent de meilleures performances quant à cette métrique dans le cas de topologies générales. Ceci, s'est essentiellement observé dans le cas de forte charge. Ce qui exprime une exploitation plus intense et plus équitable des jetons dans le cas de topologies générales.

En général, les graphes obtenus montrent des performances (en termes de délai moyen pour une entrée en section critique) issues de notre algorithme, meilleures que celles de l'algorithme KRL dans les cas de forte charge. Quant au nombre de messages pour une entrée en section critique, ils sont plutôt meilleurs dans les cas de faible charge. Tous ces résultats peuvent être améliorés ajustant les durées de relance.

V.1.2. Cas de réseau dynamique

Dans ce qui suit, nous allons représenter outres les courbes relatives aux différentes simulations de notre algorithme effectuées dans le cas de faible et forte mobilité, les courbes des mesures de l'algorithme 'KRL' relatives au cas de forte mobilité (en noir) qui est considérée comme faible par rapport aux valeurs de mobilité de nos simulations.

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

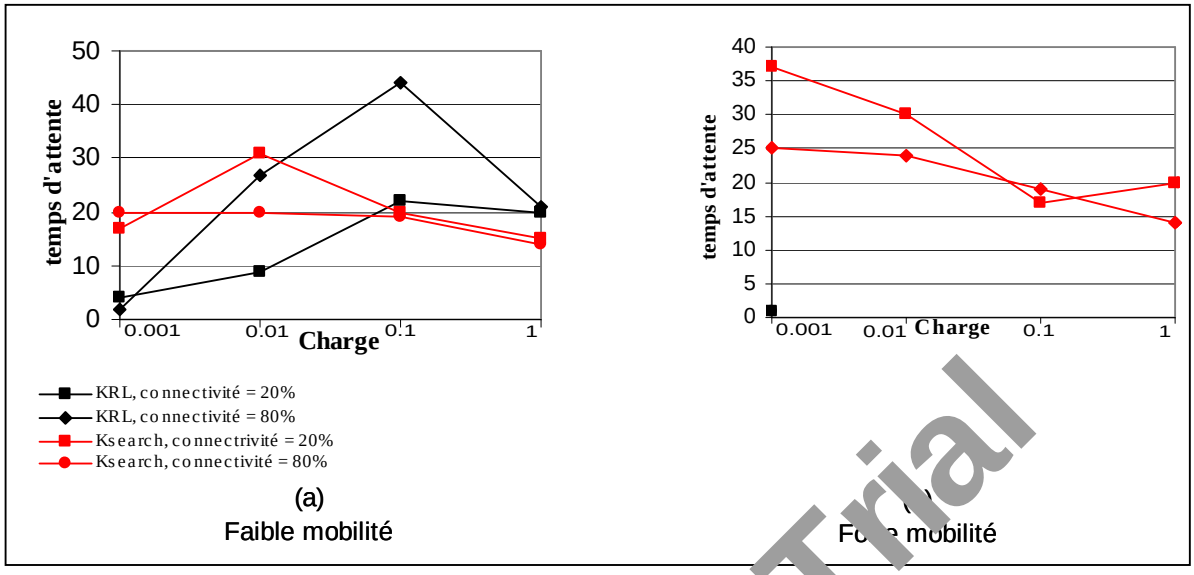


Figure 10. Temps moyen d'attente (cas dynamique, 3 jetons)

Comme dans le cas statique, topologie générale, les temps d'attente relevés par notre algorithme semblent meilleurs que ceux de l'algorithme KRL dans les cas de forte charge.

Concernant notre simulation, l'allure générale des courbes issues des cas de faible et de forte mobilité est décroissante, ce qui marque un meilleur comportement dans les cas de forte charge.

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 10.a et 10.b).

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

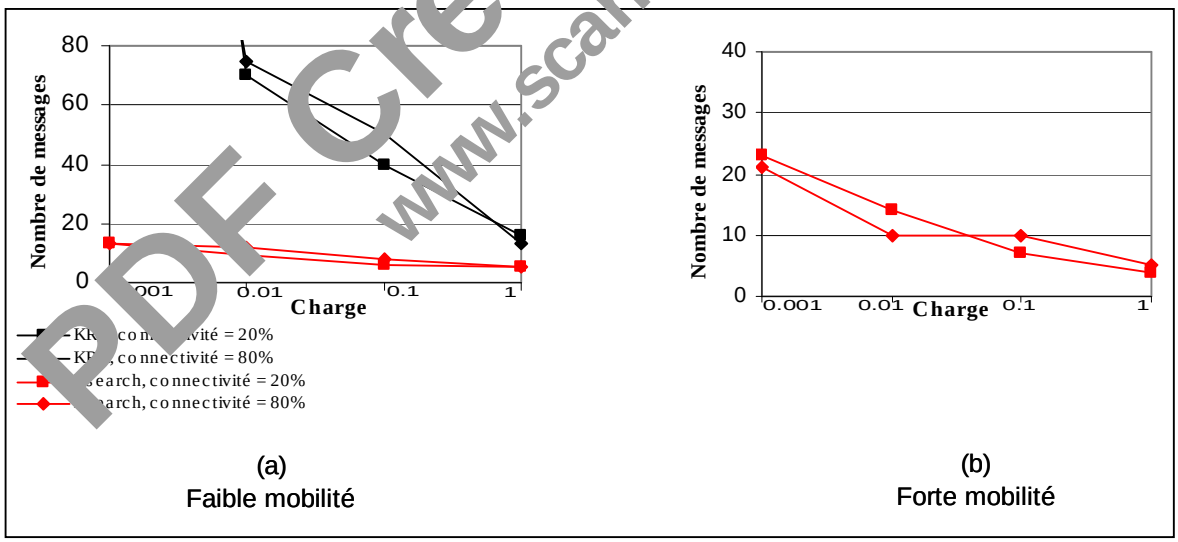


Figure 11. Nombre moyen de messages (cas dynamique, 3 jetons)

Les deux courbes (en rouge, figure 11.a) semblent se confondre en ayant une allure décroissante. L'algorithme KRL présente une dégradation des performances dans le cas de faible charge; et, dans tous les cas de charge, nos performances sont meilleures que celles de l'algorithme 'KRL'.

De même que les courbes de la figure 11.a, les deux courbes de la figure 11.b ont une allure décroissante avec un écart très réduit.

- **Délais de synchronisation**

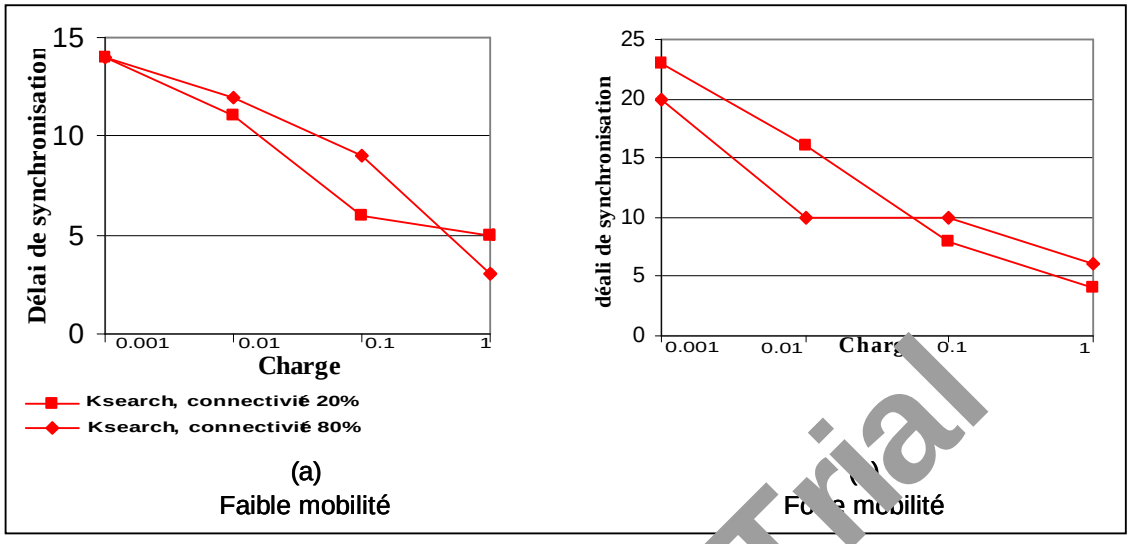


Figure 12. Délai moyen de synchronisation (cas dynamique, 3 jetons)

Il n'y a pas un grand écart entre les courbes en faisant varier la connectivité (graphe 12.a et 12.b). L'allure de celles-ci est décroissante avec un léger avantage des conditions de faible connectivité par rapport à la forte connectivité.

- **Nombre de relances**

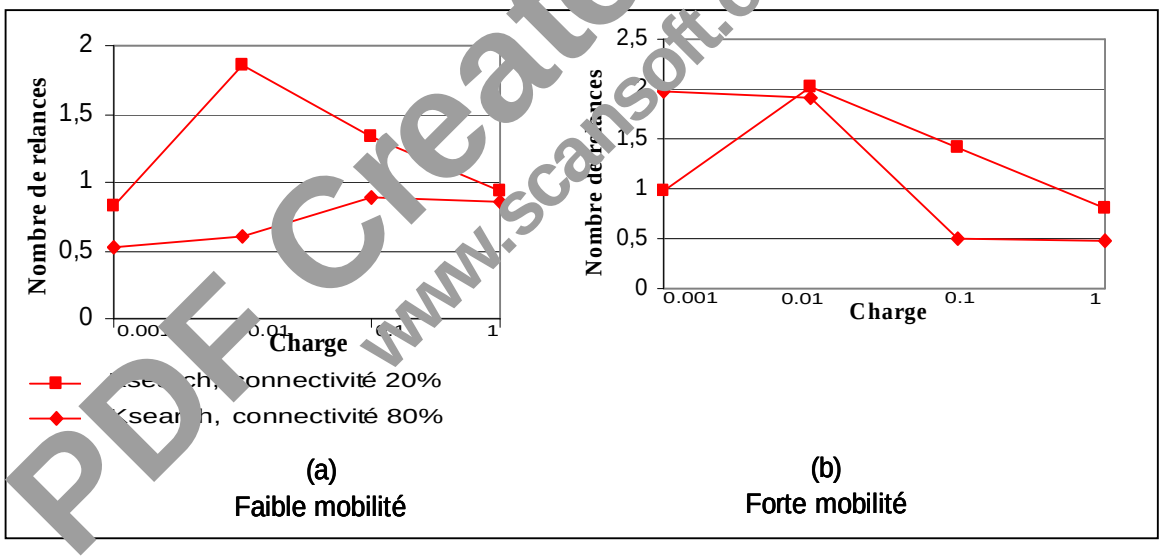


Figure 13. Nombre moyen de relances (cas dynamique, 3 jetons)

Il est à constater à partir de cette figure (Figure 13.a et 13.b), que le nombre de relances par entrée en section critique est plus important lorsque la connectivité est faible.

- **Analyse des résultats**

Les courbes obtenues des mesures effectuées (toute métrique confondue et quelque soit la mobilité) ont une allure décroissante, ce qui signifie que plus forte est la charge, meilleures sont les performances. Ceci peut être expliqué comme pour le cas de la topologie générale du cas statique par l'apport considérable des solutions d'optimisation apportées par l'algorithme à savoir: l'utilisation de jetons '*Collector*', 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 statique (topologie générale) sont meilleures que celle du cas mobile (de faible mobilité), ceci s'explique par le fait de la mobilité elle-même qui, plus intense est-elle, plus elle génère de messages de type LinkInfo (qui n'ont lieu que lorsqu'il y a des chemins en commun entre les nœuds dont la liaison vient de s'établir et qui sont pris en compte dans les calculs des performances) 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 (ce qui est visible dans la courbe relative à la connectivité 20% de la figure 13 et qui est plus élevée que celle de la connectivité 80%). Néanmoins, les performances obtenues sont acceptables et comparés aux performances de l'algorithme KRL, elles sont visiblement meilleures sauf dans le cas de faible charge où les temps d'attente de ce dernier offrent les meilleures valeurs.

Le cas de la forte mobilité n'a pas modifié le comportement général de l'algorithme. En effet, les courbes obtenues quelque soit la performance en question ont gardé la même allure avec une légère baisse des performances, ce qui est prévisible.

V.2. Influence de la variation du nombre de jetons sur les performances

Afin de montrer l'influence de la variation des jetons sur les performances nous avons simulé notre algorithme avec un système respectivement à 8, 5, 3 et 1 jeton (s). Nous avons construit des graphes pour chacune des métriques à évaluer dans les trois cas de mobilité (statique, faible et forte mobilité) sans omettre de représenter les courbes issues des simulations de l'algorithme KRL.

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 visibilité des graphes.

Le travail qui va suivre permet de vérifier une propriété prévisible selon laquelle l'augmentation du nombre de jetons va servir l'algorithme de K-exclusion mutuelle dans le sens de l'amélioration des performances.

V.2.1. Cas de réseau statique

Le cas statique est représenté par les deux topologies (générale et arborescente). Dans la structure arborescente, nous n'avons considéré que le cas d'arbre à 3 branches.

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

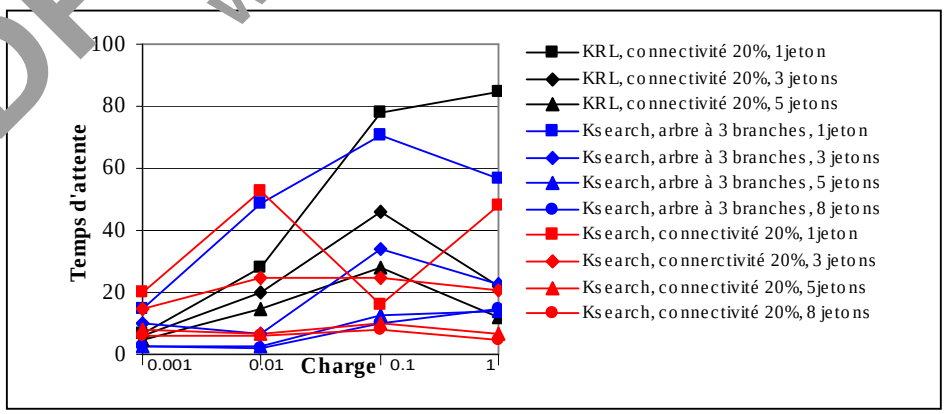


Figure 14. Temps moyen d'attente (cas statique)

Les courbes issues des simulations de l'algorithme KRL (en noir) montrent une croissance du temps d'attente proportionnellement avec l'augmentation de la charge et la diminution du nombre de jetons. Le même comportement est observé dans notre algorithme dans le cas statique avec une structure arborescente avec de meilleures performances en générale.

L'observation des courbes (en rouge) issues du cas de topologies générales, montre que plus le nombre de jetons augmente, meilleures sont les performances, principalement lorsque ce nombre vaut 5 et 8. Dans les cas de 1 et 3 jeton(s), nos résultats sont meilleurs que ceux de l'algorithme KRL dans les conditions de faible charge et, dans les mêmes cas, la topologie générale offre de meilleures performances que la topologie arborescente. En général, les courbes issues de la même topologie ne présentent pas un très grand écart entre elles sauf dans le cas d'un jeton unique dont les performances ne sont pas si dégradées que prévisible; Elles sont même relativement bonnes.

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

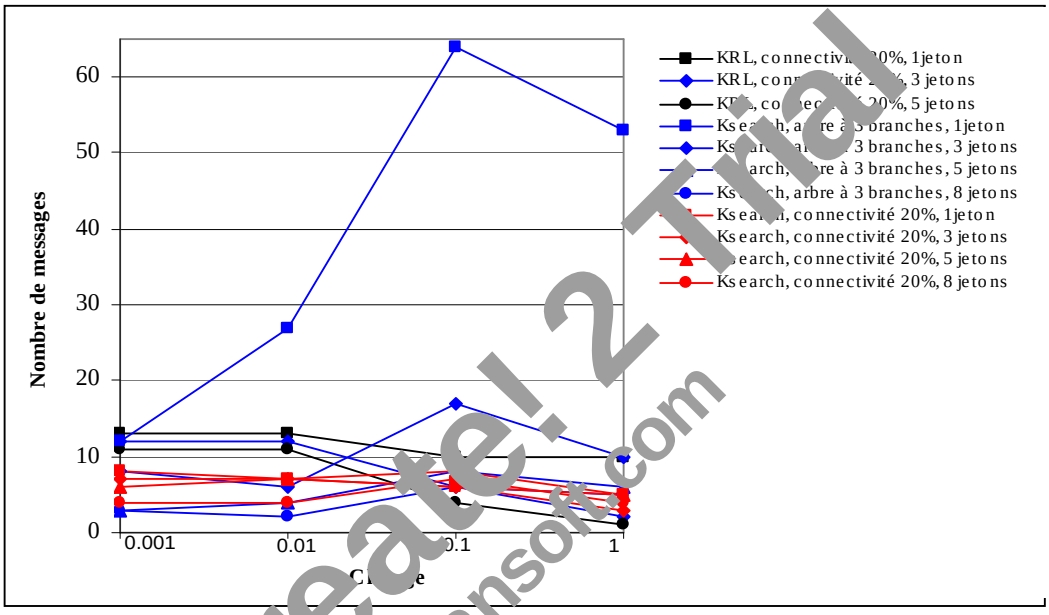


Figure 15 : Nombre moyen de messages (cas statique)

En terme de nombre de messages par entrée en section critique, il est important de souligner que les valeurs de mesures obtenues dans tous les cas (sauf le cas arborescent avec 1 jeton) appartiennent au même domaine compris entre 1 et 17. Les courbes obtenues présentent une allure plutôt décroissante, sauf le cas arborescent. Les courbes issues de notre algorithme présentent de meilleures performances que celles de l'algorithme KRL dans les conditions de faible charge. En outre, le nombre de messages par entrée en section critique est inversement proportionnel au nombre de jetons ce qui est prévisible.

- **Délais de synchronisation**

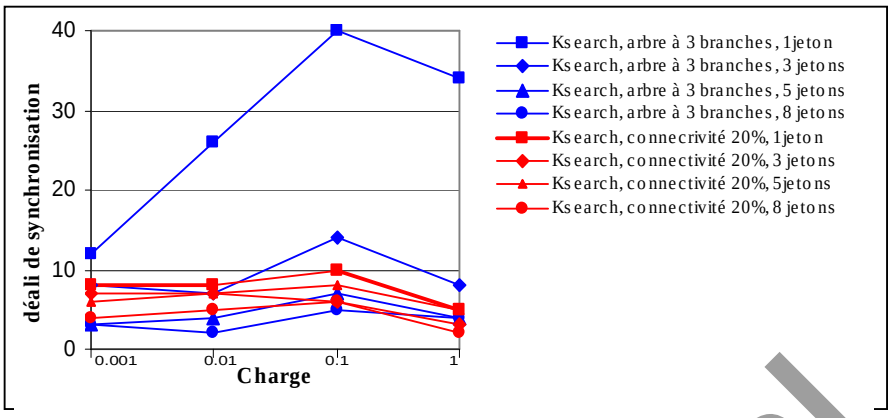


Figure 16. Délai moyen de synchronisation (cas statique)

De même que pour la métrique précédente, les valeurs des mesures effectuées pour cette métrique appartiennent à un intervalle de valeurs réduites (sauf le cas arborescent avec 1 jeton).

- **Nombre moyen de relances**

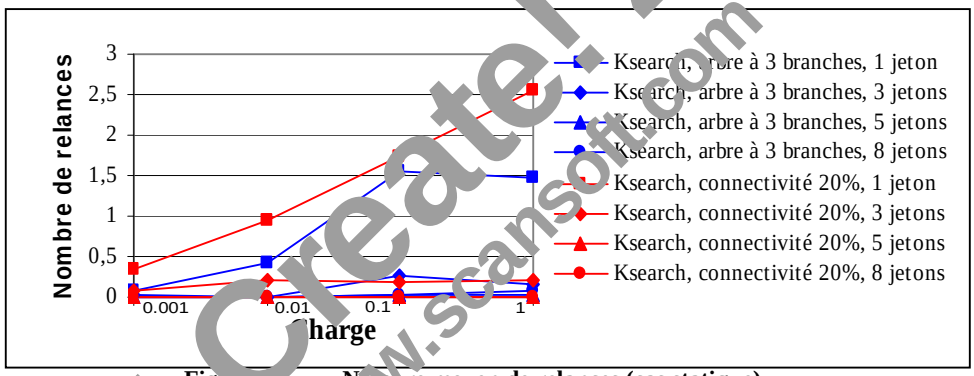


Figure . re moyen de relances (cas statique)

Dans le cas statique, le nombre de relances est négligeable lorsque le nombre de jetons est de 3, 5 et 8. Dans le cas à 1 jetons le nombre de relances est plus important, ce qui est prévisible.

- **Analyse des résultats**

Le graphe des courbes donnant le nombre moyen de messages par entrée en section critique en fonction de la charge montre clairement que les performances sont meilleures lorsque le nombre de jetons augmente notamment pour le cas de structures arborescentes. Ce résultat est aussi observable dans les courbes correspondantes relatives aux deux métriques 'Temps d'attente' et 'Délai de synchronisation'. Le nombre de relances (voir Figure 17) est plus important lorsque le nombre de jetons diminue, qu'il s'agisse de structures arborescentes ou quelconques. Les écarts entre les courbes (respectivement en rouge et en bleu) ne sont pas très importants. En effet, les valeurs des mesures sont tellement proches que les courbes obtenues ont l'air de se confondre et de prendre une allure presque horizontale, ce qui peut laisser penser que certains jetons, dans le cas où le système dispose d'un grand nombre de jetons (soit 5 ou 8 jetons) ne sont pas bien exploités dans le sens où certains jetons restent stationnaires alors qu'un besoin existe dans le système (en d'autres termes: seul un nombre réduit de jetons est actif). Cependant, cette dernière conclusion n'est pas vraie. En effet, en observant le graphe issu des mesures sur les temps d'attente par entrée en section critique en fonction de la charge, nous remarquons que les courbes représentant les mesures effectuées sur des simulations à 5 et à 8 jetons

s'écartent en présentant de meilleures performances.

V.3.2. Cas de réseau dynamique

Vu le grand écart existant entre les courbes issues des simulations de l'algorithme KRL de celles de notre algorithme, nous avons représenté les courbes respectives de chaque algorithme dans un graphe à part.

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

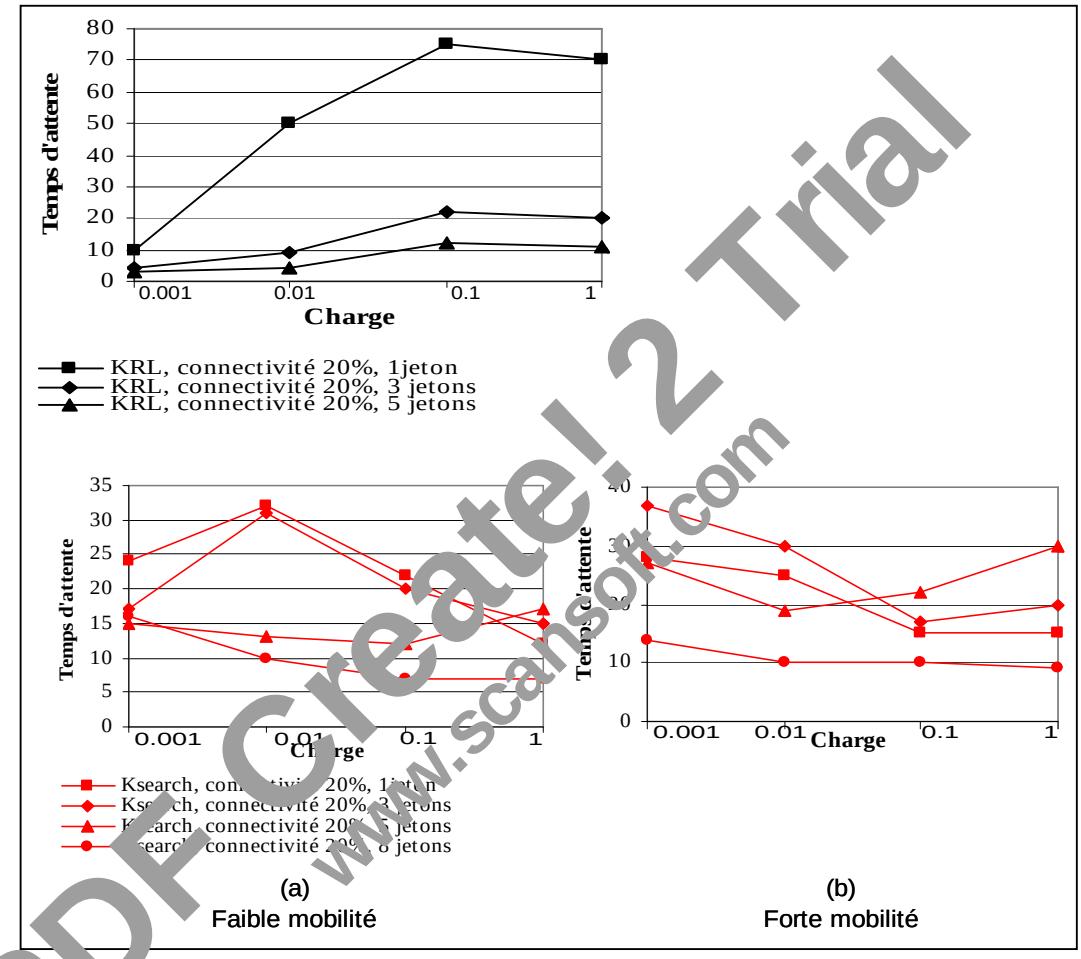


Figure 18. Temps moyen d'attente (cas dynamique)

En général, le temps d'attente augmente proportionnellement au nombre de jetons; mais dans certains cas (voir figure 18) 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.

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

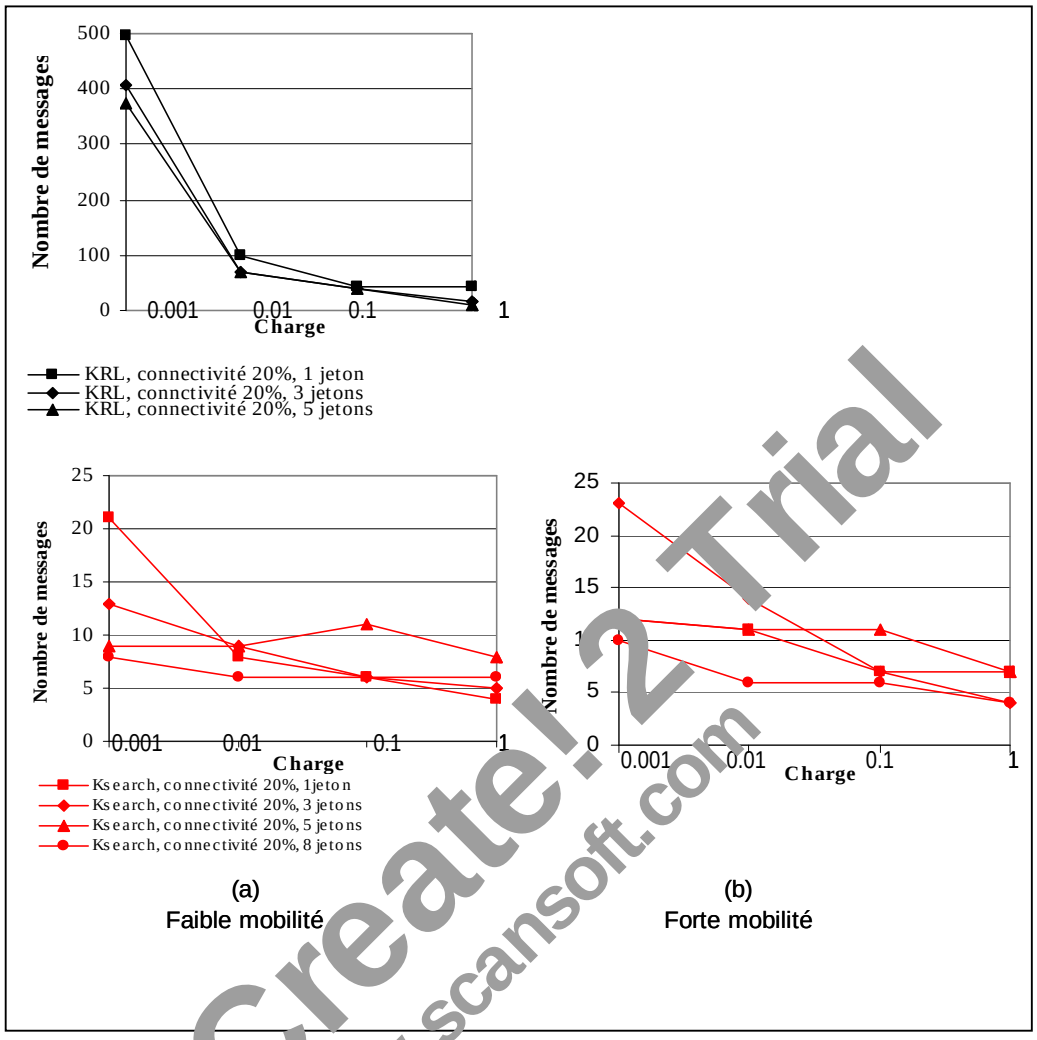


Figure 9. Nombre moyen de messages (cas dynamique)

Dans l'algorithme KRL, le nombre de messages est inversement proportionnel au nombre de jetons. L'allure des courbes relatives aux simulations de notre algorithme est décroissante avec un chevauchement important entre elles. Les courbes issues des mesures effectuées dans le cas de faible mobilité sont comparables à celles des mesures effectuées dans le cas de forte mobilité avec des valeurs de mesure plus élevées dans le dernier cas.

- **Délais de synchronisation**

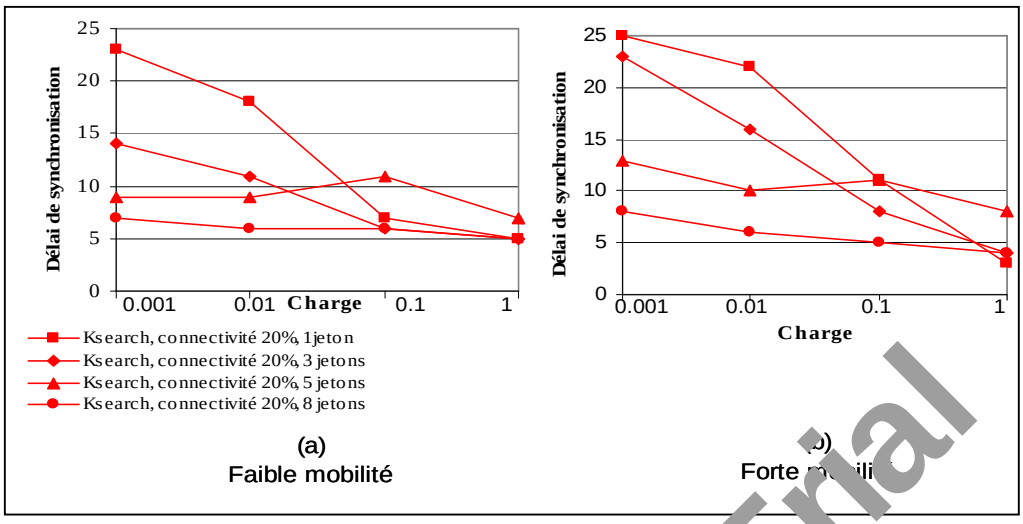


Figure 20. Délai moyen de synchronisation (cas dynamique)

Dans les deux cas de mobilité, les graphes obtenus ont la même allure décroissante. Le cas de forte mobilité (Figure 20.b) marque une légère diminution des performances par rapport au cas de faible mobilité (Figure 20.a).

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.

- **Nombre moyen de relances**

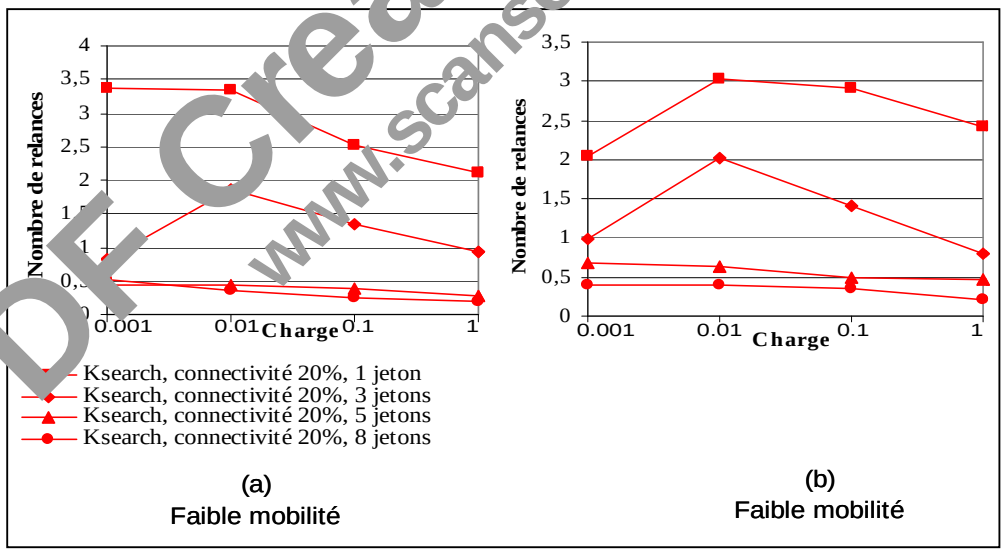


Figure 21 Nombre moyen de relances (cas dynamique)

Dans les deux figures 21.a et 21.b 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.

- **Analyse des résultats**

L'observation des courbes issues des mesures sur le nombre moyen de messages par entrée en section critique 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 performance voisines. L'influence de la variation du nombre de jetons est plus évidente dans le graphe des temps d'attente par entrée en section critique en fonction de la charge, où plus le système dispose de jetons moins un nœud attend pour entrer en section critique, 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 des 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 par deux phénomènes possibles:

- 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;

- Les cas où un nœud se trouve lié au réseau par un seul lien sont fréquents, mais le nombre de nœuds qui sont appelés à envoyer leurs jetons par 'crainte' d'un isolement futur est d'autant plus en plus réduit lorsque le nombre de jetons dans le système diminue. Cela veut dire que si le système ne dispose que d'un seul jeton, le nombre de messages de cession de jetons est au maximum égal à 1 à un instant t: Ce serait par hasard le nœud qui détiendrait le jeton unique qui se trouverait en voie d'isolement probable.

Les courbes du graphe donnant le 'Délai moyen de synchronisation pour une entrée en section critique' montre le même comportement en général.

Comparées aux mesures effectuées dans le cas statique, 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.

VII. Discussion et conclusion

Initialement, plusieurs objectifs étaient fixés pour cette étude de simulation. D'une part, il s'agissait d'étudier le comportement de l'algorithme devant des situations variées et d'évaluer ses performances relativement à ses propres caractéristiques. D'autre part, de réaliser une étude comparative entre notre algorithme et son prédécesseur (l'algorithme KRL) en termes de performances. Malheureusement, malgré que la plupart des paramètres de mesure soient issus de la simulation de Walter dans [60], l'étude comparative ne peut être réalisée de manière objective et exhaustive que si les conditions de simulation des deux algorithmes soient identiques. En effet, à titre d'exemple, les conditions de lancement des processus et la méthode de génération des fichiers scénarios de mobilité ne sont pas connus. En outre, le modèle de mobilité considéré dans la simulation de l'algorithme KRL diffère du notre. Cependant, nous avons réunit certaines conditions qui rapprochent les deux algorithmes et nous les avons comparés. Par exemple, la comparaison était possible pour le cas de réseaux statiques. Dans le cas de réseaux dynamiques, nous avons constaté que les conditions de forte mobilité relatives à l'algorithme KRL étaient proches de celles de la faible mobilité relatives à notre algorithme, c'est dans ce sens que la comparaison a été réalisée.

D'après les graphes obtenus des différentes simulations, il semble que les résultats offerts par notre algorithme sont meilleurs que ceux de l'algorithme KRL. En effet, les courbes issues de nos simulations sont en général inférieures à celles des courbes issues des simulations de l'algorithme 'KRL'.

Les courbes représentant chaque métriques: 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 (sauf cas statique arborescent). Ceci montre que notre la algorithme réagit de manière efficace dans les conditions de forte charge où les solutions d'optimisation proposées dans l'algorithme (utilisation de jetons '*Collector*', la cession de jeton en cas de crainte d'isolement, exploitation de jetons 'single' en cours de transit, etc...) son fortement sollicitées et mises en œuvre de manière intense.

L'augmentation du nombre de jetons est en la faveur de l'algorithme en termes de performances. Cependant, certaines situations présentent des chevauchements entre les courbes contrairement à nos prévisions. Dans le cas dynamique, ceci peut être du principalement à la diminution des messages de cession de jetons et des messages *LinkInfo*. En général, l'ajustement des

délais des relances est de rigueur car son influence n'est pas négligeable.

Comme dans l'algorithme KRL, la variation de la connectivité ne semble pas avoir une grande influence sur les performances de l'algorithme. En effet, les écarts entre les valeurs des mesures prises respectivement pour la connectivité 20% et 80% ne sont pas très grands pour toutes les performances (délai d'attente, nombre des messages et délai de synchronisation) et tous les cas de mobilité. Néanmoins, il existe un facteur important et qui n'est visible ni sur les tableaux de mesures ni sur les graphes, c'est celui du temps de calcul au sein de l'algorithme. En effet, l'observation des états d'avancement des différentes simulations concernant le cas de forte connectivité (80%), montre une lenteur de calcul essentiellement aux moments où il est question de calculer ou de décomposer la valeur de T (Rappelons que T est le produit des nombres premiers représentants chacun l'attribut d'un nœud qui a servi au transit de la requête correspondant à ce produit). Nous déduisons par la pratique que l'utilisation des nombres premiers nécessite une grande capacité de calcul.

Dans le cas de forte mobilité, nous constatons une légère dégradation des performances comparée à celles du cas de faible mobilité. Ces dernières étant aussi moins bonnes que celles du cas statique (topologie quelconque). Néanmoins, les résultats obtenus sont généralement acceptables comparés à ceux de l'algorithme 'KRL'.

Chaque simulation a permis de générer deux fichiers en sortie. Le premier est un journal de toutes les opérations effectuées. Le deuxième, comprend la synthèse des résultats individuels de chaque nœud et de la session en général. La consultation de ces fichiers montre que notre algorithme 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 sections critiques avec un même nombre de relances (ce nombre étant en général négligeable; voir annexe 2).

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

- Faire varier le nombre de nœuds pour voir les limites de l'algorithme;
- Faire en sorte que le nombre de nœuds soit variable: par exemple en ajoutant de nouveaux nœuds dans le système de manière aléatoire;
- Faire varier la taille de file en fonction de l'environnement de la simulation. Et, éventuellement faire en sorte que dans la même simulation l'algorithme décide de manière autonome de la taille de la file au niveau d'un nœud en fonction des informations collectées;
- Maintenant que nous avons une idée sur le fonctionnement de l'algorithme, il est possible de lancer des simulations avec des durées de relance adéquates. Ce paramètre a une importance particulière car si la durée de relance est très petite, cela peut engendrer plus de messages inutilement et si elle est trop grande les temps d'attente pour une entrée en section critique pourraient être trop grand. D'autre part, l'algorithme pourrait faire varier cette durée de manière autonome en fonction du déroulement de la simulation, c'est-à-dire des informations collectées;
- Faire varier les conditions de lancement du système;

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

CONCLUSION GENERALE

La K-exclusion mutuelle dans les réseaux mobiles ad hoc est un problème crucial, d'une part, vu son importance en tant que système de contrôle et de gestion de conflit d'accès aux ressources et d'autres part, à cause de la problématique engendrée qui incite à aborder le sujet sur deux volets différents mais interférents:
- Un système basé sur un réseau mobile ad hoc est un système distribué;
- Un réseau mobile ad hoc est dynamique (du fait du changement fréquent de la topologie) et ne présente aucune forme d'administration centralisée des échanges.

L'étude bibliographique réalisée a permis de dégager un grand nombre de solutions apportées au problème d'exclusion mutuelle et de K-exclusion mutuelle dans les systèmes distribués classiques et dans l'environnement mobile ad hoc. Toutes ces solutions sont d'un apport considérable vu l'originalité de la plupart. Néanmoins, dans le contexte des réseaux mobiles ad hoc, les algorithmes proposés sont une reprise de leurs prédécesseurs en termes d'adaptation ou de généralisation. Par exemple, Le seul algorithme de K-exclusion mutuelle dans les réseaux mobiles ad hoc connu 'algorithme KRL' [60] présente une généralisation de l'algorithme d'exclusion mutuelle dans les réseaux mobiles ad hoc de [43] qui est lui même une adaptation d'un algorithme d'exclusion mutuelle dans les systèmes distribués de [5]. Les résultats obtenus par l'algorithme [60] restent encore à améliorer dans le sens où le nombre de messages générés doit diminuer avec une consommation optimale de la puissance de calcul et de stockage; par exemple en ne faisant intervenir dans l'exécution de l'algorithme que les nœuds qui ont sollicité l'entrée en section critique.

Notre solution s'écarte de toutes celles qui la précèdent: elle ne généralise ni n'adapte aucune autre solution, elle se veut indépendante de toute topologie et paramétrée en fonction de l'état du système afin d'interagir avec tous les mouvements qui peuvent survenir. Elle est orientée recherche de jeton et suppose l'existence de K jetons dans le système.

Le réseau étant quelconque et les nœuds ne connaissant que leurs voisins directs, il est nécessaire pour un noud i demandeur d'accès à une section critique, de définir une stratégie de recherche de jeton; d'où l'exploitation du concept des heuristiques; Il s'agit de mettre la requête sur la trace du jeton ayant traversé le nœud le plus récemment ou à défaut, envoyer la requête à un voisin 'quelconque'. Cette stratégie pose le problème des chemins cycliques. En effet, rien n'empêche la requête de passer par un nœud déjà exploré. D'où la nécessité de sauvegarder ces chemins. La forme la plus élémentaire d'un chemin est la liste. En effet, au fur et à mesure qu'un nœud effectue des sauts, il est possible d'insérer dans la liste l'identité des nœuds traversés; Mais, par souci d'optimisation de la taille du message de requête nous avons opté pour une forme plus réduite. Il s'agit de maintenir de manière distribuée et dans une seule valeur T le produit des attributs de tous les nœuds traversés (l'attribut d'un nœud i étant le $(i+1)^{ième}$ nombre premier. Une fois le chemin d'une requête connu, le jeton n'a qu'à suivre le chemin inverse pour arriver à destination. En outre, la formation de nouveaux liens permet d'optimisation de ce dernier.

Face aux contraintes imposées par l'environnement mobile ad hoc, nous avons proposé plusieurs techniques qui seront sollicitées et mises en œuvre par le système chaque fois que c'est nécessaire dans le but de favoriser la sûreté de l'algorithme; 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 section critique dans le service de K-exclusion mutuelle n'est que partielle. En effet, ces nœuds ont pour rôle d'acheminer les requêtes reçues selon les règles prédéfinies et de maintenir la structure '*Trace*' afin de pouvoir acheminer les jetons reçus vers leurs destinations.

L'étude de simulation a été effectuée en utilisant le simulateur NS2 sous Linux. Ce simulateur fournit un environnement complet à plusieurs plates formes telles que les réseaux filaires, les réseaux mobiles, etc... et inclue plusieurs outils tels que Tcl, Otcl, etc...

Après l'implémentation de l'algorithme, la première étape de la simulation était de définir les paramètres de mesure tels que la connectivité, la mobilité et d'autres paramètres liés à l'environnement tels que le rayon de communication, la surface de simulation, etc... ainsi que les critères de performances à mesurer à savoir: le nombre de messages générés par entrée en section critique, le temps de réponse, le délai de synchronisation et le nombre de relances. Les paramètres de mesure n'ont pas été choisis aléatoirement car en plus des simulations réalisées, il a été question de comparer nos résultats avec ceux de l'algorithme KRL. En effet, nous nous sommes appuyés sur les définitions données dans la simulation de l'algorithme KRL [60] pour déterminer nos propres paramètres et critères. Ceci a rendu la comparaison des deux algorithmes (KRL et *Ksearch*) possible car nous avons essayé de nous rapprocher dans la mesure du possible des conditions de simulation de l'algorithme KRL. La comparaison a été possible seulement dans le cas statique et le cas mobile avec une faible mobilité. Et les critères comparés étaient: 'le nombre moyen de messages générés par entrée en section critique' et 'la durée d'attente moyenne pour une entrée en section critique' car ces deux critères seuls, ont été évalués dans la simulation de l'algorithme KRL [60].

La simulation a été effectuée pour les deux types de réseaux: Les réseaux statiques et les réseaux mobiles ad hoc. Dans le premier type, l'intérêt est porté sur la topologie. En effet, deux topologies ont été traitées: une topologie en arbre et une topologie quelconque. Dans le deuxième type, deux cas de mobilité ont été proposés: la forte et la faible mobilité.

D'après les graphes obtenus des différentes simulations, il semble que les résultats offerts par notre algorithme sont meilleurs que ceux de l'algorithme KRL. Néanmoins, les conditions de simulation dans lesquels l'algorithme KRL a été exécuté ne sont pas connues dans le détail. Par exemple: les conditions de lancement des processus, les fichiers de scénario de mobilité, etc… ; Ce qui ne nous permet pas de réaliser une étude comparative qualitative et objective.

En général, l'algorithme que nous avons proposé offre de meilleures performances dans les conditions de forte charge. 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 de l'algorithme. En effet, en lisant les tableaux (présentés en annexe 2) pour toute performance confondue, dans le sens vertical nous pouvons remarquer que les écarts entre les valeurs ne sont pas très grands, qu'il s'agisse de performances en terme de délai d'attente, de nombre des messages ou de délai de synchronisation. Ceci se reflète sur les graphes obtenus: par exemple, les courbes obtenues à partir de mesures effectuées sur des structures d'arbre forment un faisceau de courbes ayant la même allure. Ceci concerne toutes les performances calculées; il en est de même pour les courbes issues de topologies générales dans le cas statique de faible mobilité ou de forte mobilité.

Malgré que l'influence de la connectivité sur les performances soit nulle, négligeable ou faible par rapport à toutes les simulations effectuées, il existe un facteur important et qui n'est visible ni sur les tableaux de mesures ni sur les graphes, c'est celui du temps de calcul au sein de l'algorithme. En effet, 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. Nous déduisons par la pratique que l'utilisation des nombres premiers nécessite une grande capacité de calcul. Ce résultat, était prévu théoriquement (voir paragraphe 'Discussion' du chapitre 'Conception').

En consultant les fichiers résultats générés par les différentes simulations, nous constatons que notre algorithme 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 sections critiques avec un nombre négligeable de relances en général.

Les simulations réalisées n'ont pas tenu compte de tous les aspects paramétrés de l'algorithme. Par exemple: lorsque le nombre de jetons est élevé (cas de 8 et 5 jetons), si un nœud i demandeur d'accès ou en section critique reçoit une requête et n'a pas de jetons libres, celle-ci est immédiatement acheminée si la file contient une seule requête en attente. Dans ce cas, il n'est pas tenu compte du nombre de sauts effectués par la requête reçue et du nombre de refus qu'elle a subi. Ceci montre qu'il

y a encore des aspects inconnus de l'algorithme qu'il faut évaluer afin de déterminer leur influence sur les performances. Parallèlement, la simulation de l'algorithme 'KRL' dans les mêmes conditions que celles de l'algorithme 'Ksearch' est de rigueur afin que la comparaison soit juste.

Annexe 1 Préliminaires sur l'environnement mobile Ad Hoc

I. Introduction

L'évolution technique dans le développement des ordinateurs portables et le déploiement rapide de la technologie des réseaux sans fil fournissent la base pour un nouvel environnement de calcul appelé ***environnement mobile***.

On peut classer les réseaux mobiles en deux catégories :

- Les réseaux avec infrastructure (infrastructured) qui utilisent le modèle de communication cellulaire, ce modèle est composé de deux ensembles d'entités distinctes: un grand nombre d'hôtes mobiles et un nombre relativement réduit de stations fixes. L'ensemble des stations fixes reliées par des liaisons filaires constitue le réseau fixe (statique). Certains sites fixes appelés ***stations de base*** sont munis d'une interface de communication sans fil pour la communication directe avec les unités mobiles localisées dans une zone géographique appelée ***cellule***. Pour envoyer un message d'une unité mobile UM1 à une autre unité mobile UM2, UM1 envoi le message à sa station de base SB1 à travers la liaison sans fil. SB1 envoi le message à la station de base de UM2 appelée SB2 qui à son tour l'envoi à UM2. Dans ce modèle, la mobilité est représentée par la migration des hôtes mobiles entre les cellules. Le ***handoff*** est le processus enclenché quand un mobile actif change de station de base.
- Les réseaux sans infrastructures (infrastructureless) qui sont les réseaux mobiles ad hoc ne requièrent aucune infrastructure fixe. En d'autres termes, un réseau mobile ad hoc est un réseau temporaire formé d'un ensemble de sites communiquant entre eux sans aucune forme d'administration centralisée où chaque nœud se comporte non seulement comme une station mais aussi comme un routeur. Un réseau mobile ad hoc possède un certain nombre de caractéristiques qui imposent de nouvelles exigences dans l'implémentation des algorithmes distribués. Les plus importants sont le changement fréquent de topologie du réseau, la déconnexion fréquente d'un hôte, l'environnement possible en plus des ressources modestes des équipements mobiles.

Dans ce qui suit, nous présentons les réseaux mobiles ad hoc et leurs principales caractéristiques.

II. Description des réseaux mobiles ad hoc

Un réseau mobile ad hoc appelé MANET (Mobile Ad hoc NETwork) consiste en une population d'unités mobiles qui se déplacent dans un territoire quelconque et dont le seul moyen de communication est l'utilisation d'interfaces sans fil telles que les voies hertziennes utilisant des antennes pouvant être omnidirectionnels (broadcast), bidirectionnels (point-to-point), ou une combinaison des deux. A un instant donné, en fonction de la position des nœuds, de la configuration de leurs émetteurs-récepteurs, des niveaux de puissance de la transmission et d'interférence entre les canaux, il existe une connectivité entre les nœuds sous forme d'un graphe aléatoire. Cette topologie peut changer avec le temps en fonction du mouvement des nœuds ou d'ajustement de leurs paramètres d'émission-réception. On peut modéliser un MANET par un graphe orienté $G_t=(V_t,E_t)$, où V_t représente l'ensemble des nœuds et E_t représente l'ensemble des liens qui existent entre ces nœuds. Si $e=(u,v)\in E_t$, cela veut dire que les nœuds u et v sont en mesure de communiquer directement à l'instant t. La topologie du réseau peut changer à tout moment, elle est dynamique et imprévisible donc le mouvement et la déconnexion des unités mobiles où la fragmentation du réseau sont possibles et le nombre de nœuds n'est pas fixé définitivement.

Les réseaux mobiles ad hoc ne requièrent aucune infrastructure fixe (infrastructureless). En d'autres termes, un réseau mobile ad hoc est un réseau temporaire formé d'un ensemble de sites communiquant entre eux sans aucune forme d'administration centralisée où chaque nœud se comporte non seulement comme une station mais aussi comme un routeur. En effet, deux nœuds communiquent soit directement si le lien sans fil est établi soit en passant les messages à travers des liens incluant un

ou plusieurs nœuds du système. Ceci nous conduit à définir les caractéristiques des réseaux mobiles ad hoc comme suit:

- *Absence d'infrastructure*: Les nœuds d'un réseau mobile ad hoc sont complètement autonomes et communiquent via des interfaces d'émission réception sans fil; les hôtes mobiles sont responsables d'établir et de maintenir la connectivité du réseau d'une manière continue. Un MANET peut être isolé, toutefois, il peut avoir des passerelles ou des interfaces qui le relient à un réseau fixe;
- *La composante physique*: Les hôtes mobiles sont alimentés par des sources d'énergie autonomes comme les batteries dont l'énergie est limitée; En outre, l'utilisation d'un médium de communication partagé fait que la bande passante réservée à un hôte est modeste.
- *Topologie Dynamique*: Les nœuds sont libres de se déplacer arbitrairement. Par conséquent la topologie du réseau est susceptible de changer aléatoirement et rapidement d'une manière imprévisible;
- *Nombre d'unités mobiles variable*: Ce nombre peut augmenter ou diminuer de manière imprévisible;
- *Sécurité physique limitée*: Les réseaux mobiles sont plus vulnérables par le paramètre de sécurité, et plus menacés par les attaques que les réseaux classiques. Ceci est dû aux limitations physiques qui font que le contrôle des données doit être minimisé.

III. Applications des MANET

Le concept de réseau mobile ad hoc essaie d'étendre la mobilité à toutes les composantes de l'environnement. Il permet de relier des ordinateurs là où il serait difficile ou trop coûteux d'entretenir un câble. Ce sont alors les ondes radio qui remplacent les fils. Aucune limitation n'est faite sur la taille d'un réseau mobile ad hoc.

Beaucoup d'efforts ont été déployés pour le développement des réseaux mobiles ad hoc. Au début, ces réseaux étaient principalement orientés vers des applications militaires. En effet, certains équipements militaires sont dotés d'une sorte d'ordinateurs dont l'interconnexion grâce à la technologie des réseaux mobiles ad hoc permet de maintenir un réseau d'informations entres les soldats, les véhicules, …etc. D'autres applications sont possibles avec la technologie des MANET, nous citons par exemple:

- *Les opérations de secours* en cas d'incendie, d'inondation ou de tremblement de terre dans des régions désastreuses où aucune infrastructure de réseau filaire n'existe ;
- *Conférences et enseignement à distance*: Un réseau mobile ad hoc est réalisable pour un groupe de personnes qui veulent créer un réseau multimédia autonome et temporaire qui relie les différentes unités portables (Palmtops, PDA, Notebook);
- *Applications industrielles et commerciales* nécessitant un échange coopératif des données ;
- *Contrôle d'environnement*: Des petits véhicules peuvent équipés de caméras et des détecteurs de son peuvent être utilisés dans des régions déterminées afin de collecter un ensemble d'informations et de l'envoyer à travers le MANET à une station qui les traite.

IV. Le routage dans les MANET

Généralement, le routage est une méthode d'acheminement des informations à la bonne destination à travers un réseau de connexion donné. Le problème de routage consiste, pour un réseau dont les arcs, les nœuds et les capacités sur les arcs sont fixés, à déterminer un chemin optimal des paquets à travers le réseau au sens d'un certain critère de performance.

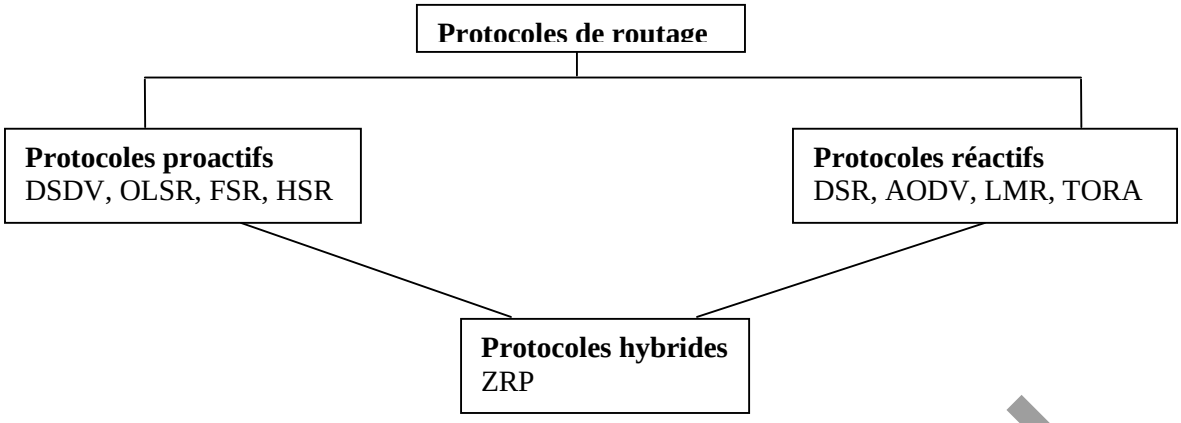
Comme nous l'avons déjà vu, l'architecture d'un réseau mobile ad hoc est caractérisée par une absence d'infrastructure fixe préexistante, à l'inverse des réseaux de communication classiques (filaires ou cellulaires). Un réseau mobile ad hoc doit s'organiser automatiquement de façon à être deployable rapidement et pouvoir s'adapter aux conditions de propagation, au trafic et aux différents mouvements pouvant intervenir au sein des unités mobiles. Dans le but d'assurer la connectivité du réseau, malgré l'absence d'infrastructure fixe et la mobilité des stations, chaque nœud est susceptible d'être mis à contribution pour participer au routage et pour retransmettre les paquets d'un nœud qui

n'est pas en mesure d'atteindre sa destination ; tout nœud joue ainsi le rôle de station et de routeur. Le problème qui se pose dans le contexte des réseaux mobiles ad hoc est l'adaptation de la méthode d'acheminement utilisée avec un grand nombre d'unités existant dans un environnement caractérisé par de modestes capacités de calcul et de sauvegarde. Dans le cas où le nœud destination se trouve dans la portée de communication du nœud source, le routage devient évident et aucun protocole de routage n'est initié. Malheureusement, ce cas est généralement rare dans les réseaux mobiles ad hoc. Une station source peut avoir besoin de transférer des données à une autre station qui ne se trouve pas dans sa portée de communication, par exemple dans le réseau illustré par la figure suivante l'unité mobile W n'est pas dans la portée de communication de l'unité U et vice versa. Dans le cas où l'unité U veut transférer des paquets à W, elle doit utiliser les services de l'unité V dans l'envoi des paquets, puisque l'unité V contient dans sa portée de communication les unités U et W.

Trois unités mobiles constituant un réseau mobile ad hoc

Le problème de routage est plus compliqué à cause de la non uniformité de la transmission sans fil et de la possibilité du déplacement imprévisible de tous les nœuds concernés par le routage.

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



Classification des protocoles de routage

Annexe 2

Echantillons des mesures prises lors de la simulation de notre algorithme de K-exclusion mutuelle dans les réseaux mobiles ad hoc

Dans les tableaux de mesures que nous présentons, nous considérons nos propres mesures et nous incluons les mesures effectuées par Walter [60]. Les paramètres et critères considérés par les deux protocoles KRL et Ksearch sont donnés dans le tableau suivant:

Paramètres/critères considérés / Algorithme	Charge	Nombre de jetons	Critères mesurés
KRL	1,10,100,1000	1,3,5	-Nombre de messages/entrée en CS - Délais d'attente pour une entrée en CS
Ksearch	1,5,10,50,100,500,1000	1,3,5,8	-Nombre de messages/entrée en CS - Délais d'attente pour une entrée en CS - Délais de synchronisation - Nombre de relances pour une entrée en CS

Ainsi la comparaison n'est possible que par rapport aux éléments en commun. Pour cela les mesures correspondantes aux mêmes conditions sont prises dans le même tableau. En outre les valeurs de mesures prises dans [60] pour le cas de réseau dynamique avec forte mobilité sont considérées avec nos mesures dans le cas de réseau dynamique avec une faible mobilité. Dans ce qui suit, nous présentons les mesures:

I. Cas de réseau statique

- **Nombre de jetons = 8**

Charge / Topologie	1	5	10	50	100	500	1000
Arbre binaire	8	8	7	4	3	5	5
Arbre à trois branches	5	7	6	3	2		
Arbre à cinq branches	5	3	4	2	2		3
Connectivité=20%	4	8	7	6	4	4	4
Connectivité=80%	2	3	3	5	5	7	6

Table 1 Nombre moyen de messages par entrée en section critique

Charge / Topologie	1	5	10	50	100	500	1000
Arbre binaire	14	13			2	6	7
Arbre à trois branches	15	12	10	2		3	3
Arbre à cinq branches	9	9	4	2	2	2	2
Connectivité=20%	5	7	8	7	6	5	6
Connectivité=80%			5	6	5	6	6

Table 2 Délai d'attente moyen pour chaque entrée en section critique

Charge / Topologie	1	5	10	50	100	500	1000
Arbre binaire	5	6	7	4	3	5	5
Arbre à trois branches	4	5	5	3	2	3	3
Arbre à cinq branches	4	3	3	2	2	2	3
Connectivité=20%	2	8	6	6	5	4	4
Connectivité=80%	1	2	3	5	5	5	6

Table 3 Délai moyen de synchronisation

Charge / Topologie	1	5	10	50	100	500	1000
Arbre binaire	0.01	0.01	0.02	0.00	0.00	0.01	0.01
Arbre à trois branches	0.07	0.02	0.02	0.00	0.00	0.00	0.00
Arbre à cinq branches	0.01	0.04	0.00	0.00	0.00	0.00	0.00
Connectivité=20%	0.00	0.00	0.00	0.00	0.00	0.00	0.01
Connectivité=80%	0.00	0.00	0.01	0.00	0.00	0.00	0.00

Table 4 Nombre de relances

- **Nombre de jetons = 5**

Algorithme Ksearch							
Charge / Topologie	1	5	10	50	100	500	1000
Arbre binaire	10	10	9	6	5	5	5
Arbre à trois branches	6	8	8	4	4	4	3
Arbre à cinq branches	7	6	6	3	3	3	3
Connectivité=20%	6	8	8	7	6	5	5
Connectivité=80%	7	7	6	7	7	7	8
Algorithme KRL							
Connectivité=20%	1	/	4	/	11	/	11
Connectivité=80%	3	/	8	/	20	/	13

Table 5 Nombre moyen de messages par entrée en section critique

Algorithme Ksearch							
Charge / Topologie	1	5	10	50	100	500	1000
Arbre binaire	17	19	17	8	7	7	5
Arbre à trois branches	14	13	13	3	3	4	3
Arbre à cinq branches	15	13	8	2	2	2	2
Connectivité=20%	7	8	10	9	7	8	8
Connectivité=80%	12	11	10	8	8	8	9
Algorithme KRL							
Connectivité=20%	12	/	28	/	15	/	5
Connectivité=80%	12	/	28	/	14	/	2

Table 6 Délai d'attente moyen pour chaque entrée en section critique

Charge / Topologie	1	5	10	50	100	500	1000
Arbre binaire	7	8	8	6	5	5	5
Arbre à trois branches	4	5	7	4	4	4	3
Arbre à cinq branches	6	6	5	3	3	3	3
Connectivité=20%	5	8	8	7	6	5	6
Connectivité=80%	6	6	6	7	8	7	7

Table 7 Délai moyen de synchronisation

Charge / Topologie	1	5	10	50	100	500	1000
Arbre binaire	0.08	0.09	0.03	0.02	0.01	0.01	0.00
Arbre à trois branches	0.03	0.01	0.03	0.00	0.00	0.00	0.00
Arbre à cinq branches	0.07	0.05	0.01	0.00	0.00	0.00	0.00
Connectivité=20%	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Connectivité=80%	0.00	0.00	0.03	0.00	0.00	0.02	0.00

Table 8 Nombre de relances

- **Nombre de jetons =**

Algorithme Ksearch							
Charge / Topologie	1	5	10	50	100	500	1000
Arbre binaire	12	13	13	9	7	3	7
Arbre à trois branches	10	15	17	9	6	8	8
Arbre à cinq branches	11	17	16	10	6	6	7
Connectivité=20%	3	4	6	7	7	7	7
Connectivité=80%	3	5	6	7	8	11	8
Algorithme KRL							
Connectivité=20%	2	/	6	/	12	/	12
Connectivité=80%	5	/	12	/	25	/	15

Table 9 Nombre moyen de messages par entrée en section critique

Algorithme Ksearch							
Charge / Topologie	1	5	10	50	100	500	1000
Arbre binaire	27	22	25	16	9	9	9
Arbre à trois branches	23	29	34	14	7	10	10
Arbre à cinq branches	24	34	35	18	7	7	7
Connectivité=20%	21	13	25	29	25	11	15
Connectivité=80%	19	9	19	13	12	15	12
Algorithme KRL							
Connectivité=20%	22	/	46	/	20	/	6
Connectivité=80%	22	/	46	/	19	/	3

Table 10 Délai d'attente moyen pour chaque entrée en section critique

Charge / Topologie	1	5	10	50	100	500	1000
Arbre binaire	9	12	12	10	7	7	7
Arbre à trois branches	8	12	14	10	7	8	8
Arbre à cinq branches	7	12	11	9	7	7	7
Connectivité=20%	3	5	6	7	7	7	7
Connectivité=80%	3	5	6	8	8	11	8

Table 11 Délai moyen de synchronisation

Charge / Topologie	1	5	10	50	100	500	1000
Arbre binaire	0.21	0.12	0.13	0.05	0.03	0.01	0.01
Arbre à trois branches	0.15	0.17	0.23	0.02	0.00	0.02	0.03
Arbre à cinq branches	0.17	0.28	0.25	0.11	0.01	0.02	0.02
Connectivité=20%	0.20	0.20	0.18	0.45	0.20	0.07	0.08
Connectivité=80%	0.25	0.40	0.20	0.20	0.02	0.07	0.0

Table 12 Nombre de relances

- **<u>Nombre de jetons = 1</u>**

Algorithme Ksearch							
Charge / Topologie	1	5	10	50	100	[illegible]	[illegible]000
Arbre binaire	53	67	67	45	[illegible]	12	13
Arbre à trois branches	53	61	64	39	[illegible]	10	1[illegible]
Arbre à cinq branches	48	51	57	[illegible]	19	7	[illegible]
Connectivité=20%	11	6	6	[illegible]	[illegible]	7	[illegible]
Connectivité=80%	9	8	[illegible]0	11	14	14	16
Algorith[illegible]e KRL							
Connectivité=20%	10	/	[illegible]	[illegible]	13	[illegible]	13
Connectivité=80%	27	/	26	/	[illegible]0	/	17

Table 13 Nombre moyen de messages par entrée en section critique

Algorithme Ksearch							
Charge / Topologie	1	5	10	50	100	500	1000
Arbre binaire	[illegible]7	58	76	58	58	18	21
Arbre à trois branches	[illegible]	71	83	73	49	12	15
Arbre à cinq branches	67	71	86	62	32	9	11
Connectivité=20%	48	36	16	86	53	17	20
Connectivité=80%	42	30	30	44	39	25	29
Algorithme KRL							
Connectivité=20%	85	/	78	/	28	/	7
Connectivité=80%	86	/	78	/	28	/	4

Table 14 Délai d'attente moyen pour chaque entrée en section critique

Charge / Topologie	1	5	10	50	100	500	1000
Arbre binaire	33	43	44	39	30	12	13
Arbre à trois branches	34	36	40	34	26	10	12
Arbre à cinq branches	31	36	39	28	19	7	8
Connectivité=20%	5	7	10	8	8	7	8
Connectivité=80%	6	9	12	14	14	13	13

Table 15 Délai moyen de synchronisation

Charge / Topologie	1	5	10	50	100	500	1000
Arbre binaire	1.50	1.46	1.54	0.95	0.68	0.06	0.09
Arbre à trois branches	1.47	1.50	1.55	0.93	0.42	0.04	0.07
Arbre à cinq branches	1.38	1.21	1.36	0.67	0.20	0.03	0.06
Connectivité=20%	2.56	1.68	1.75	1.87	0.96	0.19	0.34
Connectivité=80%	1.50	1.46	1.11	0.85	1.25	1.01	0.98

Table 16 Nombre de relances

II. Cas dynamique de faible mobilité

- **Nombre de jetons = 8**

Charge / Topologie	1	5	10	50	100	500	1000
Connectivité=20%	7	7	7	10	10	9	16
Connectivité=80%	9	9	9	7	8	7	7

Table 17 Durée d'attente moyenne pour chaque entrée en section critique

Charge / Topologie	1	5	10	50	100	500	1000
Connectivité=20%	6	6	6	6	6	7	8
Connectivité=80%	5	6	6	6	7	6	5

Table 18 Nombre moyen de messages par entrée en section critique

Charge / Topologie	1	5	10	50	100	500	1000
Connectivité=20%	5	6	6	6	6	7	7
Connectivité=80%	3	5	6	6	7	6	5

Table 19 Délai moyen de synchronisation

Charge / Topologie	1	5	10	50	100	500	1000
Connectivité=20%	0.19	0.25	0.24	0.53	0.36	0.49	0.51
Connectivité=80%	0.10	0.16	0.02	0.03	0.09	0.01	0.01

Table 20 Nombre de relances

- **Nombre de jetons = 5**

Algorithme Ksearch							
Charge / Topologie	1	5	10	50	100	500	1000
Connectivité=20%	17	8	12	12	13	14	1
Connectivité=80%	10	12	15	12	9	10	10
Algorithme KRL							
Connectivité=20%	11	/	12	/	4	/	3
Connectivité=80%	12	/	19	/	25	/	2

Table 21 Durée d'attente moyenne pour chaque entrée en section critique

Algorithme Ksearch							
Charge / Topologie	1	5		50	100	500	1000
Connectivité=20%	8	8	1		9	9	9
Connectivité=80%	8	10	1	10	8	9	9
Algorithme KRL							
Connectivité=20%	10	/	40	/	170	/	373
Connectivité=80%	8	/	50	/	80	/	328

Table 22 Nombre moyen de messages par entrée en section critique

Charge / Topologie	1	5	10	50	100	500	1000
Connectivité=20%	7	8	11	8	9	9	9
Connectivité=80%	7	9	10	10	8	9	9

Table 23 Délai moyen de synchronisation

Charge / Topologie	1	5	10	50	100	500	1000
Connectivité=20%	0.27	0.31	0.37	0.49	0.45	0.48	0.44
Connectivité=80%	0.27	0.32	0.33	0.21	0.07	0.12	0.14

Table 24 Nombre de relances

- **Nombre de jetons = 3**

Algorithme Ksearch							
Charge / Topologie	1	5	10	50	100	500	1000
Connectivité=20%	15	23	20	20	31	18	17
Connectivité=80%	14	18	19	23	20	22	20
Algorithme KRL							
Connectivité=20%	20	/	22	/	9	/	4
Connectivité=80%	21	/	22	/	8	/	2

Table 25 Durée d'attente moyenne pour chaque entrée en section critique

Algorithme Ksearch							
Charge / Topologie	1	5	10	50	100	500	1000
Connectivité=20%	5	5	6	9	9	11	13
Connectivité=80%	5	7	8	10	12	11	13
Algorithme KRL							
Connectivité=20%	16	/	40	/	70	/	408
Connectivité=80%	13	/	58	/	80	/	336

Table 26 Nombre moyen de messages par entrée en section critique

Charge / Topologie	1	5	10	50	100	500	1000
Connectivité=20%	5	4	6	11	11	12	14
Connectivité=80%	3	6	9	10	12	11	14

Table 27 Délai moyen de synchronisation

Charge / Topologie	1	5	10	50	100	500	1000
Connectivité=20%	0.94	1.17	1.33	1.75	1.85	0.60	[illegible]
Connectivité=80%	0.86	0.88	0.89	0.54	0.60	0.67	0.5[illegible]

Table 28 Nombre de relances

- **Nombre de jetons = 1**

Algorithme Ksearch							
Charge / Topologie	1	5	10	[illegible]	[illegible]	500	[illegible]
Connectivité=20%	12	9	22	[illegible]	32	26	24
Connectivité=80%	7	8	[illegible]	[illegible]	44	[illegible]	47
Algorithme [illegible]							
Connectivité=20%	70	/	75	/	[illegible]	/	10
Connectivité=80%	79	/	79	/	35	/	4

Table 29 Durée d'attente moy[illegible]e pour chaque entrée en section critique

Algorithme Ksearch							
Charge / Topologie	1	5	10	50	100	500	1000
Connectivité=20%	4	5	6	8	8	23	21
Connectivité=80%	4	5	7	7	10	20	14
Algorithme KRL							
Connectivité=20%	43	/	43	/	100	/	496
Connectivité=80%	44	/	44	/	98	/	331

Table 30 Nombre moyen de messages par entrée en section critique

Charge / Topologie	1	5	10	50	100	500	1000
Connectivité=20%	5	6	7	15	18	37	23
Connectivité=80%	4	8	9	14	15	30	18

Table 31 Délai moyen de synchronisation

Charge / Topologie	1	5	10	50	100	500	1000
Connectivité=20%	2.12	2.90	2.52	2.84	3.34	2.72	2.45
Connectivité=80%	2.28	2.17	2.72	2.47	2.83	2.08	2.40

Table 32 Nombre de relances

III. Cas dynamique de forte mobilité

- **Nombre de jetons = 8**

Charge / Topologie	1	5	10	50	100	500	1000
Connectivité=20%	9	11	10	10	10	17	14
Connectivité=80%	6	8	8	7	7	9	9

Table 33 Durée d'attente moyenne pour chaque entrée en section critique

Charge / Topologie	1	5	10	50	100	500	1000
Connectivité=20%	4	6	6	6	6	13	10
Connectivité=80%	4	7	7	7	7	8	8

Table 34 Nombre moyen de messages par entrée en section critique

Charge / Topologie	1	5	10	50	100	500	1000
Connectivité=20%	4	6	5	6	6	11	8
Connectivité=80%	3	6	7	7	7	7	8

Table 35 Délai moyen de synchronisation

Charge / Topologie	1	5	10	50	100	500	1000
Connectivité=20%	0.20	0.41	0.35	0.34	0.40	0.41	0.41
Connectivité=80%	0.08	0.17	0.12	0.04	0.04	0.07	0.06

Table 36 Nombre de relances

- **Nombre de jetons = 5**

Charge / Topologie	1	5	10	50	100	500	1000
Connectivité=20%	30	12	22	16	19	14	21
Connectivité=80%	9	13	12	16	10	13	11

Table 37 Durée d'attente moyenne pour chaque entrée en section critique

Charge / Topologie	1	5	10	50	100	500	1000
Connectivité=20%	7	11	11	12	11	14	27
Connectivité=80%	7	12	11	17	11	15	17

Table 38 Nombre moyen de messages par entrée en section critique

Charge / Topologie	1	5	10	50	100	500	1000
Connectivité=20%	8	11	11	12	10	11	25
Connectivité=80%	8	11	12	18	11	15	18

Table 39 Délai moyen de synchronisation

Charge / Topologie	1	5	10	50	100	500	1000
Connectivité=20%	0.48	0.50	0.49	0.49	0.64	0.46	0.67
Connectivité=80%	0.32	0.40	0.28	0.41	0.11	0.40	0.45

Table 40 Nombre de relances

- **Nombre de jetons = 3**

Charge / Topologie	1	5	10	50	100	500	1000
Connectivité=20%	20	22	17	34	30	29	37
Connectivité=80%	14	17	19	22	24	28	32

Table 41 Durée d'attente moyenne pour chaque entrée en section critique

Charge / Topologie	1	5	10	50	100	500	1000
Connectivité=20%	4	6	7	11	14	17	23
Connectivité=80%	5	9	10	20	10	10	19

Table 42 Nombre moyen de messages par entrée en section critique

Charge / Topologie	1	5	10	50	100	500	1000
Connectivité=20%	4	6	8	12	16	17	23
Connectivité=80%	6	9	10	21	10	10	20

Table 43 Délai moyen de synchronisation

Charge / Topologie	1	5	10	50	100	500	1000
Connectivité=20%	0.80	0.86	1.42	1.50	2.03	0.91	0.98
Connectivité=80%	0.48	0.50	0.51	0.89	1.91	1.20	0.90

Table 44 Nombre de relances

- **Nombre de jetons = 1**

Charge / Topologie	1	5	10	50	100	500	1000
Connectivité=20%	15	9	15	19	25	34	35
Connectivité=80%	15	20	24	27	39	40	42

Table 45 Durée d'attente moyenne pour chaque entrée en section critique

Charge / Topologie	1	5	10	50	100	500	1000
Connectivité=20%	7	7	7	10	11	37	33
Connectivité=80%	5	9	10	13	16	18	21

Table 46 Nombre moyen de messages par entrée en section critique

Charge / Topologie	1	5	10	50	100	500	1000
Connectivité=20%	3	8	11	21	22	53	
Connectivité=80%	6	11	15	25	27	31	32

Table 47 Délai moyen de synchronisation

Charge / Topologie	1	5	10	50	100	500	1000
Connectivité=20%	2.43	3.04	2.92	3.4	3.	2.40	2.0
Connectivité=80%	1.99	2.17	2.7	2.81	3.03	3.05	3.0

Table 48 Nombre de relances

BIBLIOGRAPHIE

L'exclusion mutuelle distribuée

[1] Suzuki and Kasami. 'A distributed mutual exclusion algorithm'. ACM Transactions on computer systems; Novembre 1985;

[2] Ricart and Agrawala. 'Author response to 'On mutual exclusion in computer networks' by Calvalho and Roucairol'.Communication of the ACM; Fevrier 1983;

[3] Mishra and Srimani. 'Fault-tolerant mutual exclusion algorithms'. Journal of systems software. Fevrier 1990;

[4] Nishio, Li, K.F. and Manning. 'A resilient mutual exclusion algorithm for computer networks'. IEEE Transactions on parallel and distributed systems; Juillet 1990;

[5] K. Raymond. 'A tree based algorithm for distributed mutual exclusion '. ACM Transactions on computer systems; Fevrier 1989;

[6] Neilsen and Mizuno. 'A DAG based algorithm for distributed computing systems'. 11th International conference on distributed computing systems; Mai 1991;

[7] Helary, Plouzeau and Raynal. 'A distributed algorithm for mutual exclusion in an arbitrary network'. Computer Journal; 1988;

[8] Naimi, Trehel and Arnold. 'A log(n) distributed mutual exclusion algorithm based on the path reversal'. CNRS: unité associée 040822 and the C3 co-ordinated research program;

[9] Naimi and 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; 1987;

[10] J.M.Hélary and A.Mostefaoui 'A O($\log_2$ n) fault-tolerant distributed mutual exclusion algorithme based on open-cube structure' Rapport de recherché N°2047, INRIA, Septembre 1993;

[11] Martin G.Velazquez 'A survey of distributed Mutual Exclusion Algorithms'.Colorado state University.Technical report CS-93-116; September 1993;

[12] Mizuno, Neilsen and Rao. 'A token based distributed mutual exclusion algorithm based on quorum agreements'. Conference on distributed computing systems; Mai 1991;

[13] Lelann G. 'distributed systems toward a formal approach' IFIP Congress, Toronto, Aout 1977 paper 155-160;

[14] Singhal. 'A heuristically-aided algorithm for mutual exclusion in distributed systems'. IEEE Transactions on computers; Mai 1989;

[15] D.Manivannan and M.singhal 'A decentralized token génération Sheme For Token Based Mutual Exclusion Algorithms' International Journal of computer science and Engineering janvier 1996;

[21] Gifford. 'Weighted voting for replicated data'. In proceedings of 7th Symposium on operating systems; 1979;

[22] Garcia, Molina and Barbara. 'How to assign votes in a distributed systems principles'. Journal of the ACM; 1985;

[23] L. Lamport 'Time, clocks, and the ordring of events in a distributed system'. Communications of the ACM; Juillet 1977;

[24] Ricart and Agrawala. 'An optimal algorithm for mutual exclusion in computer networks'. Communications of the ACM; Janvier 1981;

[25] Calvalho and Roucairol. 'On mutual exclusion in computer networks'. Technical correspondance; Communication of the ACM; Fevrier 1983;

[26] M. Maekawa. 'A sqrt(n) algorithm for mutual exclusion in decentralized systems'. ACM transactions on computer systems; Mai 1985;

[27] Sanders. 'The information structure of distributed mutual exclusion algorithms'. ACM Transactions on computer systems; Aout 1987;

[28] Agrawal and El Abadi. 'An efficient and fault-tolerant solution for distributed mutual exclusion'. ACM Transactions on computer systems; Fevrier 1991;

[29] Singhal. 'A dynamic information structure mutual exclusion algorithm for distributed systems'. IEEE Transactions on parallel and distributed systems; Janvier 1992;

[31] Chang Y.I., Singhal M. et Liu M.T. 'A hybrid approach to mutual exclusion for distributed systems' Proc. 1990 Annual International computer software and application conference, IEEE computer society, Chicago, paper 289-294;

[32] Housni, Trehel algorithme hybride?
[33] Chang, Y.I., M. Singhal and M.T. Liu 'A fault tolerant algorithm for distributed mutual exclusion'; IEEE 1990
[34] A. Bouabdallah,J.C. Konig 'An improvement of the Maekawa's mutual exclusion algorithm to make it fault tolerant' CNRS .
[35] Raynal. 'Prime numbers as a tool to design distributed algorithms'. Information processing Letters; Octobre 1989;
[36] D.Agrawal and A.El Abadi. 'The Generalized Tree Quorum Protocol: An Efficient Approach for Managing Replicated Data'. ACM Transactions on Database systems, vol.17,N°.4, pp.689-717, Dec.1992;
[37] E.W. Dijsktra 'Self stabilizing systems in spite of distributed control'. Communication of the ACM, Novembre 1974
[38] D. M. Dhamdhere, S. S. Kulkarni'A token based k-resilient mutual exclusion algorithm for distributed systems' Indian institute of technology.
[39] A. Bouabdallah,J.C. Konig 'A distributed algorithm for mutual exclusion' French project C^3 of CNRS .

L'exclusion mutuelle dans les réseau mobiles ad hoc
[40] A.Derhab 'Algorithme d'exclusion mutuelle pour un environnement mobile ad hoc'. Thèse de magister, USTHB 2003 ;
[41] J.Walter, S.Kini 'Mutual Exclusion on Multihop, Mobile Wireless Networks', Texas A&M University, College Station, December 9, 1997.
[42] Y.Chen and J.Welsh. 'Self stabilizing mutual exclusion using tokens in ad hoc networks'. Technical report 2002-4-2; Departement of computer science, Texas A&M univ. Avril 2002;
[43] J.Walter, J.Welsh and N.Vaidya. 'A mutual exclusion for ad hoc mobile networks'. Departement of computer science, Texas A&M University, college station, USA, 2001;
[44] R.Baldoni and A.Virgilito. 'A token based mutual exclusion algorithm for mobile ad hoc networks'. Technical report 28-01;
[45] N.Malpani, N.Vaidya and J.Welsh. 'Distributed token circulation on mobile ad hoc networks', Technical report, Intel corporation 505 Etna land Dr.Suite 550, Austin TX 78752;

La K-exclusion mutuelle distribuée
[50] K. Raymond. 'A distributed algorithm for multiple entries to a critical section'. Information Processing Letters, N° 30, fevrier 1989;
[51] Srimani and Reddy. 'Another distributed algorithm for multiple entries to a critical section'. Information processing letters; Janvier 1992;
[52] Kakugawa, Fujita, Yamashita and AE. 'A distributed K-mutual exclusion algorithm using K-coterie'. Information Processing Letters, vol.49, pp.213-2188, Mars 1994;
[53] N.Neilsen and M.Mizuno. 'Nondominated K-coterie for multiple mutual exclusion'. Information proceedings, 1994;
[54] YenChang and Bor-Hsu Chen. 'An extended Binary tree Quorum strategy for K-mutual exclusion in distributed systems'. Proceedings of the 1997 Pacific Rim International Symposium on Fault-Tolerant Systems; 1997;
[55] J.Jiang, S.Huang and Y.Kuo. 'Cohorts Structures for Fault Tolerant K entries to a critical section'. IEEE Transactions on computers; Fevrier 1997;
[56] Agrawal and a.El abbadi. 'Analysis of Quorum-Based Protocols for Distributed (K+1)-mutual Exclusion'.IEEE Transactions on parallel and distributed systems. 1997;
[57] Kakugawa, Fujita, Yamashita and AE. 'Availability of K-coterie'. IEEE transactions on computers; mai 1993;
[58] IV. S.Bulgannaxar&N.H.Vaidya 'A distributed K-mutual exclusion Algorithm' Department of computer science Texas A&M university, Technical Report 1994;

[59] V. M.Naimi 'Distributed Algorithm for K-entries to Critical Section on the Directed Graphs' Laboratoire informatique Besancon Franc;

La K-exclusion mutuelle dans les réseau mobiles ad hoc
[60] J.walter, G.Cao and Mohanty. 'A K-mutual exclusion algorithm for wireless ad hoc networks'. Departement of coputer science, Texas A&M University, college station, USA; 1998.
[61] Jennifer E. Walter Guangtong Cao Mitrabhanu 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;

Généralités
[70] Tony Larson, Nicholas Hedman 'Routing protocol in wireless ad-hoc networks- A simulation study' Master thesis LULEA TERNISKA university 1998;
[71] Elaine RICH 'Intelligence artificielle' traduit de l'anglais au Français par David GUEDJ Edition MASSON 1987;
[72] Ronald L.Graham, Donald E.Knuth et Oren Patashnik 'Mathématiques concrètes' International Thomson publishing 2ième édition 1997.