ÉCOLE DOCTORALE SCIENCES ET INGÉNIERIE
UNIVERSITÉ DE CERGY PONTOISE

THÈSE

présentée pour obtenir le grade de

DOCTEUR EN SCIENCES
DE L'UNIVERSITÉ DE CERGY PONTOISE

SPÉCIALITÉ : INFORMATIQUE
OPTION : SYSTÈMES DISTRIBUÉS ET RÉSEAUX

TITRE

EXCLUSION MUTUELLE DE GROUPE DANS LES SYSTÈMES DISTRIBUÉS
GROUP MUTUAL EXCLUSION IN DISTRIBUTED SYSTEMS

PAR

Ousmane THIARE

Laboratoire Informatique de Cergy-Pontoise (LICP) EA 2175

Soutenue le 25 Juin 2007

Devant le jury composé de:

M.	Christian	LAVAULT	Professeur	Univ. Paris 13	Président
M.	Jean Frédéric	MYOUPO	Professeur	Univ. Amiens	Rapporteur
M.	Abdelmadjid	BOUABDALLAH	Professeur	UTC Compiègne	Rapporteur
M.	Mohamed	NAIMI	Professeur	Univ. Cergy	Dir. de thèse
M.	Mary Teuw	NIANE	Professeur	UGB St-Louis	Examinateur
M.	Mourad	GUEROUI	Maître de conférences	Univ. Versailles	Examinateur
M.	Laurent	AMANTON	Maître de conférences	Univ. du Havre	Examinateur

RÉSUMÉ

L'exclusion mutuelle de groupe est une généralisation intéressante du problème de l'exclusion mutuelle. Ce problème a été présenté par Joung, et quelques algorithmes pour le problème ont été proposés en incorporant des algorithmes d'exclusion mutuelle. L'exclusion mutuelle de groupe se produit naturellement dans une situation où une ressource peut être partagée par des processus du même groupe, mais pas par des processus de différents groupes. Il est aussi appelé « problème des philosophes parlant d'une même voix ».
Un exemple d'application intéressant est un serveur de CD (documentation, films...) sur Internet : plusieurs utilisateurs désirant accéder au même CD pourront le faire en même temps au lieu d'attendre la fin de la requête des autres utilisateurs. Des solutions efficaces, écrites dans le modèle à passage de messages et basées sur les quorums d'une part et sur la circulation de jeton d'autre part sont présentées dans cette thèse.
Nous abordons aussi le problème de l'exclusion mutuelle de groupe sur les réseaux mobiles ad hoc et nous proposons un algorithme qui est sensible aux formations et coupures de liens et est ainsi approprié pour les réseaux mobiles ad hoc.

Mots-clés : Exclusion mututelle, Exclusion mutuelle de groupe, Quorum, Réseaux ad hoc.

ABSTRACT

Group mutual exclusion is an interesting generalization of the mutual exclusion problem. This problem was introduced by Joung, and some algorithms for the problem have been proposed by incorporating mutual exclusion algorithms. Group mutual exclusion occurs naturally in a situation where a resource can be shared by processes of the same group, but not by processes of different groups. It is also called the « congenial talking philosophers problem ».

An example of application interesting is a CD jukebox (documentation, films...) on Internet : several users wishing to reach same CD will be able to do it at the same time instead of awaiting the end of the request of the other users. Effective solutions, written in the message passing model and based on the quorums on the one hand and the circulation of token on the other hand are presented in this thesis.

We tackle also the problem of the mutual exclusion of group on the ad hoc mobile networks and we have proposed an algorithm which is sensitive to link formation and link breaking and thus which is suitable for ad hoc mobile networks.

Keywords : Mutual exclusion, Group mutual exclusion, Quorum, Ad Hoc Networks.

REMERCIEMENTS

Je tiens à remercier ici l'ensemble des personnes, qui par leur conseils, leurs remarques et leurs encouragements, ont contribué à l'aboutissement de ce travail :

Christian LAVAULT, Professeur à l'Université de Paris 13, pour m'avoir fait l'honneur de présider le jury de cette thèse.
Jean-Frederic MYOUPO, Professeur à l'Université de Picardie Jules Vernes, et Abdelmadjid BOUABDALLAH, Professeur à l'Université Technologique de Compiègne, qui ont accepté de consacrer une partie de leur temps pour juger ce travail.
Mohamed NAIMI, Professeur à l'Université de Cergy-Pontoise, mon directeur de thèse, pour m'avoir fait confiance depuis le début et pour les innombrables discussions que nous avons eues. Sa rigueur intellectuelle et morale est pour moi plus qu'un exemple à suivre, un modèle.
Laurent AMANTON, Maître de Conférences à l'Université du Havre, qui m'a fait l'honneur d'accepter de faire partie du jury.
Mourad GUEROUI, Maître de Conférences à l'Université de Versailles Saint-Quentin en Yvelines, pour avoir accepté de participer à ce jury, et pour m'avoir permis de bénéficier des ressources du laboratoire PriSM.
Mary Teuw NIANE, Professeur à l'Université Gaston Berger de Saint-Louis du sénégal, pour m'avoir fait l'honneur d'accepter de faire partie du jury.

L'ensemble des membres du laboratoire L.I.C.P de l'Université de Cergy-Pontoise et plus particulièrement Mohamed Djoudi (ARA), Omar Moussaoui et Adlen Ksentini, pour les nombreuses discussions fructueuses que nous avons eues.
Sebastien CANTARELL, à la direction générale de l'armement, pour ses encouragements. Son aide pour la rédaction de la thèse m'a été précieux.
Je ne peux terminer ces remerciements sans parler de mon Cher Professeur Abdou SENE qui

m’a initié à la recherche en acceptant de m’encadrer en DEA de Mathématiques Appliquées à l’Université Gaston Berger de Saint-Louis.
Mes remerciements vont aussi à tous mes collègues enseignants de l’UFR SAT de l’Université Gaston Berger de Saint-Louis.
Je tiens particulièrement à remercier mes amis : Assane Racine M’baye, Armand Kouedou (qui est présentement aux USA et qui aurait bien voulu être présent à mes côtés le jour de la soutenance), Nadia, AKD, Djibril Ndome à Génève, Diby Fall aux USA, Cdt et Cdte Thiam au Canada, Nadir Matringe, Mounia, Sakina, Jaya sans oublier personne.

À ma famille et à ma femme

TABLE DES MATIÈRES

TABLE DES FIGURES

LISTE DES TABLEAUX

LISTE DES ALGORITHMES

INTRODUCTION

L'apparition des réseaux d'ordinateurs ou de processeurs inter-communiquants a contribué à l'émergence de l'informatique dite répartie. La proximité physique des ordinateurs, le fait que ce soit dans la plupart des cas des micro-ordinateurs, qui ne peuvent parfois supporter seuls l'exécution d'une tâche, a conduit à considérer l'ensemble du réseau, ordinateurs et lignes de communications comme formant une entité unique, un système réparti.

Les besoins potentiels dans certains domaines sont si gourmands au regard des progrès attendus des technologies monoprocesseurs que l'idée s'impose d'en combiner un certain nombre afin d'ajouter une dimension au facteur de croissance. En effet, dans de nombreux domaines scientifiques, mécanique des fluides, triatement d'images, intelligence artificielle, etc., les besoins de calculs sont très importants. La résolution des problèmes, souvent très complexes, relevant de tels domaines nécessite l'utilisation de machines parallèles qui ont répondu partiellement à ces besoins de calcul. L'intérêt effectif du traitement à grande puissance étant admis maintenant, deux aspects critiques sont à maîtriser : leur coût d'une part, la puissance de pointe de l'autre.

Il existe déjà un très grand nombre de machines parallèles ayant plusieurs processeurs à usage général traitant chacun une partie du problème à résoudre. La classification de ces machines peut être faite suivant plusieurs critères. La taxonomie de M.J. Flynn [Fly96], classifie les architectures en fonction de la présence d'un ou plusieurs flux de contrôle (*instruction stream*) ou de données (*data stream*). Cependant, suivant la répartition des données et le mode de contrôle des processeurs, on peut distinguer grossièrement deux types de machines parallèles qui sont venus compléter la technologie réalisée sur un seul processeur appelé aussi SISD (*Single Instruction Single Data*).

Dans la catégorie SIMD (*Single Instruction Multiple Data*), les exemples les plus connus

de machines commercialisées sont : CM-1 et CM-2 de la «Connection Machine» [Hil85].

La catégorie MIMD (*Multiple Instruction Multiple Data*) utilise des flots d'instructions et plusieurs flots de données. Cette architecture est la plus générale, puisqu'elle permet le traitement parallèle d'une application et le multitraitement de plusieurs applications simultanément. De nombreuses machines sont de types MIMD. La caractéristique principale de cette architecture repose sur la façon dont l'information est partagée. Mémoire partagée ou mémoire distribuée.

- Les machines à mémoire partagée représentent l'architecture la plus fortement couplée. La mémoire partagée permet d'avoir un état global instantané, cohérent et accessible à tous les processus d'une application. Parmi les problèmes posés par le partage de la mémoire, on peut citer le conflit d'accès aux données, l'ordonnacement des calculs, la parallélisation des programmes etc. Les problèmes posés par ce type d'architecture sont résolus partiellement par l'algorithmique parallèle.
- Les machines à mémoire distribuée, sont caractérisées par l'absence de mémoire partagée, avec ou sans horloge globale. La *T_Node* est une machine à mémoire distribuée synchrone (avec horloge globale), alors que l'*hypercube* est une machine à mémoire distribuée asynchrone (sans horloge globale).
  Parmi les problèmes posés par l'absence de mémoire partagée et d'horloge globale, on peut citer l'exclusion mutuelle, l'exclusion mutuelle de groupe qui est plus récente, la détection de terminaison d'un calcul distribué, l'ordonnancement des événements, la cohérence d'une base de donnée répartie, la tolérance aux fautes etc. Les problèmes posés par ce type d'architecture sont résolus par l'algorithmique distribuée.

L'apparition récente de machines parallèles et surtout leur grande diversité a contraint les chercheurs à reconsidérer l'approche algorithmique classique. L'idée de base du calcul parallèle consiste à faire exécuter simultanément le plus grand nombre d'instructions en faisant collaborer, au sein d'une même architecture, plusieurs processeurs pour résoudre un problème donné. Dans un algorithme, certaines parties peuvent s'exécuter de façon indépendante, il est donc nécessaire de les identifier pour envisager par la suite, leur exécution simultanée sur des processeurs différents afin de réduire le temps d'exécution de l'agorithme. Intuitivement, l'exécution d'un algorithme sur une machine parallèle est beaucoup plus rapide que son exécution sur une machine séquentielle. Cependant, l'exécution en parallèle d'un algorithme nécessite la résolution de nombreux nouveaux problèmes de synchronisation et de communication.

Une application du calcul parallèle, plutôt distribuée, est l'acquisition et le contrôle d'informations dans un système géographiquement distribué. Par exemple, un réseau de communication de données dans lequel certaines fonctions réseau (comme le routage des messages)

sont contrôlées de façon distribuée.

Plus formellement, un système réparti est défini par la donnée d'un ensemble de processeurs et d'un système de communication sous jacent permettant aux processeurs de communiquer entre eux par échange de messages. Une application répartie est composée d'un ensemble de processus (ou tâches) s'exécutant sur les différents processeurs. Les processeurs n'ont pas d'horloge globale et ne partagent pas de mémoire (chacun a sa mémoire locale). La communication inter-processus repose sur un réseau de transport fiable, cela suppose l'existence d'un ensemble de protocoles qui permet d'assurer cet échange.

Le choix d'un tel système est justifié par le fait qu'il permet notamment :

- la communication des applications existantes, tout en respectant l'autonomie locale de chaque application.
- l'adaptation de la structure d'un système à celle des applications qu'il traite ; de nombreuses applications sont intrinsèquement réparties, soit géographiquement, soit fonctionnellement.
- l'offre de services interactifs à un coût raisonnable ; compte tenu de l'évolution technique récente, un système fournissant une puissance donnée avec une interactivité suffisante est réalisée de manière plus économique par un réseau de machines que par une machine unique centralisée.
- la réduction du temps d'exécution d'une application par l'exécution parallèle de plusieurs programmes sur des processeurs différents.
- le partage des ressources (matériels, programmes, données) entre un ensemble d'utilisateurs. Ce partage peut être réalisé de manière statique (accès d'un certain nombre de clients répartis à des serveurs) ou dynamique (répartition d'une charge globale entre un ensemble de machines).
- une bonne prédisposition pour l'évolution progressive, grâce à la modularité d'un système constitué d'éléments pouvant être remplacés indépendamment ; cette organisation facilite l'extension incrémentale, les modifications locales, le remplacement de sous systèmes.
- une sûreté de fonctionnement accrue par l'isolation physique des différentes parties du système.

Pour résoudre des problèmes dans de tels systèmes, et pour permettre aux différents processus de coopérer, une algorithmique nouvelle basée sur l'échange de messages est née. Son utilisation est indispensable dès qu'il s'agit de concevoir la solution à un problème donné en terme d'agents qui n'ont chacun qu'une connaissance partielle des divers paramètres du problème à résoudre.

Les algorithmes de cette forme sont appelés protocoles dans le domaine des réseaux : ils réalisent des services de transfert d'informations ou de contrôle d'activités dans un ensemble de processus réalisant une application, s'exécutant sur des sites distincts et reliés par des liens de communications. Dans le domaine des systèmes informatiques répartis, ces algorithmes sont appelés algorithmes distribués. Ils réalisent généralement les fonctions de base inhérentes à tout système informatique (exclusion mutuelle par exemple) et des fonctions de contrôle inhérentes à la distribution (détection de terminaison d'un programme réparti ou gestion des copies dupliquées d'un même ensemble de données par exemple). Indépendamment des réseaux et des systèmes, les algorithmes exprimés traditionnellement en séquentiel peuvent être reformulés en terme de processus communiquant par messages. le domaine de l'algorithmique distribuée est vaste, et indépendamment du support sur lequel ils s'exécuteront et de leur finalité propre, M. Raynal, dans [Ray85a], les caractérise par l'équation suivante : *algorithme distribué= processus + messages*.

Dans les systèmes distribués, le contrôle, le calcul et les données sont distribués sur différents noeuds du système. Il devient important et nécessaire que le système soit fiable pour garantir l'échange de données sans erreur et sans perte. Ces erreurs peuvent se produire à cause de la défaillance d'un ou de plusieurs noeuds du système.

Les différents processus du système sont soit en coopération, c'est-à-dire qu'ils échangent des messages, soit en compétition d'accès à une ressource partagée. L'accès simultané par plusieurs processus à une ressource peut provoquer une situation incohérente. Pour cela, les différents processus doivent être synchronisés pour accéder à une ressource. C'est le problème de l'exclusion mutuelle dans un système distribué.

Le problème de l'exclusion mutuelle dans un système centralisé est résolu par plusieurs méthodes telles que les sémaphores. L'allocation des ressources est gérée par un allocateur qui dispose d'un état global cohérent du système.

Les premières solutions au problème de l'exclusion mutuelle ont été proposées par Dekker [Dij65] et Lamport [Lam74]. Elles reposent sur des hypothèses d'atomicité ou de non-atomicité des opérations de lecture et d'écriture. Ainsi, une multitude d'algorithmes répartis garantissant l'exclusion mutuelle dans le contexte réparti sont apparus [Ray92b, Ray92a]. Plusieurs nouvelles techniques et théories ont été développées pour prouver les bonnes performances de ces algorithmes. ce problème a même fait l'objet d'études particulières pour différentes topologies de réseau de communication.

Dans le contexte des systèmes répartis où les échanges de messages ont remédié à l'absence de mémoire globale, les premières solutions au problème de l'exclusion mutuelle sont

dues à Le Lann [LeL77] et à Lamport [Lam78b].

Le présent travail est une contribution à l'étude de l'exclusion mutuelle de groupe qui est une généralisation de l'exclusion mutuelle. Cette fois, le système dispose de plusieurs ressources partagées. Lorsqu'une ou plusieurs machines désirent accéder à une même ressource, elles peuvent y accéder directement si la ressource demandée est celle qui est utilisée actuellement, ou si aucune autre machine n'a demandé d'autres ressources. L'exclusion mutuelle de groupe est un problème naturel, formulé pour la première fois par Joung [Jou98], qui généralise le problème de l'exclusion mutuelle classique. Dans le problème de l'exclusion mutuelle de groupe, un processus demande une « session ou ressource » avant d'entrer en section critique. Les processus peuvent être simultanément en section critique seulement s'ils ont demandé la même ressource. Aussi il faut remarquer que si un ou plusieurs processus accèdent à une même ressource R_i, alors aucun autre processus n'accède à une ressource différente $R_j, (i \neq j)$. Nous avons proposé des solutions basées sur les quorums d'une part et sur la circulation de jeton d'autre part dans cette thèse.

Dans un premier temps, nous nous sommes intéressés à l'exclusion mutuelle de groupe basée sur les quorums. Ce problème appartient à la classe des algorithmes répartis fondés sur les permissions. Les quorums sont couramment utilisés dans l'exclusion mutuelle classique. Dans le cas de l'exclusion mutuelle de groupe, un processus voulant accéder à la section critique envoie une requête à tous les processus dans un quorum. Un processus entre en section critique après avoir obtenu les permissions de tous les processus appartenant à son groupe, et libère les permissions au moment de quitter la section critique. Deux algorithmes pour le problème de l'exclusion mutuelle de groupe ont été proposés dans ce sens. Dans le premier algorithme, nous avons proposé un algorithme d'exclusion mutuelle de groupe basé sur un système ordinaire de quorums contrairement à Joung [Jou01b] qui a utilisé la définition de systèmes de *m*-groupes quorum définis dans [Jou01b], où *m* est le nombre de ressources dans le réseau. La construction du système de quorum utilisée par Joung [Jou01b] est assez complexe, ce qui nous a amené à considérer un système ordinaire de quorums. Cet algorithme est inspiré de celui de Maekawa [Mae85]. Notre contribution dans ce premier algorithme a été de faire la différence entre les processus et les ressources ou sessions contrairement à Joung [Jou01b]. On a utilisé dans cet algorithme les systèmes ordinaires de quorums en généralisant le problème de l'exclusion mutuelle. Cet algorithme réduit aussi le nombre de messages par entrée en section critique par rapport à l'algorithme Maekawa_M de Joung dans [Jou01b]. Cette généralisation permet aux processus d'acquérir des quorums simultanément, et ainsi leur permet d'entrer en section critique de façon concurrente.

Un autre algorithme d'exclusion mutuelle basé sur les quorums et traitant les accès concurrents aux sessions a aussi été proposé. Dans cet algorithme, nous avons proposé une idée

originale consistant à considérer dans un groupe de processus voulant accéder à une même ressource, un processus jouant un rôle de coordonnateur, qui va se charger d'entrer en section critique seulement avec les membres de son groupe. Il est aussi inspiré de celui de Maekawa [Mae85].

Ces deux algorithmes proposés ont aussi l'avantage de servir les demandes d'accès suivant l'ordre dans lequel elles ont été demandées (propriété *FIFO, First-In-First-Out*).

Dans un deuxième temps, nous avons proposé un algorithme d'exclusion mutuelle basé sur le modèle client-serveur et un autre algorithme pour le problème d'exclusion mutuelle pour les réseaux mobiles ad hoc. Ces deux algorithmes appartiennent à la catégorie des algorithmes distribués fondés sur une circulation de jeton.

Ces deux algorithmes sont fondés sur une circulation de jeton écrits dans le modèle à passage de message. Dans le premier algorithme proposé, c'est-à-dire celui basé sur le modèle client-serveur, nous avons utilisé une topologie en arbre (arborescence). Dans cet algorithme, un processus n'interroge pas les autres processus lorsqu'il veut participer à une session. Le jeton sert juste à ouvrir et fermer les sessions. Avant d'ouvrir une nouvelle session, un processus devra initialiser une phase de fermeture de la session courante, afin de s'assurer qu'aucun processus ne se trouve encore en section critique.

Dans le deuxième algorithme d'exclusion mutuelle de groupe pour les réseaux mobiles ad hoc, nous nous sommes inspirés de l'algorithme *RL* (Reverse Link) présenté par J. Walter et al. dans [WWV98]. L'objet de notre contribution dans cet algorithme a été de proposer d'ajouter quelques variables et messages afin de résoudre le problème de l'*entrée concurrente*. Nous avons prouvé que cet algorithme satisfait les propriétés d'exclusion mutuelle, d'absence de famine et d'entrée concurrente. Cet algorithme est sensible aux formations et coupures de liens et est ainsi approprié pour les réseaux mobiles ad hoc.

Le plan de cette thèse est organisé comme suit :

- Dans le chapitre introductif, nous présentons le contexte général de notre travail.
- Un état de l'art des problèmes d'allocation de ressource dans le cadre des systèmes distribués est présenté dans ce premier chapitre. Nous y présentons aussi l'exclusion mutuelle de groupe ainsi que des utilisations et un état de l'art.
- Dans le deuxième chapitre, nous présentons les définitions formelles des structures de

quorum, incluant les ensembles de quorum, coteries et bicoteries. Aussi, les définitions des quorums non dominés sont données. Ensuite, quelques propriétés qui appartiennent à des travaux précédents sont présentées.

- Dans le troisième chapitre, nous présentons un algorithme d'exclusion mutuelle de groupe basé sur les quorums. Nous présentons aussi dans ce chapitre un algorithme d'exclusion mutuelle de groupe sur les accès concurrents de sessions basés sur les quorums.
- Dans le chapitre quatre, nous présentons un algorithme d'exclusion mutuelle de groupe basé sur le modèle client-serveur.
- Un algorithme d'exclusion mutuelle de groupe pour les réseaux mobiles ad hoc est présenté dans le chapitre cinq.
- La dernière partie de la thèse est consacrée à la conclusion et quelques perspectives liées à cette thèse.

CHAPITRE 1

Exclusion mutuelle et exclusion mutuelle de groupe : état de l'art

DAns ce chapitre, nous présentons un état de l'art de quelques problèmes classiques d'exclusion mutuelle. Le problème de l'exclusion mutuelle aura bientôt 40 ans, et est sûrement l'un des problèmes les plus étudiés dans les systèmes. Nous présentons ici les principaux algorithmes et techniques qui sont apparus depuis 1965. D'autres exemples sont présentés dans [BB90, Ray85b].

Nous présentons aussi dans ce chapitre le problème de l'exclusion mutuelle de groupes en énonçant les spécifications de ce problème.

1.1 le problème de l'exclusion mutuelle

Le problème d'allocation en exclusivité d'une seule ressource (Exclusion Mutuelle) est un paradigme des problèmes de compétition dans les systèmes. Il s'agit d'un problème de compétition dont l'énoncé est très simple : une entité, appelée ressource non partageable ou critique, ne peut être octroyée à un instant donné qu'à un seul processus parmi N processus du système.

Autrement dit, dans un tel problème, c'est l'aspect de concurrence qui domine lorsque plusieurs processus tentent en même temps d'accéder à une même ressource.

En effet, l'utilisation de la ressource critique simultanément par un ensemle de processus peut créer une situation d'incohérence.

D'où la nécessité du développement des algorithmes de contrôle ayant comme fonctions principales : synchroniser les différents processus d'un système distribué pour l'accès à la même ressource, et éviter toute incohérence. Ce sont les algorithmes distribués d'exclusion mutuelle.

1.1.1 Environnement

Nous considérons un système distribué constitué par un ensemble fini de processus $\{P_1, \cdots, P_i, \cdots P_N\}$ qui communiquent exclusivement par messages.

Le canal de communication où transitent les messages est le moyen qui permet aux processus de communiquer entre eux. Tous les algorithmes qui seront examinés dans cette section se basent sur les hypothèses suivantes :

1. Les canaux :

- ne dupliquent pas les messages,
- peuvent être FIFO ou non : ceci signifie que les messages peuvent être reçus ou non dans l'ordre de leur emission,
- délivrent les messages sans perte,
- peuvent être bidirectionnels

2. Le délai :

- de traitement est nul,
- de transmission de messages est arbitraire mais fini.

1.1.2 Spécification des contraintes

Dans un algorithme distribué d'exclusion mutuelle, deux procédures[1] sont généralement offertes à un processus désirant accéder à la ressource critique : une procédure d'**acquisition** et une procédure de **libération**.

Les exécutions de ces procédures doivent être synchronisées de façon à ce qu'à tout moment, un processus et un seul soit en cours d'utilisation de la ressource critique. Généralement, la partie de l'algorithme distribué qui utilise la ressource est appelée **section critique**.

[1] On parle aussi de protocole

Le schéma général d'un algorithme distribué d'exclusion mutuelle est illustré par la figure 1.1.

1.1.3 Invariants du problème

Dans le contexte de l'exclusion mutuelle, nous distinguons deux invariants fondamentaux auxquels est assujetti tout algorithme distribué d'exclusion mutuelle.

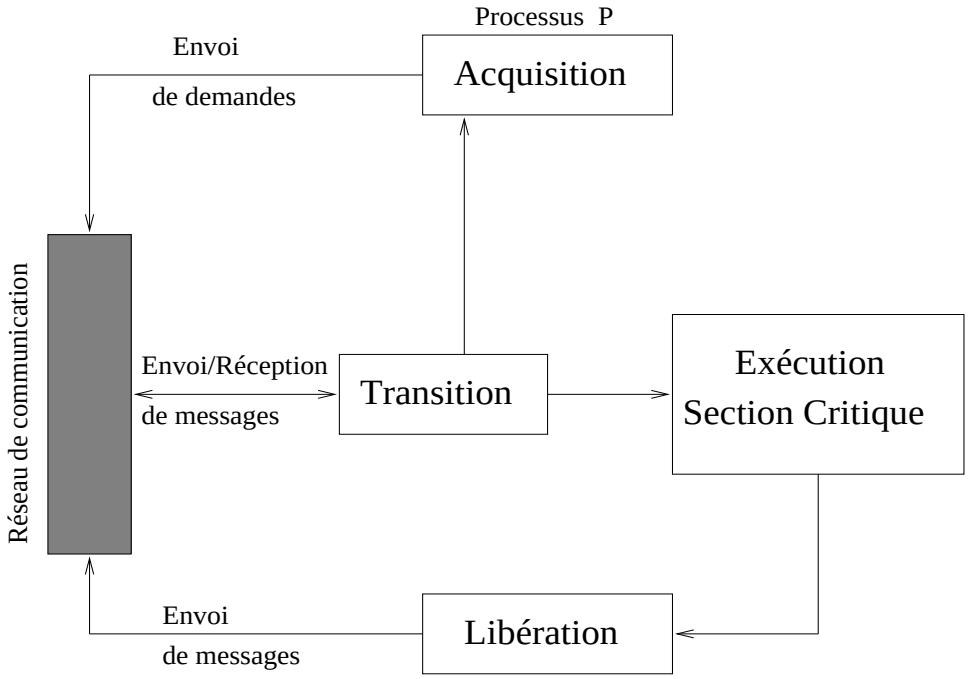

FIG. 1.1 – Structure générale d'un algorithme distribué d'exclusion mutuelle

- **Invariant 1 :** La sûreté[2], à tout moment au plus un seul processus exécute la section critique.
- **Invariant 2 :** La vivacité[3], la section critique est toujours accessible par un processus demandeur au bout d'un temps fini.

Autrement dit, le premier invariant implique que l'exclusion mutuelle à une seule ressource est garantie à tout moment. Tandis que le second indique qu'une situation d'interblocage ne se présente jamais.

Généralement, 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 : ce qui

[2]Ou safety en anglais
[3]Ou liveness en anglais

implique l'absence d'une situation de famine.

1.2 Classification des algorithmes

Notre étude approfondie des algorithmes d'exclusion mutuelle existants a donné naissance à une classification très synthétique sous forme d'approches. La notion d'approche est intimement liée aux outils et mécanismes utilisés dans l'ensemble des algorithmes qui y sont exhibés [KHI98].

Nous nous sommes interessés aux premiers algorithmes proposés et nous avons étudié leur évolution progressive ; ce qui nous a aidé à établir une classification hiérarchique que nous illustrons par la figure ci-dessous.

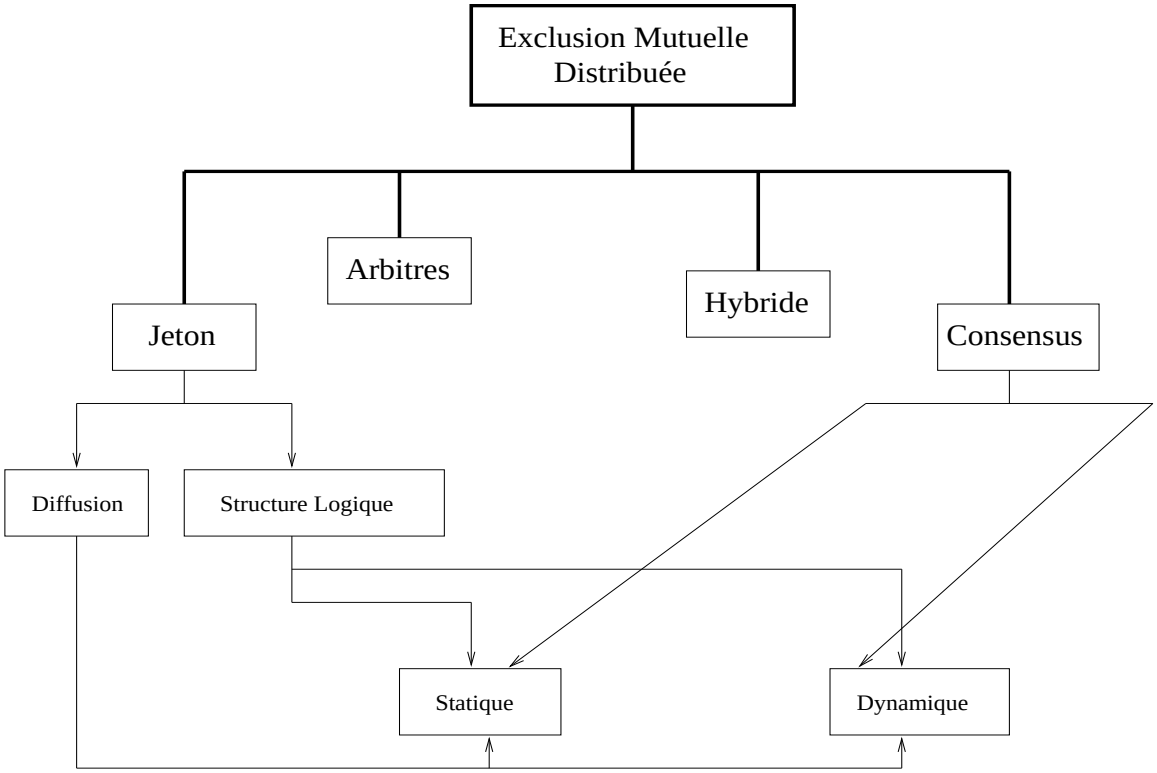

FIG. 1.2 – Classification des algorithmes de réalisation de l'exclusion mutuelle distribuée

1.3 approche basée sur le principe du consensus

Cette première approche est fondée sur la notion de permission qui consiste à accorder à un processus demandeur une réponse favorable d'accès à la section critique.

Un processus demandeur désirant entrer en section critique doit demander un ensemble de permissions à d'autres processus. Autrement dit, un rpocessus demandeur ne pourra accéder à la ressource critique que s'il a reçu toutes les permissions attendues.

On se donne Rp_i, un ensemble de processus auxquels chaque processus P_i demande ses permissions. Selon l'état de cet ensemble, les algorithmes d'exclusion mutuelle de cette approche peuvent être subdivisés en deux catégories : les algorithmes statiques et les algorithmes dynamiques (figure 1.3).

- **Les algorithmes statiques.** Dans cette première catégorie, l'ensemble Rp_i est défini avant toute exécution de l'algorithme, et il demeure inchangé tout au long du déroulement de l'algorithme.
- **Les algorithmes dynamiques.** Dans cette seconde catégorie, en revanche, l'ensemble Rp_i peut évoluer dans le temps. Ceci signifie qu'il ne représente que les processus détenant les permissions manquantes au processus P_i.

Outre la notion de permission décrite ci-dessus, l'absence d'horloge globale dans un système réparti crée un problème crucial : c'est l'ordonnancement des événements entre processus (ici les demandes d'accès à la ressource critique) du système, qui n'évoluent généralement pas indépendamment.

Dans un contexte distribué, la difficulté de mise en oeuvre des outils matériel (assurant une parfaite synchronisation) a conduit certains auteurs à proposer des techniques logicielles telles que les horloges logiques que nous allons examiner maintenant.

Horloges Logiques et estampilles (time stamps)

La méthode de l'horloge logique ou de l'estampille est due à [Lam78b]. Chaque processus dispose d'une horloge logique H qui est incrémentée d'une unité chaque fois qu'un événement se produit dans le processus (événement signifiant émission ou réception d'un message).

Lorsqu'un processus doit émettre un message à destination d'un autre processus, il affecte à ce message la valeur courante de son horloge logique. Cette valeur est appelée estampille E.

Lorsque le messaqge parvient au récepteur, l'horloge de ce dernier devrait, en bonne logique, avoir une valeur supérieure à la valeur E de l'estampille du message reçu, puisque la date de réception est nécessairement postérieure à la date d'émission.

En fait, rien ne garantit une telle synchronisation des horloges ; aussi, si l'horloge du récepteur est inférieure à l'estampille du message reçu, elle subit une mise à jour en prenant pour valuer E+1. Le message reçu est alors affecté de l'estampille E+1. Ainsi, grâce à ce mécanisme de datation, la chronologie entre événement émission et événement réception est assurée.

En outre, lorsque l'on est en présence de deux événements indépendants, rien n'empêche de leur donner la même estampille. Pour lever l'ambiguîté dans ce cas, il suffit de numéroter les processus (leurs identités) et de convenir qu'à estampilles égales, le premier événement est celui qui correspond à la plus petite identité du processus (deux événements issus d'un même processus ne peuvent avoir la même estampille).

Dans la suite, nous décrirons deux algorithmes d'exclusion mutuelle, dont l'un est statique, et l'autre dynamique. ces algorithmes sont à consensus total ou permissions individuelles : cela signifie que chaque processus peut donner sa permission à plusieurs autres processus simultanément. Il gère donc individuellement chacun des conflits qui le met en compétition avec les autres processus. L'accès à la ressource critique est donc autorisé par tous les processus du système.

Les messages utilisés dans ces algorithmes sont :

- **Requête.** Ce type de message exprime le souhait d'entrée en section critique par un processus.
- **Libération.** Il annonce la fin d'utilisation de la section critique.
- **Acquittement.** Ce message est un accusé de réception d'un message **Requête**.
- **Réponse.** Il remplace les deux messages précédents : **Libération** et **Acquitement**.

1.3.1 Algorithmes statiques. L'algorithme de Lamport

Dans cet algorithme [Lam78b], chaque processus P_i gère une copie de la file d'attente d'entrée en section critique. Cette fille d'attente est un tableau où chaque élément *j* contient le type du dernier message reçu par le processus P_j et l'estampille de ce message. Cet algorithme utilise trois messages : **Requête, Libération** et **Acquittement** (décrits précédemment).

L'algorithme fonctionne ainsi : lorsqu'un processus P_i souhaite accéder à la section critique, il envoie une requête estampillée à l'ensemble de processus Rp_i. A la réception d'un message **Requête** par un processus P_j, celui-ci est mis dans l'entrée *j* de la file d'attente. Toutefois, la réception d'un message **acquittement** est ignorée si l'entrée *j* contient déjà un message **Requête**.

Le processus P_i accède à la section critique lorsque sa requête devient la plus ancienne parmi toutes les autres requêtes. Une fois que le processus P_i sort de la section critique, il diffuse un message **Libération**.

1.3.2 Algorithmes dynamiques. L'algorithme de Carvalho et Roucairol

Comme son type l'indique, cet algorithme [CR83] est dynamique dans le sens où un processus P_i qui désire entrer en section critique, n'envoie un message **Requête** estampillé qu'aux processus qui ont accédé à la section critique depuis sa dernière demande. Le processus P_i considère alors qu'il a encore les permissions des autres processus (bien entendu, ceux-ci ne figurent pas dans l'ensemble Rp_i).

Lorsqu'un processus P_i reçoit un message **Requête** de P_i, il répond par un message **Réponse** s'il n'est pas demandeur de la section critique. Si par contre, le processus P_j a déjà formulé sa demande d'entrée en section critique, il procède alors à un calcul de priorité :

- Si l'estampille de sa dernière requête est plus ancienne par rapport à celle de la requête de P_i, il diffère l'emission du message **Réponse**.
- Dans le cas contraire, il envoie une réponse favorable (sa permission) au processus P_i.

Une caractéristique fondamentale de cet algorithme est que toute permission qui parvient à un processus, reste valable tant qu'il n'y a aucune requête de demande d'accès à la section critique.

Performances

Dans l'algorithme de Lamport, un processus demandeur doit envoyer (N-1) messages **Requête** aux autre processus. Par la suite, il attend (N-1) messages **Réponse** pour accéder à la section critique ; enfin à sa sortie de la section critique, il doit envoyer (N-1) messages **Libération**. Par conséquent, tout accès à la section critique nécessite 3*(N-1) messages.

Quant à l'algorithme de Carvalho et Roucairol, le nombre de messages échangés durant l'exécution de la section critique varie entre 0 et 2*(N-1). Il représente une amélioration de l'algorithme de Ricart et Agrawala [RA81] qui lui-même est une amélioration de l'algorithme de Lamport.

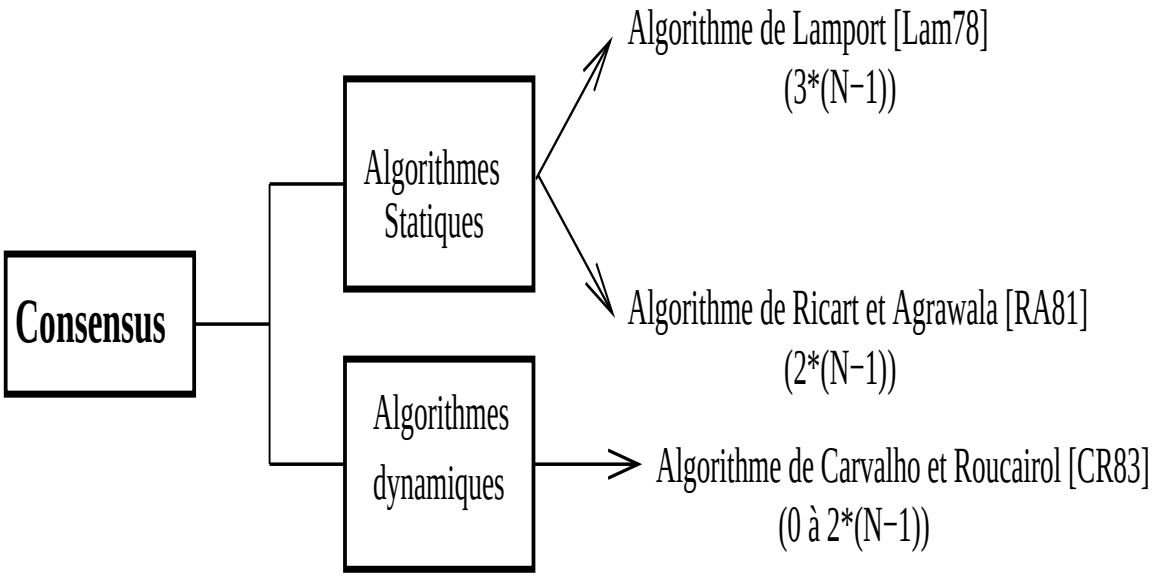

FIG. 1.3 – Les algorithmes basés sur l'approche à consensus

1.4 Approche basée sur des arbitres

Cette seconde approche se base sur une méthode dite des plans projectifs finis, dont nous rappelons brièvement le principe.

Méthode des plans projectifs finis

Comme dans la section 2.3, à chaque processus P_i est associé un ensemble Rp_i. Les ensembles Rp_i $(1 \leq i \leq N)$ à construire sont assujettis aux contraintes suivantes :

- Contrainte C_1. $\forall P_i \, \forall P_j, P_i \neq P_j : Rp_i \cap Rp_j \neq \emptyset$
- Contrainte C_2. $\forall P_i, 1 \leq i \leq N : P_i \in Rp_i$
- Contrainte C_3. $\forall P_i, 1 \leq i \leq N : |Rp_i| = K$
- Contrainte C_4. $\forall P_j, 1 \leq i \leq N : P_j$ appartient à D ensembles $Rp_i, \ 1 \leq i \leq N$

Le problème est donc de trouver N ensemble Rp_i qui vérifient la conditions suivante :

$$N = K(K-1)+1$$

Ceci revient à trouver un plan projectif fini de N points définis par un ensemble de couples (point, ligne) :

- un point représente un processus P_i,
- et une ligne du plan représente un ensemble Rp_i.

Dans [Mae85], l'auteur a montré que le plan projectif à N sommets (processus) vérifie les quatre contraintes précédentes et que $K = D = O(\sqrt{N})$. Il a également prouvé qu'il existe des plans projectifs finis d'ordre *k* lorsque $k = p^m$, où *p* est un nombre entier premier et *m* un entier positif. Un tel plan est constitué de $k(k+1)+1$ points et autant de lignes. En outre, ces plans se caractérisent par les propriétés fondamentales suivantes :

- Chaque point est porté par (k+1) lignes,
- Deux points distincts possèdent une seule ligne en commun.
- Chaque ligne possède (k+1) points,
- Deux lignes distinctes ne se croisent qu'en un seul point.

Dans la figure 1.4, nous illustrons cette méthode pas un exemple simple. Nous supposons que N=7, et de la condition $N = K(K-1)+1$ nous déduisons que K=3.

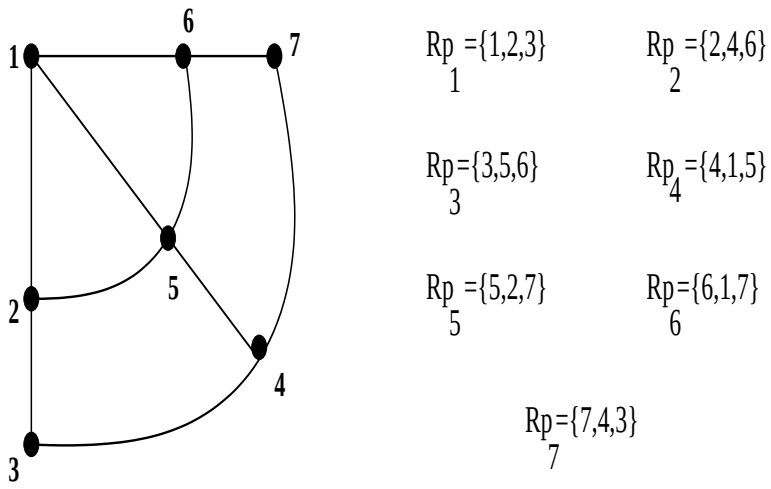

FIG. 1.4 – Un plan projectif fini d'ordre 2

Les algorithmes de cette approche sont dits à consensus partiel ou à permission d'arbitres : ceci signifie qu'un processus ayant reçu plusieurs demandes, n'accorde son autorisation qu'à l'une d'entre elles à un moment donné ; les autres sont mises en attente. Dans

le paragraphe 2.4.1 (à revoir), nous présentons un algorithme à consensus partiel, celui de Maekawa.

Les messages illustrés dans cet algorithme sont :

- **Requête :** exprime le souhait d'entrée en section critique par un processus.
- **Libération :** annonce la fin d'utilisation de la section critique.
- **Autorisation :** exprime une réponse favorable à un message **Requête**.
- **Echec :** exprime une réponse défavorable à un message **Requête**.
- **Interrogation :** indique si un processus a réussi à accéder à la section critique.
- **Relaxe :** exprime une réponse favorable à la question posée via le message **Interrogation**.

1.4.1 L'algorithme de Maekawa

Dans [Mae85], l'autre applique la méthode précédente afin de résoudre le problème de l'exclusion mutuelle distribuée à une seule entrée. Il propose donc un algorithme dont le comportement est le suivant :

Lorsqu'un processus P_i souhaite entrer en section critique, il envoie un message **Requête** à un ensemble de processus Rp_i. A la réception de $|Rp_i|$ messages **Autorisation**, le pocessus P_i accède à la section critique. A sa sortie de la section critique, il envoie un message **Libération** aux processus de l'ensemble Rp_i afin de leur restituer les permissions obtenues.

Comme les ensembles Rp_i vérifient la contrainte C_1, un processus n'attribue qu'une autorisation à la fois. Nous avons donc d'emblée l'exclusion mutuelle garantie.

Lorsqu'un processus P_j reçoit la requête du processus P_i, il lui répond favorablement (par l'envoi d'un message **Autorisation**) s'il n'a pas déjà attribué son autorisation à un autre processus, ou défavorablement (par l'envoie d'un message **Echec**) s'il a déjà donné sa permission à une requête d'un processus P_k plus ancienne que celle du processus P_i.

Lorsque la requête du processus P_k est plus ancienne que celle reçue du processus P_i, le processus P_j adresse au processus P_k un message **Interrogation** afin de savoir s'il est parvenu à accéder à la section critique ou non. A la réception du message **Interrogation**, le processus P_k informe le processus P_i en envoyant un message **Relaxe** s'il n'a pas réussi à entrer en section critique : cela signifie qu'il a reçu déjà un message **Echec**.

Performances

La complexité en messages de l'algorithme de Maekawa a été étudié dans les deux situations suivantes :

- Un processus P_i qui désire entrer en section critique, doit demander les permissions aux processus P_j tels que $\forall\ P_j, P_i \neq P_j\ :\ Rp_i \cap Rp_j \neq \emptyset$; donc, $|Rp_i|$ messages **Requête** sont nécessaires. A sa sortie de la section critique, le processus doit envoyer $|Rp_i|$ permissions et autant de restitutions de permissions. Par conséquent, 3*$|Rp_i|$ messages sont nécessaires pour accéder à la section critique.
- En cas de conflit entre les processus et afin d'éviter une situation d'interblocage, 2*$|Rp_i|$ messages supplémentaires sont nécessaires.

Par conséquent, la complexité en messages est comprise entre 2*$|Rp_i|$ et 5*$|Rp_i|$.

1.5 Approche hybride

L'approche dite hybride, quant à elle, dérive des deux approches précédentes où les processus sont structurés sous forme de groupes ayant des processus en commun [CSL90].

La première approche (basée sur les consensus) est utilisée pour résoudre les conflits entre processus au sein d'un même groupe.
La seconde apprche (basée sur les arbitres) permet de résoudre les conflits entre les processus qui appartiennent à des groupes distincts.

Dans cette optique, les N processus du système sont subdivisés en *g* groupes disjoints $(G_1, \cdots, G_i, \cdots, G_g)$ avec $N = \sum_{i=1}^{i=g} |G_i|$.

Par ailleurs, l'ensemble Rp_i consiste en deux ensembles : un ensemble local noté RpL_i et un ensemble global noté RpG_i. Le choix de tels ensembles doit satisfaire les conditions suivantes :

- Condition 1. $RpL_i \cap RpL_j \neq \emptyset,\ \forall P_i,\ P_j \in G_K$.
- Condition 2. $RpL_i \cap RpL_j = \emptyset, \forall P_i \in G_K, \forall P_j \in G_P$ tel que $K \neq P$
- Condition 3. $RpG_i = RpG_j, \forall P_i, P_j \in G_K$.
- Condition 4. $RpG_i \cap RpG_j \neq \emptyset, \forall P_i \in G_K, \forall P_j \in G_P$ tel que $K \neq P$

La difficulté majeure dans une telle approche réside dans le contrôle des interactions entre les deux niveaux local et global. Pour ce faire, la démarche suivante a été proposée :

Tout processus demandeur P_i effectue les opérations suivantes :

- Envoyer des demandes d'accès locales à tous les processus de l'enesemble RpL_i.
- Attendre jusqu'à ce que la condition $RpL_i = \emptyset$ soit vérifiée.
- Envoyer des demandes d'accès globales à tous les processus de l'ensemble RpG_i. Les processus demandeurs d'un même groupe sont prioritaires pour obtenir une permission globale, par rapport aux processus demandeurs appartenant à d'autres groupes.
- Attendre jusqu'à ce que la condition Nombre_Réponses$_i = |RpG_i|$ soit vérifiée.

Dans la suite, nous décrirons brièvement l'aspect général de l'algorithme d'exclusion mutuelle de Chang et al. [CSL90].

1.5.1 L'algorithme de Chang et al.

Cet algorithme est une combinaison de deux algorithmes d'exclusion mutuelle proposés dans la litterature. Dans [CSL90], l'auteur les a expliqués à deux niveaux d'exclusion mutuelle : un niveau local et un niveau global en faisant appel à la demande décrite précédemment.

- A ce niveau, l'algorithme de Singhal [Sin92] joue le rôle d'un algorithme local. Il utilise les ensembles locaux pour résoudre l'exclusion mutuelle localement.
- En revanche, ici, nous retrouvons l'algorithme de Maekawa [Mae85] (décrit au paragraphe 2.4.1). C'est l'algorithme global qui résout les conflits globalement.

1.6 Approche basée sur la diffusion

D'autres approches destinées à résoudre le problème de l'exclusion mutuelle, sont celles fondées sur le mécanisme du jeton. Ce mécanisme, simple dans son principe et très efficace, est utilisé notamment dans de nombreux problèmes inhérents au domaine des systèmes distribués et à celui des réseaux locaux de communication.

Dans le contexte de l'exclusion mutuelle, le jeton est chargé de contrôler l'accès d'un processus parmi N à une ressource critique. En outre, l'unicité du jeton assure d'emblée l'exclusion mutuelle. Dans la suite, nous proposons trois types d'approches. La première est décrite dans ce paragraphe, les deux autres seront examinés dans les deux paragraphes suivants.

Cette première approche est basée sur le mécanisme de diffusion. Celui-ci est une opération de communication où un processus, appelé source, doit communiquer certaines informations à d'autres processus du système.

Les algorithmes de cette approche ont recours à la diffusion totale, où l'information émise, la requête d'entrée en section critique par un processus demandeur, est diffusée aux (N-1) autres processus, contrairement à la diffusion partielle où l'information concerne certains d'entre eux seulement.

Dans cette approche, l'ensemble Rp_i dépend des quatres états possibles d'un processus :

- «D» : Demandeur de la section critique
- «N» : Non demandeur de la section critique
- «E» : Exécuteur de la section critique
- «P» : Possesseur du jeton et non demandeur

Ainsi, l'état du système est défini par l'état de chacun des processus du système. Selon l'état du système, les algorithmes de cette approche peuvent être soit statiques, soit dynamiques (figure 1.5).

• Dans les **algorithmes statiques**, un processus P_i demandeur diffuse sa requête aux (N-1) autres processus sans se préoccuper de l'état du système.

• En revanche, dans les **algorithmes dynamiques**, un processus P_i adresse sa requête uniquement à un sous-ensemble de l'ensemble Rp_i. Un tel sous-ensemble varie en fonction des processus qui demandent et utilisent le jeton.

Dans la suite nous décrirons deux algorithmes d'exclusion mutuelle dont l'un est statique et l'autre dynamique. Ils utilisent les messages suivants :

- **Requête.** Ce type de message exprime le souhait d'entrée en section critique par un processus.
- **Privilège.** Message qui définit un jeton.

1.6.1 Algorithmes statiques. L'algorithme de Ricart et Agrawala

Dans cet algorithme [RA83], le jeton est à tout instant détenu par un et un seul processus même si ce dernier ne souhaite pas accéder à la section critique.

Le principe de l'algorithme de Ricart et Agrawala se présente ainsi.

Le jeton est demandé par un processus P_i à l'aide d'un message **Requête** estampillé et diffusé à tous les autres processus (le processus P_i ne connait pas le processus qui détient le jeton). Il reste ensuite en attente d'un message **Privilège**. ce dernier mémorise le numéro d'ordre (estampille) de la dernière visite que le jeton a effectué à chacun des processus P_k.

Lorsqu'un processus P_j qui possède le jeton ne désire pas accéder à la section critique, il cherche le premier processus P_k (P_k est choisi dans l'ordre : $P_{j+1}, \cdots, P_N$, $P_1, \cdots, P_{j-1}$) tel que l'estampille de la dernière requête du processus P_k soit supérieure à l'estampille mémorisée par le jeton (message **Privilège**) lors de sa dernière visite au processus P_k.

1.6.2 Algorithmes dynamiques. L'algorithme de Singhal

Contrairement à l'algorithme précédent, un processus qui veut exécuter la section critique doit adresser sa requête de demande uniquement au processus qui détient le jeton oui qui va le posséder dans un futur proche.

Dans [Sin89], l'auteur a défini les tableaux d'états suivants :

1. Deux tableaux SV_i et SN_i qui mémorisent respectivement les états d'un processus P_i (c'est-à-dire «D», «N», «E», «P») et le nombre de demandes faites par le processus P_i.

2. Deux tableaux TSV_i et TSN_i qui mémorisent respectivements les états («D», «N») d'un processus P_i et le nombre de fois qu'un processus P_i a utilisé le jeton. Le message **Privilège** a pour paramètres ces deux tableaux d'états.

Le comportement d'un tel algorithme se déroule selon trois phases :

- **Invocation de la secton critique.** Lors de cette phase, un processus demandeur P_i incrémente de 1 sa composante $SN_i[i]$ et envoie ensuite un message **Requête**$(P_i, SN_i[i])$ uniquement aux processus dont l'état est «D».
- **Libération de la section critique.** Un processus P_j qui sort de la section critique, exécute des règles «de modification». Ces règles consistent à mettre à jour tous les tableaux d'états des processus du système en fonction des informations contenues dans le message **Privilège** (c'est-à-dire TSV et TSN) et celles du processus P_j (c'est-à-dire SV et SN). Cela signifie qu'on prendra en compte seulement les informations récentes : toute donnée obsolète est éliminée.

- **Traitement d'un message requête.** L'auteur a défini également des règles d'«arbitrage», qui permettent de comparer et de mettre à jour les quatre tableaux d'états (c'est-à-dire SV_i, SN_i, TSV_i et TSN_i). Selon le résultat des comparaison, le processus qui a reçu le message **Requête**, décide soit d'envoyer un message **Requête**, soit un message **Privilège** aux autres processus

Performances

La complexité en messages dans les algorithmes statiques est de 0 messages dans le cas où le processus demandeur possède le jeton. Dans le cas contraire, elle est égale à N messages : (N-1) message **Requête** et un message **Privilège**.

En revanche, dans les algorithmes dynamiques le nombre total de messages nécessaires par entrée en section critique varie entre 0 et N.

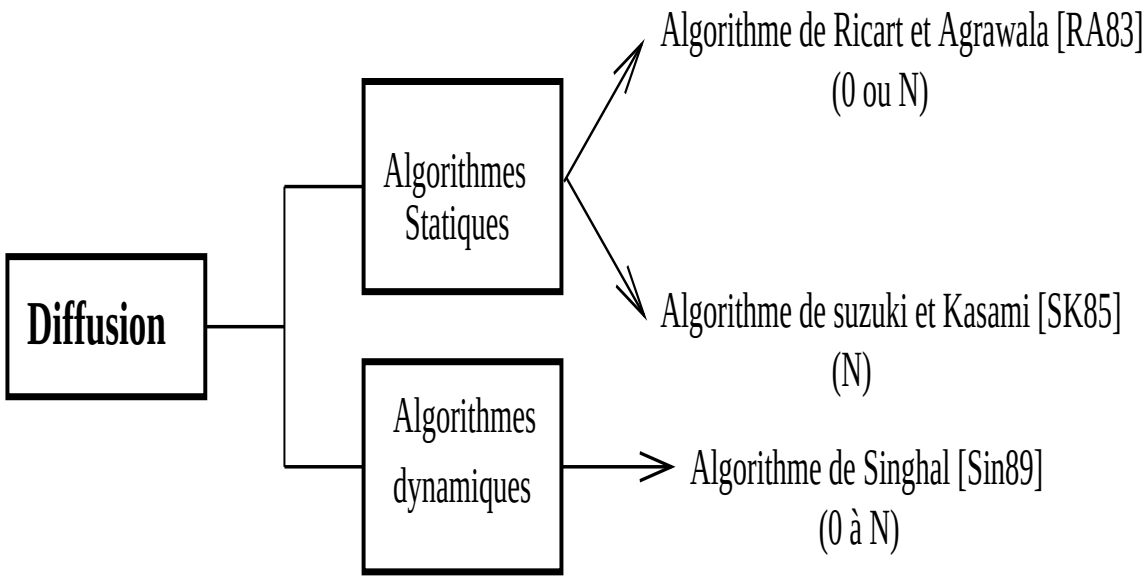

FIG. 1.5 – Les algorithmes basés sur l'approche à diffusion

1.7 Approche basée sur une structure logique statique

Dans les approches précédentes, la conception des algorithmes d'exclusion mutuelle était indépendante de la structure des processus du système. La présente approche possède un aspects structurel : les processus du système sont structurés sous forme d'une configuration logique statique (une topologie de réseau de communication) sur laquelle toute demande est propagée à travers les processus demandeurs et ce jusqu'à atteindre le processus possesseur

du jeton (figure 1.6).

La structure logique statique considérée ici est une topologie en arbre. Cette structure est définie initialement et ne peut évoluer dans le temps. Elle possède les propriétés suivantes :

1. Les arcs sur lesquels transitent les messages échangés entre les processus ne sont pas orientés. Tout processus P_i connaît uniquement ses voisins dans l'arbre, il ignore la structure globale de l'arbre.
2. Un seul processus dans l'arbre possède le jeton. Les requêtes des processus demandeurs doivent se propager séquentiellement à travers l'arbre jusqu'à atteindre le processus possesseur du jeton.

Dans la suite, nous décrirons l'algorithme de Raymond qui utilise deux types de messages **Requête** et **Jeton**.

1.7.1 L'algorithme de Raymond

Dans cet algorithme, les processus sont structurés sous forme d'un arbre : un processus communique uniquement avec ses voisins [Ray89a].
Chaque processus P_i maintient une variable présent$_i$ indiquant l'identité d'un processus voisin dans le chemin qui mène au processus détenteur du jeton (présent$_i = P_i$ implique que le processus P_i possède le jeton). En outre, le processus P_i gère une file d'attente notée Q_i qui consiste à mémoriser les processus demandeurs de la section critique.

L'algorithme de Raymond se comporte ainsi : lorsqu'un processus P_i désire entrer en secton critique, il met sa demande dans la file selon la politique FIFO et envoie ensuite un message **Requête** au processus présent$_i$.

Lorsqu'un processus P_j reçoit un message **Requête** provenant d'un de ses voisins, il rajoute la requête dans sa file Q_j et achemine le même message **Requête** (celui du processus P_i) au processus présent$_j$. A son tour, son voisin procède de la même manière et ce jusqu'à ce que la requête de P_i atteigne le processus qui détient le jeton. Une fois que ce dernier l'a bien reçu, il renvoie un message **Jeton** le long du même chemin, emprunté par la requête de P_i, mais dans le sens inverse.

Lorsqu'un processus P_k reçoit un message **Jeton**, il envoie ce dernier au premier processus de la file Q_k ; ce processus peut être le processus P_k lui-même ou un autre processus demandeur P_j. le processus P_k supprime ensuite de la file Q_k la requête en question. En effet, si la première requête de la file Q_k n'est pas celle du processus P_k et que la file Q_k n'est

pas vide, P_k lui envoie à nouveau un message **Requête** au processus P_j afin que le jeton lui soit retourné.

Performances

Le nombre moyen de messages nécessaires pour l'accès à la section critique est de l'ordre de $\log(N)$.

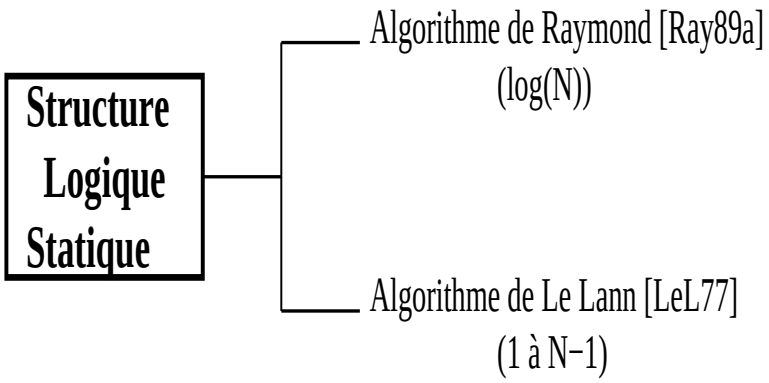

FIG. 1.6 – Les algorithmes basés sur l'approche structure logique statique

1.8 Approche basée sur une structure logique dynamique

Cette approche est basée, quant à elle, sur une structure logique dynamique reconfigurable en fonction des demandes d'entrées en section critique. Contrairement à l'approche précédente, cette approche utilise une structure logique dynamique appelée arborescence. La topologie initiale sous-jacente au système de N processus est un réseau complet.

L'arborescence orientée est construite sur le réseau logique et présente les propriétés suivantes :

1. L'arborescence possède un processus particulier dit racine auquel tous les processus sont reliés directement ou indirectement. Les autres processus sont reliés les uns aux autres selon la relation fils-père.
2. La racine de cette arborescence est l'unique processus qui détient le jeton.
3. Cette arborescence fournit aux différents processus une structure d'adressage facilitant la connaissance de la racine de l'arborescence.
4. Elle peut se reconfigurer d'une part en inversant le sens des arcs entre les processus, d'autre part, en ajoutant certains arcs et en en détruisant d'autres.

Dans la suite, nous décrirons l'algorithme de Naimi et Tréhel. Il utilise les deux messages suivants : **Requête** et **Jeton**.

1.8.1 L'algorithme de Naimi et Tréhel

Dans [TN87a], chaque processus P_i possède deux variables : $dernier_i$ et $suivant_i$.

• La première variable indique le processus auquel P_i doit envoyer un message **Requête**. Initialement, cette variable est à Nil uniquement pour le processus racine, les autres processus ont leur variable qui pointe vers la racine.

• La seconde indique les processus auxquels P_i transmettra un message **Jeton** à sa sortie de la section critique. L'ensemble des variables locales $suivant_i$ de tous les processus constitue une file d'attente distribuée dite file des processus demandeurs du jeton.

En outre, tout processus P_i qui fait une demande d'accès à la section critique devient racine de l'arborescence.

Un processus demandeur P_i ne peut accéder en section critique que s'il a obtenu le jeton. En effet, le message **Requête** du processus P_i est acheminé séquentiellement le long de l'arborescence jusqu'à ce qu'elle arrive à la racine P_r. Tout processus intermédiaire P_j (qui fait suivre la requête de P_i vers son dernier) met à jour sa variable $dernier_i$ à P_i.

Le processus racine P_r peut être soit le processus qui détient le jeton : il transmet le jeton directement au processus P_i ; ou c'est le dernier processus qui recevra le jeton dans un futur proche. Dans ce cas, il met à jour sa variable $suivant_r$ à P_i.

1.8.2 Performances

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

Pour le problème de l'allocation dynamique de ressource, A. Bouabdallah et C. Laforest dans [BL00], ont proposé un algorithme distribué basé sur le jeton. Cet algorithme utilise entre 0 et n+3*k messages (ou *k* est le nombre total de ressources et *n* le nombre total de processus). En moyenne, $O(\log n)$ messages sont utilisés si *k* est constant.

1.9 Généralisation

Même si le problème de l'exclusion mutuelle reste le problème de base, beaucoup de généralisations sont apparues.

1.9.1 Le dîner des philosophes

Le problème du *dîner des philosophes* [Dij78] (noté *DiP*) consiste en *n* philosophes assis autour d'une table ronde garnie à leur intention de cinq assiettes de spaghettis. Malheureusement, il n'y a qu'une seule fourchette entre chaque assiette. Lorsqu'un philosophe a faim, il a besoin de deux fourchettes, il saisit les fourchettes à sa droite et à sa gauche et empêche par la même ses deux voisins de manger. Les philosophes ne peuvent donc pas tous manger en même temps. Un *blocage* peut arriver si chaque philosophe prend la fourchette qui se trouve à sa gauche et attend celle de droite. Le but consiste donc à éviter un tel blocage et permettre à tous les philosophes de manger évitant ainsi une *famine* (au sens propre du terme). Une famine arrive quand un philosophe n'arrive jamais à obtenir deux fourchettes[4].

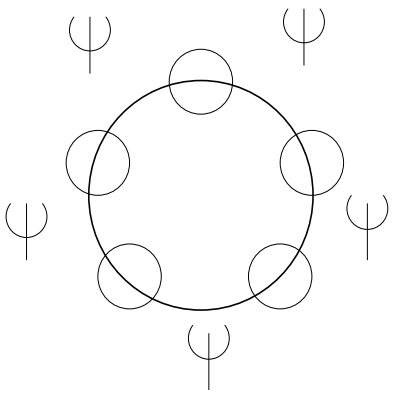

FIG. 1.7 – Le dîner des philosophes

Le dîner des philosophes concerne une topologie en anneau. Lorsqu'un philosophe possède un nombre arbitraire de voisins et qu'il a besoin des fourchettes de tous ses voisins pour manger, ce problème s'appelle le problème du *dîner des philosophe généralisé*, ce problème est aussi plus connu sous le nom du problème d'*exclusion mutuelle locale*. En effet, ce problème étend le problème de l'exclusion mutuelle en modifiant la propriété de sûreté. Celle-ci ne s'applique plus à l'ensemble du système mais seulement à chaque processus et à son voisinage. La ressource n'est plus une notion globale, mais locale.

[4]le problème de famine a été identifié par Dijkstra et fut présenté pour la première fois avec le problème du dîner des philosophes

La propriété de sûreté se transforme simplement de la manière suivante :

- **Surêté** : deux processus **voisins** ne peuvent pas accéder à la section critique (SC) concurremment.
- **Vivacité** : tout processus désirant accéder à la section critique finit par exécuter sa section critique en un temps fini.

1.9.2 Les philosophes qui boivent

Le problème des *philosophes qui boivent*, (noté *DrP*) a été introduit par Chandy et Misra [CM84]. Ce problème est une généralisation du dîner des philosophes. Dans ce problème, les philosophes possèdent un nombre arbitraire de voisins et deux philosophes voisins se partagent une bouteille. Lorsqu'un philosophe a soif, il doit posséder un certain nombre (variable) de bouteilles pour pouvoir faire un cocktail.

Dans le même article, les auteurs proposent une autre généralisation : les philosophes ne se partagent plus une seule bouteille, mais plusieurs. Pour pouvoir différencier les bouteilles, celles-ci sont colorées et un philosophe peut demander plusieurs bouteilles à un même voisin.

- **Surêté** : À tout instant, deux processus voisins p_i et p_j ne peuvent pas utiliser la même bouteille.
- **Vivacité** : tout processus désirant accéder à la section critique finit par exécuter sa section critique en un temps fini.

Ce problème généralise l'exclusion mutuelle locale lorsque chaque processus a besoin de toutes les bouteilles qu'il partage avec ses voisins.

1.9.3 *l*-exclusion

La *l*-exclusion, (notée *l-E*), a été introduit par Fisher, Lynch, Burns et Borodin [FLBB79]. Pour ce problème l'accès à la ressource n'est plus limitée à un seul processus mais au plus à *l* processus, où *l* est un entier ($1 \leq l < n$). Le problème de la *l*-exclusion résout le problème de l'exclusion mutuelle lorsque *l=1*.

- **Surêté** : Au plus *l* processus peuvent accéder à la ressource concurremment.
- **Vivacité** : tout processus désirant accéder à la section critique finit par exécuter sa section critique en un temps fini.

Comme précisé dans [Had02a], il manque une notion d'efficacité à cette spécification. Sans celle-ci, n'importe quel algorithme d'exclusion mutuelle classique vérifie cette spécification. Pour être efficace, un processus ne doit pouvoir être bloqué par un autre processus s'il y a moins de *l* demandes dans le système. Le but d'un algorithme de *l*-exclusion sera donc de permettre à un nombre maximum de processus ($\leq l$) d'accéder à la ressource concurremment.

Comme le problème de l'exclusion mutuelle, les algorithmes résolvant la *l*-exclusion peuvent aussi se classer dans les deux mêmes catégories, ceux basés sur la permission pour autoriser *l* processus, et ceux basés sur la circulation de *l* jetons.

Dans [Ray89a], Raymond propose une modification de l'algorithme de Ricart et Agrawala [RA81]. Pour pouvoir utiliser l'une des *l* ressources, un processus doit recevoir la permission de *n-l* processus. D'autres solutions basées sur les permissions et les quorums ont été proposées dans [NM94, KFYA94]. Dans [SR92], Srimani et Reddy étendent l'algorithme de Suzuki et Kasami [SK85] et permettent à *l* jetons de circuler dans le système. Dans le cadre de l'auto-stabilisation, des solutions auto-stabilisantes pour ce problème sont présentées dans [ADHK01, Had02b]. Dans le cadre de sa thèse, Hadid a proposé d'autres solutions dans [Had02a].

1.9.4 *k* parmi *l*-exclusion

La *k parmi l-exclusion* (notée *k* parmi *l-E*), a été introduit par Raynal [Ray91a]. Ce problème est une généralisation de la *l-E*. Le système contient *l* exemplaires de la ressource et les demandes peuvent se faire pour un nombre *k* d'exemplaires à la fois, avec $1 \leq k \leq l$.

- **Surêté** : il existe *k* exemplaires de la ressource et chaque exemplaire peut être utilisé par au plus un processus à chaque instant.
- **Vivacité** : tout processus désirant accéder à la section critique finit par exécuter sa section critique en un temps fini.

Comme pour le problème de la *l*-exclusion, le but est de permettre un maximum de concurrence. D'autres solutions sont proposées dans [Ray92b, MT99].

1.9.5 Le problème des lecteurs/rédacteurs

Le problème des *lecteurs/rédacteurs* (noté *R/W*) a été introduit par Courtois, Heymans et Parnas [CHP71]. Le problème doit coordonner l'accès à une ressource par deux classes de processus, les lecteurs et rédacteurs. Un processus peut soit lire, soit écrire. Les processus rédacteurs ont besoin d'un accès exclusif à la ressource alors que les processus lecteurs peuvent partager la ressource avec d'autres lecteurs. L'ordonnancement entre les lecteurs et les rédacteurs est un critère important dans [CHP71]. Les auteurs présentent plusieurs solutions en supposant des priorités différentes suivant si l'action est une lecture ou une écriture.

1.10 Le problème de l'exclusion mutuelle de groupe

Dans cette section, nous allons présenter formellement le problème de l'exclusion mutuelle de groupe. Nous présentons aussi les complexités qui permettent d'évaluer les algorithmes de ce type. Ensuite, nous présentons un état de l'art et quelques utilisations de ce problème.

1.10.1 Introduction

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

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

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

1.10.2 Spécifications

Le problème de l'*exclusion mutuelle de groupe* (noté *GME*) traite à la fois du problème d'exclusion mutuelle et du problème des accès concurrents. Une solution à ce problème permet à un nombre quelconque de processus d'accéder à une même ressource *S* (parmi *m*) concurremment. Un algorithme d'*exclusion mutuelle de groupe* doit satisfaire les trois propriétés suivantes :

- **Exclusion mutuelle (Surêté)** : Si deux processus distincts, *p* et *q* exécutent simultanément leur section critique respectivement dans les sessions S_p et S_q, alors $S_p = S_q$.
- **Absence de blocage (Vivacité)** : Si un processus *p* veut accéder à une ressource *S*, alors *p* exécute finalement sa section critique.
- **Entrée concurrente (Efficacité)** : Si des processus veulent accéder à une ressource, et qu'aucun autre processus ne veut accéder à une autre ressource différente, alors les processus peuvent y entrer concurrement.

Dans le reste de cette thèse, nous allons supposer qu'un processus reste en *section critique* durant un temps fini.

La propriété d'*absence de blocage* définie ci-dessus est identique à la propriété d'*attente bornée* définie et utilisée originellement dans [Jou98, Jou01a, WJ99b]. Les deux définitions n'imposent aucune borne sur le temps pour accéder à une session, il suffit juste que le délai soit fini. Dans les spécifications ci dessus, pour soutenir la propriété de « finalement », nous choisissons le terme d'« absence de blocage » pour cette propriété. La propriété d'attente bornée peut être utilisée comme une contrainte plus forte pour le problème de l'exclusion mutuelle de groupe : « il existe une borne sur le nombre de fois que les processus sont autorisés à accéder à une session après qu'un processus ait fait une demande pour accéder à une session et avant que sa requête soit satisfaite ».

Comme pour le problème de la *l-exclusion* (cf paragraphe `1.9.3`), le problème de l'exclusion mutuelle de groupe a besoin d'une propriété d'efficacité pour être correctement défini. Sans la propriété d'entrée concurrente, n'importe quel algorithme d'exclusion mutuelle classique serait une solution au problème. La propriété d'efficacité permet de rejeter les solutions trop triviales ne permettant pas de concurrence.

1.10.3 Complexités

Dans cette section, nous présentons les complexités qui permettent d'évaluer les performances d'une solution à la *GME*.

La mesure des performances d'un algorithme d'exclusion mutuelle de groupe reprend quelques mesures classiques comme la *complexité en nombre de messages, complexité en taille des messages et la complexité en espace.*

La complexité en nombre de messages et en taille mesure respectivement le nombre de messages nécessaires par entrée en section critique et le nombre de bits nécessaires par message.

La complexité en espace mesure le nombre de bits nécessaires par processus.

Dans [Jou98], Joung proposa des mesures spécifiques au problème de l'exclusion mutuelle de groupe, qui sont la *complexité en temps*, la *complexité en changement de contexte* et le *degré (maximum) de concurrence*.

La complexité en temps mesure le nombre de sections critiques exécutées pendant qu'un processus *p* attend l'accès à une session *X*. À cause de la concurrence, les accès peuvent se chevaucher, donc, la complexité en temps ne reflète pas vraiment le temps écoulé.

La complexité en changement de contexte représente le nombre de sessions qui peuvent être ouvertes pendant qu'un processus attend pour accéder à une session différente. La complexité en changement de contexte est une mesure intéressante pour calculer (dans le pire des cas) le « temps d'attente ».

Le degré (maximum) de concurrence compte le nombre de processus qui peuvent concurrement accéder à une session ouverte alors qu'un processus particulier est en train d'exécuter sa section critique et qu'un autre processus attend l'accès à une session différente. Un degré de concurrence élevé signifie une meilleure utilisation des ressources.

Keane et Moir [KM01] proposèrent une nouvelle mesure en rapport avec le temps d'accès à la section critique, appelée la *complexité en l'absence de contention*.

La complexité en l'absence de contention indique, en considérant le pire des cas, le temps nécessaire à un processus pour exécuter sa section critique en l'absence de contention.

Nous avons besoin de la notion de passage [Jou98] pour définir plus formellement les complexités.

Définition 1.1 *(**Passage**) Le passage d'un processus p à travers la session X (noté $\langle p, X \rangle$) est un intervalle $[t_1, t_2]$ (de temps) pendant lequel le processus p exécute sa section critique. Un passage est initié en t_1 et accompli en t_2.*

Soit *q* un processus demandant l'accès à la session *X*. Soit *T* l'ensemble des passages :

1. *initiés* par au moins quelques processus ($\neq$ *q*) après que le processus *q* ait fait sa demande (pour la session *X*),
2. *accomplis* avant que le processus *q* exécute le passage correspondant $\langle q, X \rangle$.

La *couverture minimale C* de *T* et l'ensemble minimal de *T* tel que chaque passage de *T* soit initié et accompli pendant l'exécution de passage de *C*.

Définition 1.2 *(**Complexité en temps**) La **complexité en temps** est le nombre de passages dans C.*

Un *round* (de passages) R_Y de *T* est l'ensemble maximal des passages consécutifs de *T* qui sont des passages à travers la session *Y*.

Définition 1.3 *(**Complexité en changement de contexte**) La **complexité en changement de contexte** est le nombre de rounds de T tel que pour chaque round R_Y, soit $X \neq Y$, soit X=Y mais $\langle q, X \rangle \notin R_Y$.*

Définition 1.4 *(**Degré de concurrence**) Le **degré de concurrence** est défini par le nombre maximum de passages qui peuvent être unitié pendant un round R_Y de T.*

1.10.4 Exemples d'applications

Dans cette section, nous présentons des problèmes et des cas pratiques pouvant être résolus à l'aide d'un algorithme d'exclusion mutuelle de groupes.

Le problème de l'exclusion mutuelle de groupes est une généralisation de l'exclusion mutuelle classique et du problème des lecteurs/rédacteurs. Nous montrons comment résoudre ces deux problèmes en utilisant un algorithme d'exclusion mutuelle de groupe.

Exclusion mutuelle. Avec un algorithme d'exclusion mutuelle de groupes, lorsque les processus sont numérotés de `1` à *n*, il est possible de faire de l'exclusion mutuelle. Pour cela, il faut prendre *m=n*, quand un processus p_i souhaite accéder à la session *S* (en exclusion mutuelle, *S* est unique), il demande la session *i*, ainsi il sera le seul à pouvoir accéder à la section *S*.

Lecteurs/Rédacteurs. Un algorithme d'*exclusion mutuelle de groupe* peut aussi implémenter le problème des lecteurs/rédacteurs. Lorsqu'un processus p_i veut devenir rédacteur, il demande la session *i* (comme nous l'avons vu dans le problème de l'exclusion mutuelle classique). En revanche, lorsque des processus veulent devenir lecteurs, ils demandent une même session spéciale *L*.

Le juke-box. L'utilisation d'un algorithme d'*exclusion mutuelle de groupe* est utile dans le cas où une ressource est partagée par plusieurs processus. Un exemple concret d'utilisation est la gestion d'un *juke-box* dans un réseau. Un seul *CD-ROM* peut être disponible à la fois. En donnant un identifiant unique à chaque *CD-ROM*, un algorithme d'exclusion mutuelle de groupe permet à plusieurs processus de lire un *CD-ROM* concurremment, tout en forçant les processus voulant accéder à un autre *CD-ROM* à attendre.

Pour illustrer cet exemple, on suppose un système où tous les processus veulent accéder soit au *CD-ROM* numéro `1` soit au *CD-ROM* numéro `2` de telle manière que les demandes soient ordonnées telles que `1,2,1,2,1`...

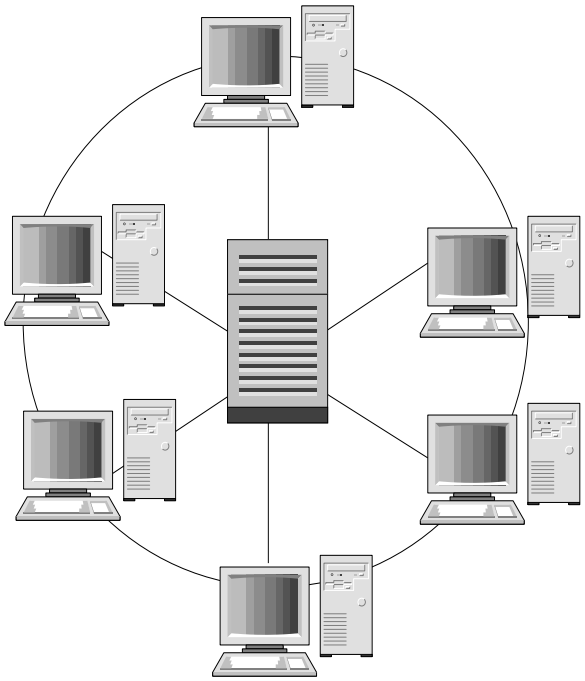

FIG. 1.8 – Le juke-box

L'utilisation d'un algorithme d'exclusion mutuelle classique pourrait entraîner le pire cas (figure 1.9 partie (I)) où les accès se font alternativement au *CD-ROM* `1` et `2`. Et à tout instant, un seul processus est autorisé à lire.

L'utilisation d'un algorithme d'exclusion mutuelle de groupe permet de grouper les requêtes pour un même *CD-ROM* afin d'éviter les changements intempestifs. Dans la figure 1.9 partie (II), les demandes sont groupées et se font concurremment. Les changements de *CD-ROM* sont alors très réduits. Dans notre exemple à la figure 1.9, il n'y a qu'un seul changement au lieu de cinq.

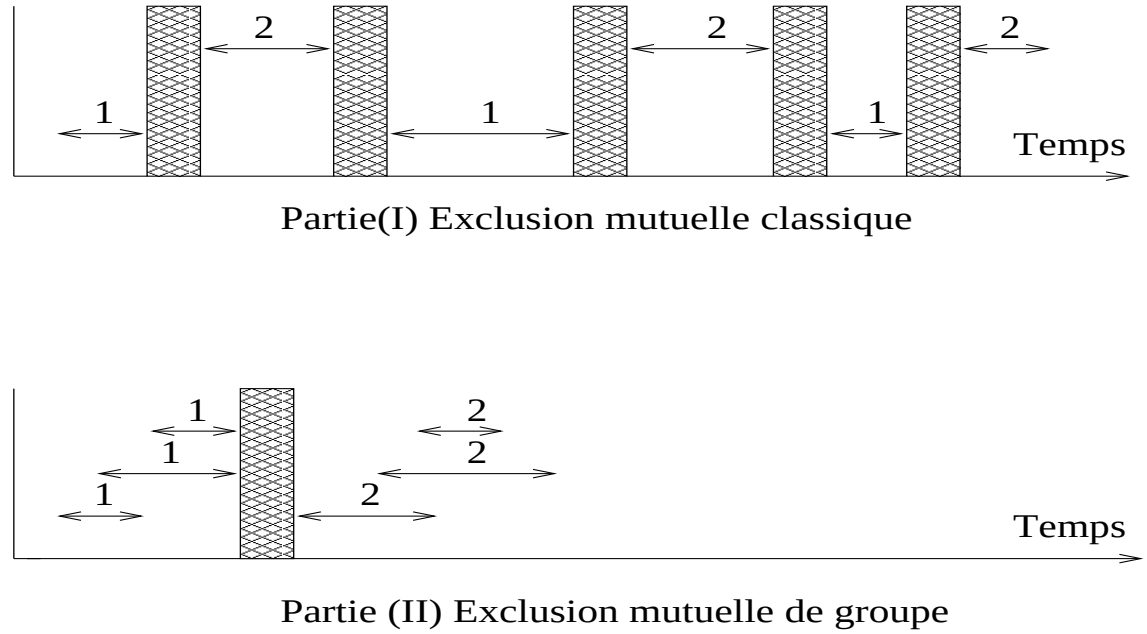

FIG. 1.9 – Comparaison Exclusion Mutuelle/Exclusion Mutuelle de Groupe

Le serveur Internet. Supposons qu'un serveur Internet stocke dans une file d'attente les demandes qu'il reçoit. Dans l'exemple précédent, nous avons vu qu'un algorithme d'*exclusion mutuelle de groupe* permet de grouper les demandes identiques. De la même manière, il est possible de grouper les demandes pour les mêmes adresses Internet.

Par exemple, on suppose que *x* demandes pour un même site *S* sont dans la file d'attente. Un algorithme d'*exclusion mutuelle de groupe* permet de regrouper ces *x* demandes afin d'interroger le site *S* qu'une seule fois, soit un seul message vers *S*. En plus de limiter le nombre de messages vers le site *S* et donc sur le réseau, cette solution réduit aussi les chargements de mémoire.

Maintenant, nous allons présenter quelques algorithmes pour résoudre le problème de l'exclusion mutuelle de groupe.

En 1998, Joung [Jou98] présente pour la première fois le problème de l'*exclusion mutuelle de groupe*. Dans cet article, Joung propose une solution pour le modèle à mémoire partagée.

Dans [Jou01a], Joung présente un algorithme dans le modèle de passage de message sur un graphe complet. Dans [WJ99b], Wu et Joung présentent un algorithme dans le modèle à passage de messages pour une topologie en anneau unidirectionnelle. Dans [KM01], P. Kean et M. Moir présentent eux aussi un algorithme dans le modèle à mémoire partagée avec une approche différente de celle de [Jou98]. Dans [Had01], Hadzilacos discute de la propriété d'entrée concurrente.

1.10.5 Les algorithmes dans le modèle à mémoire partagée

Dans cette section, nous présentons en premier l'algorithme de Joung [Jou98], ensuite, nous présentons les algorithmes de Keane et Moir [KM01] et celui de Hadzilacos [Had01].

L'algorithme de Joung [Jou98] Dans [Jou98], Joung présente les bases pour le problème de l'exclusion mutuelle de groupe. Il présente un algorithme défini pour un système *CSCW*, pour *Computer Supported Cooperative Works*. Le *CSCW* décrit les systèmes qui permettent de travailler en coopération par le biais de l'outil informatique. des exemples d'applications *CSCW* sont : le courrier électronique, la vidéo-conférence, les applications temps réel distribuées (écriture ou dessin en collaboration).

Avant d'arriver à l'algorithme CTP_m qui résout l'exclusion mutuelle de groupe, l'auteur présente plusieurs solutions plus simples : une solution centralisée, une solution répartie, une solution non équitable pour deux sessions et une solution équitable pour deux sessions.

L'algorithme CTP_m (pour *Congenial Talking Philosopher with m forums*) est exécuté par un processus uniquement lorsqu'il souhaite utliser une ressource. L'algorithme se décompose en deux parties :

- **Partie Filtrage**.
  « **Entrée dans la session** »
- **Partie Capture.**
  « **Sortie dans la session** »

Cet algorithme est fondé sur l'algorithme de Knuth [Knu66]. Il se sert d'une variable *Turn* comme dans l'exclusion mutuelle classique, non pas pour assurer l'unicité du processus en section critique (comme dans [Knu66], mais l'unicité de la session ouverte. La modification de l'algorithme de Knuth constitue la partie *filtrage*.

La *partie filtrage* garantit que si des processus ouvrent une session à un instant donné, alors tous ces processus ouvrent la même session *X*. Dans la *partie capture*, les processus ayant ouvert la session entrent en section critique et avertissent les autres processus souhaitant accéder à *X* que celle-ci est ouverte. Tant que la session *X* est ouverte et qu'aucune autre session *Y* ($Y \neq X$ n'est demandée, tout processus désirant la session X entre directement en section critique en exécutant la *partie filtrage*. A partir du moment où au moins une session *Y* ($Y \neq X$ est demandée, toutes les demandes d'accès en section critique (y compris les nouvelles demandes pour la session *X*) restent bloquées dans la *partie filtrage* jusqu'à ce que tous les processus en section critique avec la session *X* aient quitté la section critique.

L'ouverture des sessions se fait dans l'ordre naturel des numéros de sessions demandées. Ainsi dans un système avec `8` sessions, si le numéro de session actuellement accessible est `4` et qu'il existe des demandes pour les sessions `1`, `3` et `6`, les sessions seront ouvertes, dans l'ordre cyclique `6`, `1` et `3`.

L'algorithme a une complexité en changement de contexte de *O(m)*. La complexité en temps est de $O(n \times m)$. Le degré de concurrence est de $O(n^3)$.

L'algortihme de Kean et Moir [KM01]. Dans [KM01], Kean et Moir présentent un algorithme d'exclusion mutuelle de groupe en considérant les systèmes *NUMA*[5] et *CC*[6]. Cet algorithme, que l'on notera KM_{GME}, peut lui aussi se décomposer en plusieurs parties :

- **Partie Filtrage**.
- **Accès à la session**.
- **Partie Capture.**

L'algorithme KM_{GME} est un algorithme de composition, les parties *Filtrage* et *Capture*

[5] *NUMA.*(pour Non-Uniform Memory Access). Un système *SMP* (Symmetrical MultiProcessing) relie plusieurs processeurs dans un même système. Les processeurs communiquent avec une mémoire distribuée commune via le bus d'interconnexion. *NUMA* est un peu différent de *SMP*. Les processeurs sont regroupés en petit groupe (« *Noeud* ») dans lequel tous les processeurs sont interconnectés entre eux. Chaque noeud possède sa propre mémoire. Un processeur peut accéder à la mémoire locale du noeud auquel il appartient ou celle d'un noeud distant. L'accès à la mémoire n'est pas uniforme car un processeur accède plus vite à la mémoire de son noeud par rapport à celle d'un noeud distant. D'où le nom Non-Uniform Memory Access. Exemple de machine *NUMA* : Origin 2000 de SGI, NUMA-Q de sequent.

[6] *CC.* (Cache Coherent). Un système cache cohérent doit faire en sorte que toutes les versions d'un même élément soient identiques. Potentiellement, une version peut être présente dans chaque cache et dans chaque module de mémoire. Toute modification doit être repercutée aux autres versions.

sont gérées à l'aide d'un algorithme d'exclusion mutuelle classique. Un processus *X* doit posséder le privilège pour pouvoir exécuter la partie *Filtrage* ou la partie *Capture*.

« Acquérir LOCK »
- **Partie Filtrage**.
  « Libérer LOCK »
- **Accès à la session**.
  « Acquérir LOCK »
- **Partie Capture.**
  « Libérer LOCK »

L'approche de cet algorithme est différente de celle de Joung. À chaque fois qu'un processus entre dans la section critique courante, il incrémente un compteur *NUM*, tout en prenant garde de décrémenter *NUM* lorsqu'il en sort. L'utilisation du compteur *NUM* permet à tout instant de connaitre le nombre de processus dans la session. Lorsqu'un processus n'obtient pas le droit d'entrer dans la session, celui-ci se met dans une file d'attente partagée. Cette file sert à sélectionner l'ordre des sessions. Lorsqu'un processus p_i veut entrer dans une session *X*, il doit attendre l'obtention du privilège (*LOCK*). Une fois le privilège obtenu, p_i peut exécuter la partie de *filtrage*. Un processus p_i peut entrer dans la session *X* et libérer le privilège, si l'une des deux conditions suivantes est vraie.

- *X* est la session courante et personne n'est en attente (la file d'attente est vide).
- *X* n'est pas la session courante mais personne n'utilise cette session.

Dans les autres cas, p_i se place dans la file d'attente, libère le privilège et attend.

Une fois que le premier processus accède à la session *X*, il attend encore le privilège pour pouvoir exécuter la partie de *capture*. Quand il l'obtient, p_i décrémente *NUM* de un, il n'utilise plus la session *X*. Ensuite, si la file d'attente n'est pas vide et qu'il n'y a pas de processus dans la session courante (*NUM*=`0`), p_i sait que la session peut être fermée. p_i change alors la session courante en prenant le numéro de session *Y* du premier élément de la file d'attente et, pour chaque élément de la file d'attente, p_i libère les processus qui souhaitent aussi accéder à *Y* et les comptabilise. On peut donc remarquer que si une nouvelle session *Y* est demandée, le dernier processus à sortir de la session *X* a en charge d'ouvrir *Y*.

Les complexités de cet algorithme dépendent directement de l'algorithme d'exclusion mutuelle sous-jacent utilisé pour gérer le privilège.

L'algorithme KM_{GME} a la propriété d'être *LocalSpin*. Un algorithme *LocalSpin* génère un nombre borné d'accès mémoire distant dans le pire des cas. Cette propriété est liée à l'utilisation et à la demande du privilège pour exécuter les différentes parties de l'algorithme. Au lieu de demander l'état du privilège, un processus travaille localement sur une copie et il est

averti quand cette copie n'est plus à jour (l'état du privilège a changé). Cette technique a pour effet de réduire les accès aux variables distantes.

De ce fait, l'algorithme de Kean et Moir limite au maximum le nombre d'accès aux variables distantes alors que l'algorithme de Joung en lit constamment. L'algorithme de Joung génère un nombre non-borné d'accès aux variables distantes (due aux boucles dans la phase de *filtrage* pour bloquer les processus). La principal avantage de l'algorithme de Kean et Moir est qu'il permet d'accéder en temps constant en l'absence de contention.

L'algorithme de Hadzilacos [Had01]. Dans [Had01], Hadzilacos propose de rédéfinir la propriété d'entrée concurrente. Il montre que la définition utilisée par Kean et Moir ne reflète pas fidèlement l'idée que Joung en avait.

A l'origine la définition de Joung [Jou98] était :

Entrée concurrente. Si des processus veulent accéder à une ressource, et qu'aucun autre processus ne veut accéder à une autre ressource différente, alors les processus peuvent y entrer concurremment.

Dans [KM01], Kean et Moir ont défini l'entrée concurrente de la manière suivante :

Entrée concurrente. Si un processus p veut accéder à une ressource X, et qu'aucun autre processus ne veut accéder à une ressource $Y(Y \neq X)$ alors le processus p finit par accéder à sa section critique.

Cette définition reflète moins la possibilité que plusieurs processus puissent entrer concurremment en section critique. Hadzilacos renomma la propriété d'entrée concurrente de Kean et Moir en *Occupation concurrente*. L'idée originelle de Joung était que les processus peuvent non seulement occuper la section critique en même temps, mais aussi y entrer sans « surplus de synchronisation ». Cela signifie qu'un processus ne peut pas retarder un autre processus quand il essaye d'entrer en section critique. L'*occupation concurrente* garantit qu'un processus en section critique ne peut pas retarder un processus qui essaye d'entrer en section critique, mais elle ne garantit pas que le processus ne sera retardé par d'autres processus souhaitant accéder à la section critique.

Hadzilacos redéfinit la propriété d'entrée concurrente de la manière suivante :

Entrée concurrente. Si un processus p veut accéder à une ressource X, et qu'aucun autre processus q ne veut accéder à une ressource $Y(Y \neq X)$ alors le processus p entre en section critique après un nombre borné d'étapes.

De plus, pour Hadzilacos, un algorithme d'exclusion mutuelle de groupe doit vérifier à la fois, la propriété d'occupation concurrente et d'entrée concurrente.

Dans le même article, il présente un algorithme ou les demandes sont servies de manière *FIFO*. Les complexités en changement de contexte, en temps et le degré de concurrence sont toutes de $\Theta(n)$.

1.10.6 Les algorithmes dans le modèle à passage de messages

Dans cette section, nous présentons les deux seuls algorithmes écrits dans ce modèle. Ils sont tous les deux fondés sur l'exclusion mutuelle classique de Ricart et Agrawala [RA81].

Algorithme sur le graphe complet. Dans [Jou98], Joung présente deux algorithmes sur un graphe complet. Chaque processus *p* possède un unique identificateur et peut communiquer avec tous ses voisins. Il modifie l'algorithme de Ricart et Agrawala pour résoudre le problème de l'*exclusion mutuelle de groupe*.

Dans le premier algorithme *RA*1, un processus désirant entrer en section critique, diffuse une requête aux autres processus, et entre en section critique quand tous les processus lui ont répondu. Le message de demande est de la forme ***Req***$(\langle i, sn_i \rangle, X)$, la partie $\langle i, sn_i \rangle$ est l'estampille et *X* est la ressource voulue. À la réception d'une demande, un processus p_j utilise l'une des règles suivantes pour décider s'il doit répondre à la requête :

1. p_j répond immédiatement s'il n'est pas intéressé par une autre session ou s'il est intéressé par un autre forum et la priorité de sa requête est inférieure à la requête de p_i.
2. La réponse est retardée si p_j est intéressé par une session différente et la priorité de sa requête est supérieure à celle de p_i.

En moyenne, l'algorithme *RA*1 offre de faibles performances. Pour s'en convaincre, supposons que deux processus p_i et p_j veulent accéder à la même session *X*, mais qu'un troisième processus p_k veut entrer dans une session différente *Y*(avec $Y \neq X$) et p_k a une priorité comprise entre celle de p_i et p_j. A cause de la priorité de p_k, p_i et p_j ne peuvent pas entrer concurremment dans la session, parce que le processus ayant la priorité la plus petite (on supposera que ce processus est p_j) doit attendre que p_k soit sorti de la session *Y* pour pouvoir entrer dans la session *X*.

L'algorithme *RA*2 résout ce problème de la façon suivante : tant que p_i est dans la session, s'il reçoit une demande de p_j pour la même session, alors p_i lui répond directement à l'aide d'un nouveau type de message (message *START*), autorisant p_j à rentrer directement dans la session. Donc le processus ayant la priorité la plus grande capture les autres processus. En revanche, l'entrée dans la session de p_j doit être connue de p_k. p_k a peut être déjà reçu la réponse de p_j et attend seulement la réponse de p_i. Donc, quand p_i quitte la session, dans la

	*RA*1	*RA*2
Nombre de messages	2(*n*-1)	$[n \cdots 3(n-1)]$
Complexité en temps	2(*n*-1)	$\frac{(n-1)(3n-2)}{2}$
Degré de concurrence	2(*n*-1)	∞
Complexité en changement de contexte	2(*n*-1)	2(*n*-1)

TAB. 1.1 – Complexités des algorithmes *GME* sur un graphe complet

	M	M_S	M_R
Nombre de messages	$6 \times \sqrt{\frac{2n(m-1)}{m}}$	$O(n \times \min(n,m))$	$O(\sqrt{n})$
Complexité en temps			$O(\sqrt{n})$
Degré de concurrence	$\sqrt{\frac{2n}{m(m-1)}}$		∞
Complexité en changement de contexte			2(*n*-1)

TAB. 1.2 – Complexités des algorithmes *GME* basés sur les quorums

réponse envoyée à p_k, p_i doit avertir qu'il a capture p_j.

Dans cet algorithme, une entrée en section critique nécessite entre *n* et 3(*n*-1) messages.

Dans le but de réduire le nombre de messages échangé, Joung, dans [Jou81], présente un algorithme basé sur les quorums. Il propose une méthode pour construire les quorums, qu'il appelle les quorums de surface. Utilisés avec l'algorithme de Maekawa [Mae85], ces quorums permettent jusqu'à $\sqrt{\frac{2n}{m(m-1)}}$ processus simultanément en section critique. Le nombre de messages nécessaire pour entrer en section critique est de $6 \times \sqrt{\frac{2n(m-1)}{m}}$.

Ensuite Joung propose deux modifications de l'algorithme de Maekawa afin que le nombre de processus simultanément ne soit pas limité au système de quorum utilisé. Ainsi, cette modification ne limite plus le nombre de processus simultanément en section critique et peut être utilisé avec un système de quorum classique. Pour le premier algorithme, le nombre de messages nécessaires pour entrer en section critique est de $O(n \times \min(n,m))$ avec un système de quorums de surface. dans le deuxième algorithme, le nombre de messages nécessaires pour entrer en section critique et la complexité en temps sont de $O(\sqrt{n})$.

Algorithme sur un anneau. Dans [WJ99b], Wu et Joung présentent quatres algorithmes fonctionnant sur un anneau unidirectionnel. Chaque processus p_i possède un identifiant unique *i* et ne peut communiquer qu'avec son voisin p_{i+1} (les opérations sur les identifiants sont *modulo n*).

De la même façon que l'algorithme de Joung sur le graphe complet, les auteurs modifient l'algorithme de Ricart et Agrawala pour résoudre le problème de l'*exclusion mutuelle de groupe*. L'algorithme présenté est adapté pour fonctionner sur un anneau unidirectionnel, il utilise le même principe de message que l'algorithme sur le graphe complet. Dans le premier algorithme, l'algorithme $CPT_{ring}1$, lorsqu'un processus demande une session, il envoie un message de demande à son voisin. Lorsqu'un processus reçoit un message de demande, il le laisse passer si la session demandée est la même que lui ou si la demande est prioritaire, dans le cas contraire (demande moins prioritaire) la demande est bloquée. Lorsqu'un message de demande a fait le tour de l'anneau, le processus peut entrer en section critique. En sortant de la section critique, il libère les demandes qu'il avait bloquées. Cet algorithme requiert *n* messages par entrée en section critique, mais la concurrence n'est due qu'à l'ordre des demandes, c'est-à-dire que si trois processus p_0, p_1 et p_2 font des demandes respectivement pour les sessions *X*, *Y* et *Z* et que p_0 peut entrer en section critique pour la session *X*, p_2 ne pourra pas entrer en section critique pour *X* seulement si la demande de p_2 est prioritaire par rapport à p_1.

Ce problème est résolu dans l'algorithme $CPT_{ring}2$. Lorsqu'un processus reçoit un message de demande, le processus peut entrer en section critique et envoie un message pour capturer les processus demandant la même session.
Dans cet algorithme, une entrée en section critique nécessite maintenant 2(*n*-1) messages mais la ressource est potentiellement mieux utilisée. La complexité en temps et le degré de concurrence est de $O(n^2)$ pour ces deux algorithmes. La complexité en changement de contexte est aussi de $O(n^2)$. La complexité en l'absence de contention n'est pas abordée dans [WJ99b], mais dans les algorithmes on peut dire que la complexité en l'absence de contention est égale à la complexité en nombre de message, soit, $\Theta(n)$.

Avec l'algorithme $CPT_{ring}3$, Wu et Joung ont réduit la complexité en changement de contexte de $O(n^2)$ en $O(\min(n, m))$.

	$CPT_{ring}1$	$CPT_{ring}2$	$CPT_{ring}3$
Nombre de messages	*n*	2(*n*-1)	2(*n*-1)
Complexité en temps	$O(n^2)$	$O(n^2)$	$O(n^2)$
Degré de concurrence	$O(n^2)$	$O(n^2)$	$O(n^2)$
Complexité en changement de contexte	$O(n^2)$	$O(n^2)$	$O(\min(n,m))$

TAB. 1.3 – Complexités des algorithmes *GME* sur un anneau

Dans sa thèse [Can03], S. Cantarell a présenté des algorithmes pour l'exclusion mutuelle de groupe fondés sur une circulation de jeton et écrit dans le modèle à passage de messages. Il faut noter qu'avant ses travaux de thèse les algorithmes d'exclusion mutuelle de groupe dans le modèle à passage de message étaient tous fondés sur la permission.

L'auteur a utilisé dans sa thèse [Can03] des algorithmes qui utilisent des messages de tailles et qui ont une occupation en espace qui est bornée. Dans les algorithmes proposés dans [Can03], il a été supposé que l'anneau est unidirectionnel et semi-uniforme, c'est-à-dire qu'à l'exception d'un unique processus distingué, les processus n'ont pas besoin d'être identifiés.

Le premier algorithme simple d'exclusion mutuelle de groupe présenté par S. Cantarell [Can03] est l'algorithme $\mathcal{GME}_{Rn}$. Cet algorithme utilise une sous-couche gérant l'exclusion mutuelle, plus spécifiquement, un mécanisme de circulation de jeton (aussi appelé *token ring*). L'algorithme $\mathcal{GME}_{Rn}$ n'utilise aucune file pour gérer les demandes de processus. L'occupation en espace des processus dépend seulement du nombre de ressources partagées *m*, et est de $4 \times \lceil \log(m+1) \rceil + 2$ bits. La taille des messages est de $2 \times \lceil \log(m+1) \rceil$ bits. Ainsi, l'algorithme $\mathcal{GME}_{Rn}$ résout le problème soulevé par [Jou98] qui est d'obtenir une solution qui utilise des messages de taille bornée. Dans cet algorithme, chaque accès en section critique génère entre 0 et $O(n)$ messages. La complexité en temps est de $O(n^2)$, mais la complexité en l'absence de contention est de $O(n)$. Le degré de concurrence n'est pas borné, ce qui montre que l'algorithme $\mathcal{GME}_{Rn}$ utilise au mieux les ressources.

Ensuite, S. Cantarell [Can03] a présenté dans sa thèse trois modifications de l'algorithme $\mathcal{GME}_{Rn}$. Ces trois algorithmes offrent de meilleurs performances que l'algorithme $\mathcal{GME}_{Rn}$ dans certaines situations. Le coût de l'algorithme dépend de *n* et est meilleur dans des systèmes avec $n \ll m$. Le deuxième algorithme noté $\mathcal{GME}_{Rm}$ est tel que son coût dépend principalement de *m* au lieu de *n*. La complexité en temps et la complexité en changement de contexte de l'algorithme $\mathcal{GME}_{Rm}$ sont respectivement de $O(n \times m)$ et $O(m)$. De plus, ces performances ont été atteintes sans avoir à ajouter d'informations sur au jeton.

Le troisème algorithme (l'algorithme $\mathcal{GME}_{Rnm}$) est un algorithme général qui tire profit de la plus basse valeur entre *n* et *m*. La complexité en changement de contexte est de $O(n * \min(n, m))$. Cependant l'algorithme $\mathcal{GME}_{Rnm}$ nécessite $O(\log \min(n, m))$ bits en plus pour le jeton et l'occupation en espace.

La quatrième algorithme, l'algorithme $\mathcal{GME}_{TL}$, réduit l'utilisation de la bande passante en évitant que le jeton ne circule inutilement en l'absence de contention (c'est-à-dire qu'il n'y a pas de nouvelle demande pour une session différente). L'algorithme $\mathcal{GME}_{TL}$ est une solution générale qui peut être combiné dans les trois algorithmes précédents.

Dans [Can03] toujours, il a été présenté des solutions pour l'exclusion mutuelle de groupe en utilisant une topologie en arbre. L'auteur a présenté trois algorithmes qui utilisent des messages de taille bornée et qui n'ont pas besoin d'identifiant. De plus ces algorithmes utilisent au mieux les ressources grâce à une complexité en changement de contexte de $O(\min(n, m))$.

Le premier algorithme, l'algorithme $\mathcal{GME}_{T1}$, utilise une racine fixe de l'arbre. Il propose une très bonne utilisation des ressources grâce à un degré de concurrence non borné. Dans cet algorithme, une entrée en section critique coûte entre 0 et $3 \times (n-1) + h$ messages, où *h* est la hauteur de l'arbre.

Ensuite, deux autres algorithmes (les algorithmes $\mathcal{GME}_{T2}$ et $\mathcal{GME}_{T3}$) qui sont des versions modifiées de $\mathcal{GME}_{T1}$ ont été présentés. Ces deux algorithmes utilisent entre 0 et $4 \times h$ message pas accès en section critique. Ainsi le nombre moyen de messages échangés pour une entrée en section critique est typiquement de $O(\log n)$.

1.11 Résumé du chapitre

De nombreux algorithmes distribués mettant en oeuvre l'exclusion mutuelle ont été conçus. Nous venons de définir une classification de ces algorithmes, au moyen d'outils, de mécanismes, et de techniques de conception.

Ces algorithmes traitent le problème de l'exclusion mutuelle pour l'accès à une seule ressource. Leur complexité, en nombre de messages, a été également présentée.

Nous avons aussi présenté quelques utilisations possibles de ce problème ainsi que les mesures permettant d'évaluer correctement un algorithme d'exclusion mutuelle de groupe.

Nous avons présenté plusieurs algorithmes distribué qui utilisent une horloge de Lamport. Contrairement à l'algorithme de Ricart et Agrawala, l'horloge ne peut pas être bornée entre *x* et *x*+*n*-1. En effet, lorsqu'un processus p_i est dans une session et qu'un processus p_j veut également y accéder alors que les autres processus ne demandent pas de session, p_j peut également faire un nombre arbitraire d'entrées dans la session, ce qui aura pour effet d'augmenter d'autant la valeur de son horloge.

L'occupation en espace mémoire n'est donc pas bornée et comme la valeur de l'horloge est envoyée dans les messages, la taille des messages ne l'est pas non plus.

CHAPITRE 2

Définitions et Propriétés

DAns ce chapitre, les définitions formelles des structures de *quorums* sont présentées. Ensuite, ces propriétés seront utilisées dans les preuves qui seront présentées.

2.1 Structures de quorum

Plusieurs auteurs ont défini les structures de *quorums* pouvant être utilisées dans une variété de protocoles distribués [BGM86, Fu90, GMB85, IK90]. Dans cette section, ces structures sont définies. Soit *U* un ensemble non vide de noeuds. Le terme *noeud* fait référence à une machine dans un réseau ou une copie d'une certaine donnée dans une base de données répliquée.

Définition 2.1 *Une collection d'ensembles, Q, est un quorum dans U si :*

1. $(\forall G \in Q)[G \neq \emptyset$ *et* $G \subseteq U]$

2. ***(*Minimalité) :** $(\forall G, H \in Q)[G \not\subseteq H]$**.**

Les ensembles $G \in Q$ sont appelés *quorums*. Par exemple, soit $U = \{a, b, c, d\}$. Alors, $Q = \{\{a, b\}, \{b, c\}\}$ est un quorum dans *U*. Il faut noter que tous les noeuds doivent apparaître dans un certain quorum ; en particulier, le noeud *d* n'apparaît pas dans l'un ou l'autre quorum de *Q*. Les noeuds qui apparaîssent dans un quorums sont appelés les *noeuds utilisés*.

Une notion similaire a été proposée par Sperner [Spe28]. Une famille *Q* de sous-ensembles de *U* est une famille de Sperner si *Q* est un quorum et $Q \neq \emptyset$. Ainsi, la seule différence dans

les définitions est que l'ensemble vide est une une famille de Sperner. Pour tout ensemble *S*, soit $|S|$ la cardinalité de *S*. Sperner a prouvé que le nombre de quorums dans une famille de Sperner est borné par $\binom{N}{\lfloor N/2 \rfloor}$, où $N = |U|$.

Un quorum *Q* peut être représenté par l'ensemble de tous les sous-ensembles de *U* contenant un quorum de *Q*. Un tel ensemble est appelé *acceptance set* correspondant à *Q*, et il est noté par $A(Q)$.

$$A(Q) = \{H \subseteq U \mid G \subseteq H \text{ pour } G \in Q\}$$

Dans l'exemple ci-dessus, avec $Q = \{\{a, b\}, \{b, c\}\}$ dans $U = \{a, b, c, d\}$, $A(Q)$ est donné par :

$$A(Q) = \{\{a, b\}, \{b, c\}, \{a, b, c\}, \{a, b, d\}, \{b, c, d\}, \{a, b, c, d\}\}$$

Définissons $\mathbf{min}(S) = \{G \in S \mid (\forall H \in S)[H \not\subset G]\}$ pour tout ensemble *S* de sous-ensembles non vides de *U*. Alors $Q = \mathbf{min}(A(Q))$ pour tout quorum *Q*. C'est une correspondance linéaire entre les acceptance sets et les ensembles quorums.

Définition 2.2 *Deux quorums sur U, notées Q_1 et Q_2, sont isomorphes s'il existe un isomorphisme $\pi : U \longrightarrow U$ tel que $Q_2 = \{\{\pi(x) \mid x \in G\} \mid G \in Q_1\}$.*

Définition 2.3 *Soient Q_1 et Q_2, deux quorums sur U. Alors, Q_1 domine Q_2 si*

1. $Q_1 \neq Q_2$.
2. $(\forall H \in Q_2)[\exists G \in Q_1 \mid G \subseteq H]$.

Un quorum *Q* sur *U* est *dominé* s'il existe un autre quorum sur *U* qui domine *Q*. Si un tel quorum n'existe pas, alors *Q* est *non dominé*.

Définition 2.4 *Un quorum Q est une coterie sur U si la propriété d'intersection est satisfaite, c'est-à-dire, $(\forall G, H \in Q)[G \cap H \neq \emptyset]$. Une coterie $Q = \{G\}$, contenant un seul quorum est appelé une coterie singleton.*

Une coterie *Q* sur *U*, est *dominée* s'il existe une autre coterie sur *U* qui domine *Q*. Si une telle coterie n'existe pas, alors *Q* est dit *non dominée*.

Soit *Q* un quorum sur *U*. Alors, un *quorum complémentaire* Q^c, est un autre quorum sur *U* tel que $(\forall G \in Q)(\forall H \in Q^c)[G \cap H \neq \emptyset]$.

La paire *b*=(Q, Q^c) est dite bicoterie sur *U*. Si *Q* ou Q^c est une coterie, alors la paire *B* est appelée *semicoterie*.

Supposons que $\mathbf{B_1}$=(Q_1, Q_1^c) et $\mathbf{B_2}$=(Q_2, Q_2^c) des bicoteries sur *U*. Alors, $\mathbf{B_1}$ domine $\mathbf{B_2}$ si

1. $\mathbf{B_1} \neq \mathbf{B_2}$; c'est-à-dire $Q_1 \neq Q_2$ ou $Q_1^c \neq Q_2^c$.
2. $(\forall H \in Q_2)[\exists G \in Q_1 \mid G \subseteq H]$.
3. $(\forall H \in Q_2^c)[\exists G \in Q_1^c \mid G \subseteq H]$.

Une bicoterie $\mathbf{B}$ sur *U* est *dominée* s'il existe une autre bicoterie sur *U* qui domine $\mathbf{B}$. Si une telle bicoterie n'existe pas, alors $\mathbf{B}$ est *non dominée*.

2.2 Propriétés

On considère les propriétés satisfaites par les quorums et coteries.

2.2.1 Quorums et coteries

Nous allons d'abord prouver qu'il y a une correspondance linéaire entre les quorums et les acceptance sets.

Lemme 2.1 *Soient Q_1 et Q_2 deux quorums sur U. Alors $A(Q_1) = A(Q_2)$ si et seulement si $Q_1 = Q_2$.*

Preuve.

Soient Q_1 et Q_2 deux quorums sur *U*. Supposons que $A(Q_1) = A(Q_2)$. nous allons supposer que $Q_1 \neq Q_2$ et aboutir à une contradiction. Puisque $Q_1 \neq Q_2$, soit

1. $(\exists G_1 \in Q_1)$ tel que $G_1 \notin Q_2, ou$
2. $(\exists G_2 \in Q_2)$ tel que $G_2 \notin Q_1$.

Sans perte de généralités, nous allons supposer qu'il existe un quorum $G_1 \in Q_1$ tel que $G_1 \notin Q_2$. Par définition de l'acceptance set, $G_1 \in A(Q_1)$. Puisque $A(Q_1) = A(Q_2)$, il s'en suit que $G_1 \in A(Q_2)$. Ainsi, il existe un quorum $G_2 \in Q_2$ tel que $G_2 \subseteq G_1$. Puisque $G_1 \notin Q_2$, il s'en suit que $G_2 \neq G_1$. Ainsi, $G_2 \subset G_1$. Puisque $G_2 \in Q_2$, il est aussi dans $A(Q_2))$. Cependant, $G_2 \notin A(Q_1)$, autrement il existerait $G_3 \in Q_1$ tel que $G_3 \subseteq G_2 \subset G_1$ et ceci contredit la minimalité de Q_1. Puisque $G_2 \in A(Q_2)$ et $G_2 \notin A(Q_1),$ on obtient $A(Q_1) \neq A(Q_2)$, et ceci est une contradiction. Ainsi, $A(Q_1) = A(Q_2)$.

Donc, $A(Q_1) = A(Q_2)$ si et seulement si $Q_1 = Q_2$.

Soit *Q* un quorum sur *U*. Un *transversal* de *Q* est un sous-ensemble de *U* qui a une intersection avec tous les quorums dans *Q*, c'est-à-dire, $H \subseteq U$ est un transversal de *Q* si $G \cap H \neq \emptyset$ pour tout $G \in Q$. Un *transversal minimal* est un transversal *H*, tel que tout sous-ensemble de *H* n'est pas un transversal. L'ensemble de tous les transversaux minimaux de *Q* noté **Tr**(*Q*), est un quorum complémentaire.
Tr=$\min\{H \subseteq U \mid (\forall G \in Q)[G \cap H \neq \emptyset])\}$.

Dans la littérature, **Tr(**Q) est aussi appelé *antiquorum* de *Q* [BGM86].

Lemme 2.2 *Définissons $\mathcal{P}(U)$ comme étant l'ensemble des parties de U. Si Q est un ensemble de quorums sur U, alors*

$$A(Tr(Q)) = \mathcal{P}(U) - \{U - G \mid G \in A(Q)\}$$

Preuve.

Soit $H \in A(Tr(Q))$. Alors, il existe un $G \in Tr(Q)$ tel que $G \subseteq H$. Supposons que $H \notin \mathcal{P}(U) - \{U - G \mid G \in A(Q)\}$. Alors $H \in \{U - G \mid G \in A(Q)\}$. Ainsi, il existe un $K \in A(Q)$ tel que $H = U - K$. Puisque $G \subset H$ et $H \cap K = \emptyset$, il s'en suit que $G \cap K = \emptyset$. Aussi, puisque $K \in A(Q)$, il existe un $L \in Q$ tel que $L \subseteq K$. Finalement, puisque $L \subseteq K$ et $G \cap K = \emptyset$, il s'en suit que $G \cap L = \emptyset$. Ceci est une contradiction car $G \in Tr(Q)$ et $L \in Q$. Ainsi, notre supposition est fausse, et $H \in \mathcal{P}(U) - \{U - G \mid G \in A(Q)\}$.

De l'autre côté, supposons que $H \in \mathcal{P}(U) - \{U - G \mid G \in A(Q)\}$. Prenons $H \notin A(Tr(Q))$. Alors il n'existe aucun $G \in Tr(Q)$ tel que $G \subseteq H$. Puisque $Tr(Q)$ est l'ensemble des transversaux minimaux, il doit y avoir un certain $K \in Q$ tel que $H \cap K = \emptyset$. Prenons $L = U - H$. Alors, $K \subset L$, ainsi $L \in A(Q)$, et il s'en suit que $H \in \{U - G \mid G \in A(Q)\}$ car $H = U - L$. Ceci est une contradiction. Ainsi, $H \in A(Tr(Q))$.
Donc, $A(Tr(Q)) = \mathcal{P}(U) - \{U - G \mid G \in A(Q)\}$.

Lemme 2.3 *Soit Q un ensemble de quorums sur U. Alors aucun autre ensemble de quorums complémentaires est dominé par Tr(Q).*

Les ensembles de quorums non dominés ne sont pas vraiment interessants. Si *U* est un ensemble non vide de noeuds, alors le seul ensemble de quorums non dominé sur *U* est $Q = \{\{x\} \mid x \in U\}$.

Cependant, il peut y avoir beaucoup de coteries non dominées sur *U*. Le théorème suivant détermine plus facilement si une coterie est dominée.

Théorème 2.1 *Soit Q une coterie sur un ensemble non vide U. Alors, Q est dominée si et seulement si il existe un ensemble $H \subseteq U$ tel que :*

1. $G \in Q \Rightarrow G \not\subseteq H$.
2. $G \in Q \Rightarrow G \cap H \neq \emptyset$.

Par exemple, prenons $Q_2 = \{\{a, b\}, \{b, c\}\}$. L'ensemble $H = \{a, b\}$ satisfait les propriétés du théorème 4.1. Officieusement, on dit que *H domine* Q_2. Notons que Q_2 est dominée par $Q_1 = \{\{a, b\}, \{a, c\}, \{b, c\}\}$. La coterie Q_1 est non dominée. La coterie Q_2 est aussi dominée par la coterie $\{\{c\}\}$. Finalement, notons que si h satisfait les propriétés du théorème 4.1, ainsi que l'ensemble $U - H$. Dans l'exemple ci-dessus, $U - H = \{c\}$.

Théorème 2.2 *[GMB85] Soit Q une coterie sur un ensemble non vide U, où $|U| = N$. Alors, les propriétés suivantes sont équivalentes :*

1. *Q est non dominée*
2. $|A(Q)| = 2^{N-1}$
3. $Tr(Q) = Q$.

Preuve.

Par le théorème 4.1, *Q* est non dominée si et seulement si $Tr(Q) = Q$. Supposons que *Q* est non dominée et par conséquent $Tr(Q) = Q$. Par le lemme 5.1, $A(Tr(Q)) = \mathcal{P}(U) - \{U - G \mid G \in A(Q)\}$, et ainsi il s'en suit que $|A(Tr(Q))| = |\mathcal{P}(U)| - |A(Q)|$. Ainsi $|A(Tr(Q))| + |A(Q)| = 2^N$. Par le lemme 5.2, $A(Q) = A(Tr(Q))$, et ainsi il s'en suit que $|A(Q)| = 2^{N-1}$.

Finalement supposons que $|A(Q)| = 2^{N-1}$. Si $H \in A(Q)$, alors $U - H \notin A(Q)$ grâce à la propriété d'intersection imposée aux coteries. Ainsi, $A(Q)$ est le plus grand acceptance set ; c'est-à-dire, étant donné un $H \in \mathcal{P}(U)$, soit *H* ou *U-H* doit être dans $A(Q)$.
Supposons que Q est dominé. Par le théorème 4.1, il doit exister un certain $H \subseteq U$ qui domine Q. Ainsi, $H \notin A(Q)$. rappelons que $U - H$ domine aussi *Q*, ainsi $U - H \notin A(Q)$. Ceci est une contradiction.
Donc, les trois propriétés sont équivalentes.

2.2.2 Bicoterie

Soit *Q* un ensemble de quorums sur *U*. Soit Q^c un ensemble quorum complémentaire. La paire $B = (Q, Q^c)$ est appelé un *quorum agreement* si $Q^c = Tr(Q)$. Il est facile de montrer que les quorums agreement sont les mêmes que les bicoteries dominées. Pour éviter une confusion, nous allons utiliser le terme bicoteries non dominées dans le reste de du chapitre. Pour toute bicoterie non dominée (Q, Q^c), il existe seulement trois possibilités [BGM86, IK90] :

1. *Q* et Q^c sont des coteires non dominées ;
2. *Q* est une coterie non dominée et Q^c n'est pas une coterie (ou par équivalence, Q^c est une coterie non dominée et *Q* n'est pas une coterie) ; ou
3. ni *Q* ni Q^c est une coterie.

Théorème 2.3 *Soit $B = (Q, Q^c)$ une bicoterie sur un ensemble non vide U. Alors, B est dominée si et seulement si il existe un ensemble $H \subseteq U$ tel que :*

1. $G \in Q \Rightarrow G \cap H \neq \emptyset$.
2. $G \in Q^c \Rightarrow G \not\subseteq H$.

Théorème 2.4 *Soit $B = (Q, Q^c)$ une bicoterie sur un ensemble non vide U, où $|U| = N$. Alors, les propriétés suivantes sont équivalentes :*

1. *B est non dominée.*
2. $|A(Q)| + |A(Q^c)| = 2^N$.
3. *B est un quorum agreement, c'est-à-dire, $Q^c = Tr(Q)$.*

Preuve.

Par le lemme 4.3 et le théorème 2.3, *B* est une bicoterie non dominée si et seulement si *B* est un quorum agreement.

Supposons que $B = (Q, Q^c)$ est non dominée, et par conséquent $Q^c = Tr(Q)$. Alors par le lemme 5.1, $A(Q^c) = \mathcal{P}(U) - \{U - G \mid G \in A(Q)\}$. Puisque $|\mathcal{P}(U)| = 2^N$, il s'en suit que $|A(Q)| + |A(Q^c)| = 2^N$.
De l'autre côté, supposons que $|A(Q)| + |A(Q^c)| = 2^N$. Supposons que $Q^c \neq Tr(Q)$. Par le lemme 5.2, $A(Q^c) \neq A(Tr(Q))$, et il s'en suit que qu'il doit exister un certain $H \in A(Tr(Q))$ tel que $H \notin A(Q^c)$. Ainsi, $|A(Q^c)| < [A(Tr(Q))|$. Cependant, si $G \in A(Q)$, alors $U - G \notin A(Tr(Q))$ grâce à la propriété d'intersection.

Ainsi $|A(Q)| + |A(Tr(Q))| \leq 2^N$, et ceci est une contradiction.
Donc toutes ces trois proprétés sont équivalentes.

2.3 Résumé du chapitre

Les systèmes de quorums sont des constructions bien connus qui permettent d'améliorer les performances et la disponibilité des systèmes distribués. Ils sont aussi utilisés comme un moyen pour améliorer le temps de réponse de services déployés sur des grilles de calcul.
Ils ont été employés pour mettre en applicaton beaucoup de problèmes de coordination dans les systèmes répartis tels que l'exclusion mutuelle, la réplication des données etc.

Nous avons présenté dans ce chapitre les définitions formelles des structures de quorums. Nous allons utiliser ces définitions dans le but de proposer dans la suite des algorithmes pour le problème de l'exclusion mutuelle de groupe basé sur les quorums.

CHAPITRE 3

Exclusion mutuelle de groupe basée sur les quorums

DAns le chapitre précédent, nous avons donné les définitions formelles des structures de quorums. Dans ce chapitre, nous présentons un algorithme d'exclusion mutuelle de groupe basé sur les quorums. Nous présentons aussi un autre algorithme d'exclusion mutuelle de groupe pour les accès concurrents des sessions basé sur les quourms.

3.1 Algorithme d'exclusion mutuelle de groupe basé sur les quorums

Dans [Jou01b], Joung présente le système de quorum de surface (surficial quorum system) pour l'*exclusion mutuelle de groupe*. Il présente aussi une modification de l'algorithme de Maekawa [Mae85] de sorte que le nombre de processus qui peuvent accéder à une ressource en un temps fini ne soit pas limité à la structure du système de quorums. Il ne distingue pas les processus et les ressources, c'est-à-dire qu'un processus peut appartenir à un ou plusieurs groupes. Le processus doit identifier un unique groupe dans lequel il appartient quand il veut entrer en *section critique*.

Dans ce travail, nous avons fait une différence entre les processus et les ressources. Nous présentons aussi une modification de l'algorithme de Maekawa. Notre algorithme d'exclusion mutuelle de groupe est basé sur les quorums. Le nombre de processus qui peuvent entrer en *section critique* n'est pas limité [TN06].

Le problème est de proposer un algorithme pour un système satisfaisant les conditions suivantes :

- **Exclusion mutuelle (Surêté)** : À tout moment, deux sessions de différents groupes ne peuvent être ouvertes simultanément.
- **Absence de blocage (Vivacité)** : Une session demandée sera accessible au bout d'un temps fini.
- **Entrée concurrente (Efficacité)** : Si un groupe *g* de processus demandent une session et aucun autre groupe de processus n'est intéressé d'une session différente, alors les processus du groupe *g* peuvent accéder en *section critique* de façon concurrente [Jou98, KM01, Had01].

Notons que la dernière propriété est une conséquence triviale de la deuxième.
Nous considérons un système de processus asynchrones $P_1, \cdot, P_n$, chacun d'entre eux tourne à travers les trois états suivants, avec *NSC* (non section critique) étant l'état initial du système.

- **Non Section Critique** : Le processus est en dehors de *SC* (section critique) et ne souhaite pas entrer en *section critique*.
- **Attente** : Le processus attend d'entrer en *section critique* mais n'est pas encore entré.
- **Section Critique** : Le processus est en *section critique*.

3.1.1 Algorithme basé sur les quorums pour l'exclusion mutuelle de groupe

Dans cette section, nous présentons un algorithme utilisant les quorums pour résoudre le problème de l'exclusion mutuelle de groupe. Le réseau est supposé complet et fiable. Dans le chapitre 2, les définitions formelles sur les quorums et coteries ont été présentés.

3.1.1.1 Motivation et Contribution

La difficulté majeure dans le problème de l'exclusion mutuelle de groupe est d'éviter une situation d'interblocage, c'est-à-dire le blocage mutuel de plusieurs processus du système. Le problème est de gérer la propriété d'entrée concurrente pour les processus désirant entrer en section critique, qui a été définie ci-dessus.

Dans [Jou01b], Joung a présenté les quorums de surface pour résoudre le problème de l'exclusion mutuelle de groupe. Il a ensuite présenté deux modifications de l'algorithme de Maekawa [Mae85]. Dans notre travail, nous nous sommes intéressés au problème de l'exclusion mutuelle de groupe basé un système ordinaire de quorums. Notre but a été de présenter une solution au problème de l'exclusion mutuelle de groupe, qui est aussi une modification de l'algorithme de Maekawa, qui réduit le nombre de messages pas entrée en section critique, par rapport à l'algorithme de Joung [Jou01b], en $O(\sqrt{n} + |Q|)$, où *n* est le nombre de processus et $|Q|$, la taille du quorum choisi.

L'algorithme que nous avons présenté dans cette section permet aux processus non seulement d'occuper la section critique en même temps, mais aussi d'y entrer sans surplus de synchronisation.

Dans notre solution proposée, les requêtes sont envoyées simultanément par *p* aux noeuds de *Q* pour minimiser le délai de synchronisation, qui est le délai entre le temps qu'un processus souhaite entrer en *section critique* et celui d'entrée en *section critique*. Cependant, en raison du fait que le système est asynchrone, un processus doit garder une requête tout en attendant qu'un autre processus libére la session. Ceci peut créer un blocage.

L'algorithme de Maekawa résout le phénomène de blocage en exigeant une relation d'ordre d'exécution entre les processus de basse priorité et les processus de haute priorité.

3.1.2 Principe de l'algorithme

L'algorithme présenté ici peut être directement adapté à l'exclusion mutuelle de groupe comme suit. Un processus $P \in \mathcal{P}$ qui veut entrer en section critique chosit un groupe *g*, selectionne un quorum arbitraire *Q*, et entre en section critique seulement s'il reçoit la permission de tous les membres du quorum *Q*.

Notre algorithme fonctionne de la manière suivante : quand un processus P_i veut participer à une session *x*, il envoie un message *REQ*() à *x* afin d'obtenir un message *OK*() de la part de ce dernier. La session *x* à son tour envoie un message *Open_Session*() à tous les membres de son groupe. Si *x* reçoit tous les messages *Aut_Session*() de son groupe, elle envoie un message *OK*() au processus P_i. Lors de la sortie de la section critique, le processus P_i envoie un message de libération *REL*() à la session *x*.

Supposons qu'un membre du quorum donne la permission à un seul processus à la fois. Alors, par la propriété d'intersection, deux processus ayant demandé différents groupes ne

peuvent entrer en section critique simultanément. La propriété de minimalité est utilisée plutôt pour améliorer l'efficacité.

Les groupes sont en compétition pour l'entrée en section critique et sont de priorité égale (aucun n'est privilégié). Le nombre de messages requis pour entrer en section critique est indépendant du quorum choisi par un processus. Chaque noeud partage la même responsabilité dans le système.

Pour la simplicité, nous ne donnerons pas ici le code explicite pour manipuler les horloges logiques. L'estampille pour la requête courante est maintenant par une variable H_i. Une relation d'ordre notée « < » sur les estampilles est définie de la façon suivante :
$\langle P_i, H_i \rangle < \langle P_j, H_j \rangle$ si et seulement si $(H_i < H_j)$ ou $((H_i = H_j) \wedge (P_i < P_j))$.

Les priorités sont implémentées par les horloges logiques qui ont été définies par [Lam78b]. Plus l'estampille est petite, plus la priorité de la requête est haute. Spécifiquement, si un noeud *i* reçoit une requête par un noeud *p* après avoir donné une permission à *q* et que la priorité de *p* est plus haute que celle de *q*, alors *i* envoie un message d'investigation (message **Interrogation**) à *q*. Le processus *q* alors retourne à *i* sa requête (en lui envoyant un message de refus) s'il ne peut pas avoir une permission de tous les noeuds du quorum *Q* choisi. Alors le noeud *i* donne à *p* sa permission après avoir reçu la requête de *q*. Quand *p* sort de la *section critique* et libère la requête de *i*, il retourne la requête à *q* (vraisemblablement aucun autre processus de plus haute priorité que *q* est en attente de la permission du noeud *i*).

3.1.2.1 Messages echangés par processus/session et session/session

Les messages échangés entre les processus et les sessions d'une part et entre les sessions d'autre part dans l'algorithme sont :
$Open\_Session(x, H_x)$: message de demande d'ouverture de la session *x*.
$Aut\_Session(x, H_x)$: message d'autorisation de l'ouverture de la session *x*.
$End\_Session(x, H_x)$: message de fermeture de la session *x*.
$REQ(P, x)$: message envoyé par *P* pour demander la participation à la session *x*.
$OK(x)$: messade d'autorisation envoyé par *P* afin de participer à la session *x*.
$REL(x)$: message signifiant que *P* a fermé la session *x*.

3.1.2.2 Variables locales à la session x

Les variables locales utilisées dans l'algorithme pour la session *x* sont :
H_x : un nombre d'ordre utilisé pour implémenter la requête logique de la session *x*. Il est initialisé à *0*.

H_r : un nombre d'ordre utilisé pour implémenter la requête logique de la session *x*. Il est utilisé pour estampiller la requête afin d'ouvrir la session *x*. Il est initialisé à *0*.
$Next_x$: groupe de sessions en attente de participation à la session *x*. Initialement $Next_x =$ `Nil` pour toute session *x*.
HAT_x : boolean, `True` si la session *x* recoit un message *Aut_Session*() de son groupe.
WS_x : groupe de processus en attente de participation à la session *x*. Initialement, $WS_x = \emptyset$ pour toute session *x*.
$Nrel_x$: désigne le nombre de messages de libération que la session *x* attend de l'ensemble des processus. Initialement, $Nrel_x = 0$ pour toute session *x*.
X : où $X = \{x, y, z, \cdots\}$ est un ensemble dynamique de *m* sessions dans le réseau.
Q : quorum sélectionné par le processus *P*.
g : le groupe dans lequel *P* appartient.
status : le statut du processus *P*.
lockednodes : ensemble de noeuds ayant donné la permission à *P*. Initialement $lockenodes = \emptyset$.
Add() : ajoute un noeud demandeur *i* dans la file $Next_x$ telle que le noeud précédent *i* dans $Next_x$ est tel que $(j, H_j) < (i, H_i)$ et le noeud après *i* dans $Next_s$ est tel que $(k, H_k) > (i, H_i)$.
Remove() : enlève le noeud à la tête de la file $Next_x$.
Head() : retourne le premier noeud de la file $Next_x$.
SC : désigne la section critique.

3.1.2.3 Etat des processus

Les processus sont définis par trois états :
NSC : le processus est en dehors de la *section critique*.
Wait : le processus désire entrer en *section critique*, mais il est en attente.
SC : le processus est en *section critique*.

3.1.2.4 Règles pour les processus

Algorithme 3.1 When a process P_i wants to enter *SC*

1.01 $status \leftarrow Wait$
1.02 $group \leftarrow g$
1.03 **Select** an arbitrary quorum *Q*
1.04 **Send** *REQ*(P_i, x) **To** session *x* /* P_i peut accéder à la session *x* */

Algorithme 3.2 When a process P_j receives *REQ*(P_i, x)

1.05 $lockenodes \leftarrow Q$
1.06 $status \leftarrow SC$

Algorithme 3.3 When a process P_i receives *OK*(*x*)

1.07 $status \leftarrow NSC$
1.08 **Send** *REL*(P_i) **To** session *x*
1.09 $lockenodes \leftarrow \emptyset$

3.1.2.5 Règles pour les sessions

Algorithme 3.4 When a session *x* receives *REQ*(P_i, x)

1.10 **Do**
1.11 **If** $((HAT_x) \wedge (Next_x = \texttt{Nil}))$ **Then**
1.12 **Send** *OK*() to P_i
1.13 $Nrel_x \leftarrow Nrel_x + 1$
1.14 **Else**
1.15 **If** $((Next_x = \texttt{Nil}) \wedge (WS_x = \emptyset))$ **Then**
1.16 $H_x \leftarrow H_x + 1$
1.17 **Send** *Open_Session*(x, H_x) **To** all $q \in Q$
1.18 $XX \leftarrow Q$
1.19 $Next_x \leftarrow Add(Next_x, (x, H_r))$
1.20 $H_r \leftarrow H_x$
1.21 **EndIf**
1.22 $WS_x \leftarrow WS_x \cup \{P_i\}$
1.23 **EndIf**
1.24 **EndDo**

Algorithme 3.5 When a session *x* receives *Open_Session*(y,H)

1.25 **Do**
1.26 $H_x \leftarrow \max(H_x, H) + 1$
1.27 **If** $(Next_x = \texttt{Nil})$ **Then**
1.28 **Send** *Aut_Session*(x, H_x) to *y*
1.29 $HAT_x \leftarrow \texttt{False}$
1.30 **Else** /*Voir dans $Next_x$ **si** *(y,H)* a la priorité sur toutes les autres sessions **sinon** voir Maekawa [Mae85] */
1.31 **EndIf**
1.32 **EndIf**
1.33 $Next_x \leftarrow Add(Next_x, (y, H))$
1.34 **EndDo**

Le cas où $((Nrel_x = 0) \wedge (WS_x \neq \emptyset))$ peut causer un blocage. Spécialement, si une session *z* reçoit une permission d'une session *x* après avoir donné une permission à *y* et que la priorité de *x* est plus haute que celle de *y*, alors *z* envoie un message d'investigation (message **Interrogation**) à *y*. Voir Maekawa [Mae85].

Algorithme 3.6 When a session *x* receives *REL*(P_i, x)

1.35 **Do**
1.36 $Nrel_x \leftarrow Nrel_x - 1$
1.37 **If** $(Nrel_x = 0)$ **Then**
1.38 $Remove(Next_x, Head(Next_x))$
1.39 **Send** *End_Session*() **To** all $y \in Q$
1.40 **If** $(Next_x \neq \texttt{Nil})$ **Then**
1.41 **Send** *Aut_Session*(x, H_x) **To** *Head*$(Next_x)$
1.42 $HAT_x \leftarrow \texttt{False}$
1.43 **If** $(WS_x \neq \emptyset)$ **Then**
1.44 $H_x \leftarrow H_x + 1$
1.45 **Send** *Open_Session*(x, H_x) to all $q \in Q$
1.46 **EndIf**
1.47 **EndIf**
1.48 **EndIf**
1.49 **EndDo**

Algorithme 3.7 When a session *x* receives *Aut_Session(y,H)* from all $q \in Q$

```
1.50  Do
1.51      XX ← XX − {y}
1.52      If (XX = ∅) Then
1.53          HAT_x ← True
1.54          For all P_i ∈ WS_x Send OK(x) to P_i
1.55              Nrel_x = |WS_x|
1.56              WS_x ← ∅
1.57      EndIf
1.58  EndDo
```

Algorithme 3.8 When a session *x* receives *Aut_Session(y,H)* from all $q \in Q$

```
1.59  Do
1.60      H_x ← max(H_x, H)
1.61      Remove(Next_x, (y, H))
1.62      If (Next_x ≠ Nil) Then
1.63          If (Head(Next_x) ≠ x) Then
1.64              Send Aut_Session(x, H_x) To Head(Next_x)
1.65          EndIf
1.66      EndIf
1.67  EndDo
```

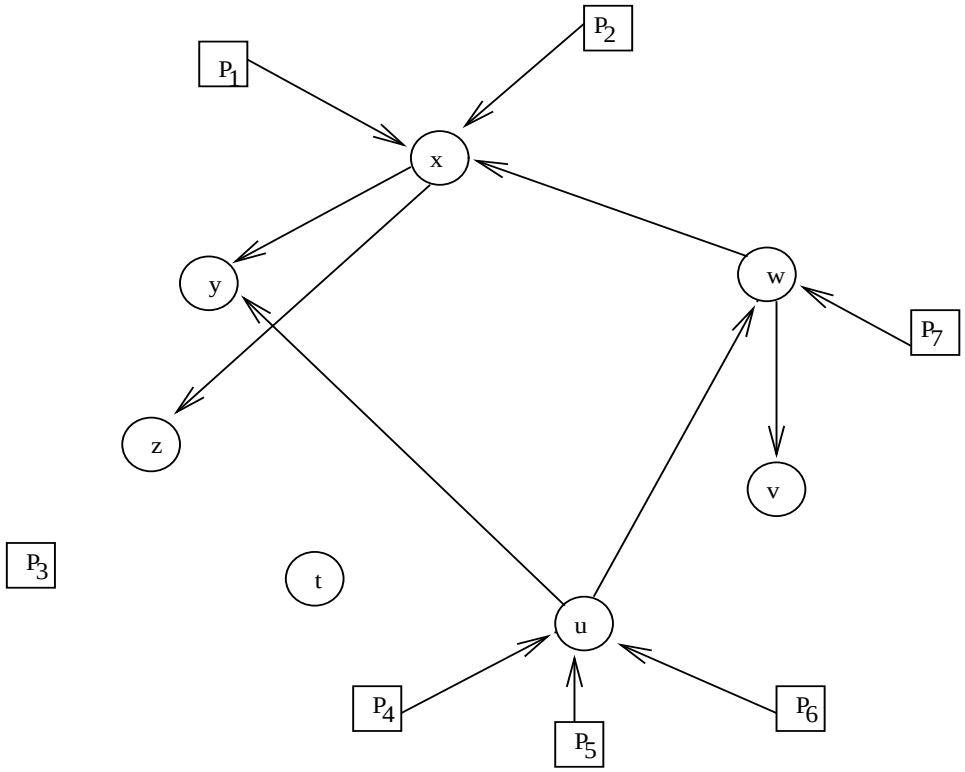

FIG. 3.1 – Messages échangés entre les processus/sessions et entre les sessions

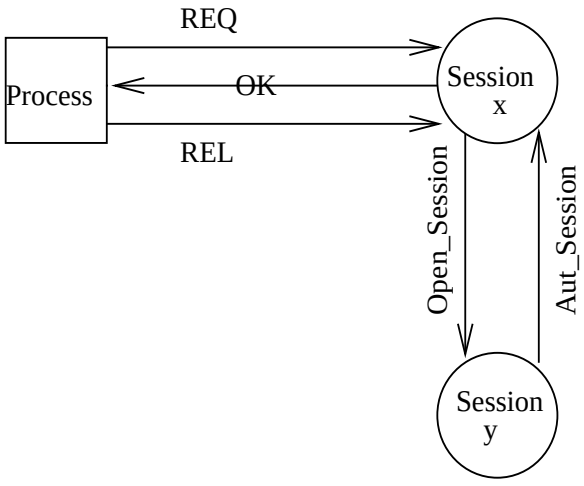

FIG. 3.2 – Messages échangés entre les processus et les sessions

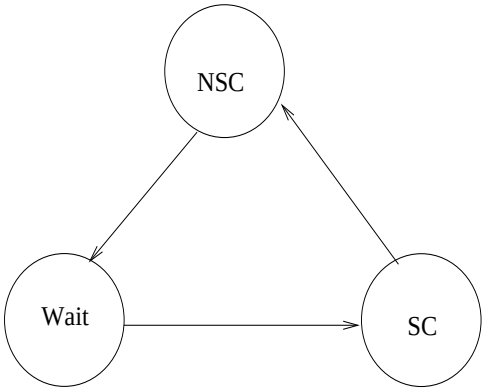

FIG. 3.3 – Etats des processus

3.1.3 Exemple d'illustration

Considérons un système de 7 quorums $(Q_i)_{1 \leq i \leq 7}$ et supposons que l'on dispose des ressources $\{x, y, z, t, u, v, w\}$ (Figure 3.1).

$Q_1 = \{P_1, P_2, P_3\}$ $G_1 = \{x, y, z\}$
$Q_2 = \{P_2, P_4, P_6\}$ $G_2 = \{y, t, v\}$
$Q_3 = \{P_3, P_4, P_7\}$ $G_3 = \{z, t, w\}$
$Q_4 = \{P_5, P_2, P_7\}$ $G_4 = \{u, y, w\}$
$Q_5 = \{P_5, P_4, P_1\}$ $G_5 = \{u, t, x\}$
$Q_6 = \{P_6, P_5, P_3\}$ $G_6 = \{v, u, z\}$
$Q_7 = \{P_7, P_1, P_6\}$ $G_7 = \{w, x, v\}$

Maintenant, nous donnons une illustration de notre exemple par le scénario suivant :

E_1 : Les processus P_1 et P_2 choisissent le quorum Q_1, ils envoient des messages *REQ*() à la session *x*.
E_2 : P_7 envoie un message *REQ*() à la session *w* après avoir choisi le quorum Q_7.
E_3 : Les processus P_4, P_5 et P_6 choisissent le quorum Q_4 et envoient un message *REQ*() à la session *u*.
E_4 : La session *x* après avoir reçu les messages *REQ*() des processus P_1 et P_2, envoie un message *Open_Session*() à tous les membres de son groupe plus précisèment aux sessions *y* et *z*.
E_5 : La session *w* reçoit le message *REQ*() du processus P_7, il envoie un message *Open_Session*() aux sessions *x* et *v*.
E_6 : La session *u* reçoit le message *REQ*() du processus P_5, il envoie un message *Open_Session*() aux sessions *y* et *w*.
E_7 : La session *x* reçoit un message *Aut_Session*() de la session *z* et doit aussi attendre

le même message provenant cette fois ci de la session *y*.

E_8 : La session *x* reçoit un message *REQ*() du processus P_2. Le groupe de processus en attente de participation à la session *x*, WS_x contient maintenant les processus P_1 et P_2.

E_9 : La session *y* envoie un message *Aut_Session*() à la session *u*, mais *u* ne peut pas envoyer un message *OK*() au processus P_5 car elle n'a pas encore reçu le message *Aut_Session*() de la session *w*.

E_{10} : La session *u* reçoit les messages *REQ*() des processus P_5 et P_6.

E_{11} : Les processus P_4, P_5 and P_6 sont maintenant en attente dans l'ensemble WS_u.

E_{12} : Maintenant la session *y* envoie un message *Aut_Session*() à *x*.

E_{13} : *x* reçoit le message *Aut_Session*() de la session *y* et envoie un message *OK*() aux processus P_1 et P_2.

E_{14} : Les processus P_1 et P_2 envoient des messages de libération *REL*() à la session *x*.

E_{15} : La session *x* envoie un message *Aut_Session*() aux sessions *y* et *z*.

3.1.4 Preuve de l'algorithme

3.1.4.1 Exclusion mutuelle

Supposons le contraire, c'est-à-dire que plusieurs sessions sont ouvertes simultanément. Les arguments suivants prouvent que ceci n'est pas possible.

- Toute session en *section critique* doit recevoir un message *Aut_Session*() de tous les membres de son groupe au préalable.
- Si deux sessions différentes *x* et *y* de différents groupes sont toutes les deux ouvertes, alors par la propriété d'exclusion mutuelle du groupe de système de quorums, les quorums qui sont choisis reçoivent un commun message *REQ*() d'un processus, soit *P*. Toutes les deux sessions *x* et *y* doivent envoyer un message *OK*() au processus *P* et les sessions *x* et *y* doivent aussi appartenir à un même groupe.

Théorème 3.1 *Au plus une seule session est ouverte à la fois.*

Preuve.

Si deux processus P_i et P_j sont en *section critique*, nous avons deux cas :

1. P_i et P_j sont dans la même session S_i.
2. P_i et P_j sont dans deux sessions différentes S_i et $S_j (i \neq j)$ et sont en *section critique*. Si le processus P_i a une plus haute priorité que celle de P_j alors :

$$(P_j, H_j) > (P_i, H_i) \text{ si et seulement si } (H_j > H_i) \; ou \; ((H_j = H_i) \wedge (P_j > P_i)). \quad (3.1)$$

Si le processus P_j a une plus haute priorité que celle de P_i alors :

$$(P_i, H_i) > (P_j, H_j) \text{ si et seulement si } (H_i > H_j \; ou \; ((H_j = H_i \wedge (P_i > P_j)). \quad (3.2)$$

Grâce aux relations (3.1) et (3.2), nous avons ($P_i = P_j$) et ainsi P_i et P_j doivent nécessairement être dans la même session. Ce qui est en contradiction avec notre supposition.

Cependant, ceci contredit la spécification de l'algorithme selon laquelle chaque session envoie seulement un message *OK*() à un processus à la fois. Donc, plus d'une seule session ne peut être ouverte simultanément.

3.1.4.2 Absence de famine

L'absence de famine est le fait que toutes les requêtes d'entrée en *section critique* seront satisfaites au bout d'un temps fini.

Quand une session *x* veut être ouverte, si son message *Aut_Session*() a une priorité plus haute que celle de tous les messages identiques, alors la session *x* sera eventuellement ouverte. Les priorités sont implémentées par des horloges logiques croissantes et une session augmente son horloge logique à chaque fois qu'elle est ouverte. Ainsi, après la session *x* envoie un message *Open_Session*(x, H_x) aux membres de son groupe. Etant donné qu'une session est ouverte au bout d'un temps fini, les requêtes de plus haute priorité que (x, H_x) cesseront éventuellement d'exister. Ainsi la session *x* sera éventuellement ouverte.

3.1.4.3 Absence de blocage

Le phénomène de blocage a lieu quand il existe une attente circulaire parmi un groupe de noeuds essayant de réaliser l'exclusion mutuelle. Les ensembles de requêtes de deux noeuds quelconques dans le réseau ont un noeud en commun. Ce noeud commun peut uniquement arbitrer les demandes d'exclusion mutuelle selon la période de leurs arrivées et la priorité relative à ces demandes. Ainsi, si un processus *p* est en attente d'une permission de *i*, alors toute autre permission de *p* doit provenir d'un certain processus *j* tel que $j < i$. Plus précisèment, chaque processus *q* en donnant à *i* une permission a soit reçu toutes les permissions des membres de son groupe, soit est en attente d'une permission de *k* tel que $k > i$. Ainsi, les blocages ne sont pas possibles parce qu'il n'y a pas d'attente circulaire.

3.1.5 Performances

Une des mesures de performance d'un algorithme distribué est le nombre de messages échangés dans l'ordre d'accomplir le but désiré. Ici, le nombre de messages pour réaliser l'exclusion mutuelle de groupe est examiné dans le cas de la reconfiguration du réseau. Dans l'algorithme de Maekawa [Mae85], puisque qu'un noeud est autorisé par seulement un processus à la fois, le nombre maximal de processus d'un groupe qui peuvent entrer simultanément en section critique est limité au degré du cartel qui lui est associé. La compléxité de

notre algorithme est de $O(\sqrt{n} + |Q|)$ messages où $|Q|$ est la taille du quorum adopté dans notre algorithme. Notons aussi que le nombre de messages requis entre un processus et une session est de 3 pour chaque accès à la session.

3.2 Accès concurrents de sessions basés sur les quorums

Après avoir proposé dans la section 3.1, un algorithme d'exclusion mutuelle de groupe basé sur les quorums, nous présentons ici un autre algorithme pour les accès concurrents de sessions basé sur les quorums. Cet algorithme est utilisé dans un système distribué asynchrone dans le modèle à passage de message.
Dans cette partie, nous présentons un algorithme d'exclusion mutuelle de groupe avec les quorums. Dans cet algorithme, tout ensemble fini de ressources est supporté et tout nombre de processus peuvent partager une ressource simultanément. L'idée dans cet algorithme consiste à considérer dans un groupe de processus, un noeud principal jouant le rôle de capitaine, se chargeant ainsi de gérer l'accès à une ressource. Quand nous adaptons un quorum basé sur la plan projectif fini d'ordre *2*, la complexité des messages est dans le meilleur des cas est de $O(\sqrt{n})$ et de $O(2\sqrt{n}-1)$ pour une grille où *n* est le nombre de processus dans le réseau.

3.2.1 Modèle du problème

Nous considérons un système distribué constitué d'un ensemble *n* de processus $V = \{P_1, P_2, \cdots, P_n\}$ et un ensemble de canaux de communications $E = V \times V$. Soit $G = \{x_1, x_2, \cdots, x_m\}$ un ensemble de sessions. Le système distribué est asynchrone, c'est-à-dire qu'il n'y a pas d'horloge globale commune ou de mémoire paratgée. Les canaux de communications sont fiables et *FIFO*. Les délais de message sont finis mais peuvent être non bornés.

Ici, nous faisons les mêmes hypothèses que celles du problème précédent. Notre contribution dans cette section a été de proposer une idée originale consistant à considérer dans un groupe de processus voulant accéder à une même ressource, un processus jouant un rôle de coordonnateur, qui va se charger de gérer l'entrée en section critique seulement avec les membres de son groupe.

3.2.2 Principe de l'algorithme

L'algorithme que nous allons présenter ici s'inspire de celui de Maekawa [Mae85].

L'algorithme de Maekawa implémente l'exclusion mutuelle en utilisant une *coterie* qui satisfait les propriétés mentionnées ci-dessus. Les horloges logiques de Lamport [Lam78b]

sont utilisées pour assigner une estampille à chaque requête pour la *section critique*. Une requête avec une plus petite estampille a une plus haute *priorité* qu'une requête avec une estampille plus grande.

D'autre part, le délai de synchronisation dans le meilleur des cas et le temps d'attente sont tous les deux un saut de deux messages. Quand on analyse le délai de synchronisation d'un algorithme basé sur le quorum dérivé de l'agorithme de Maekawa, nous ignorons le délai encouru dû aux résolutions de blocages et nous analysons seulement le délai de synchronisation dans le meilleur des cas. Ceci est conforme à la pratique utilisée par les autres chercheurs [Mae85, San87].

Dans cette section, nous proposons un algorithme distribué pour l'exclusion mutuelle de groupe basé sur les quorums.

Cet algorithme utilise une approche du même type que Maekawa [Lam86, San87, Sin85] : envoyer une requête pour avoir les permissions de quelques processus afin d'entrer en section critique, et libérer les permissions de quelques processus pour quitter la section critique. Cet algorithme est différent de celui de l'exclusion mutuelle classique. Les requêtes pour un même *groupe* (session) peuvent être accordées simultanément dans le cas de l'exclusion mutuelle de groupe. D'une part, même si quelques processus envoient de façon continue des requêtes pour libérer une même session *x*, un processus P_i voulant accéder à la session $x'(x \neq x')$ peut être satisfait éventuellement. En outre, les blocages ne se produisent pas.

Pour éviter la famine et les phénomènes de blocage, un ordre de priorité est défini pour chaque requête par une estampille (Paragraphe 3.1.4), qui est composée de la valeur de l'horloge logique de la requête et de l'identificateur du processus [Lam78b, San87]. Les permissions données à quelques processus peuvent être préemptées par une requête de plus haute priorité.

La démarche de notre algorithme est décrite de la manière suivante. Chaque processus est dans l'un des trois états : *Wait* (requête pour l'accès à une session), *SC* (exécution de la section critique), et *NSC* (sinon).

1. Un processus P_i voulant accéder à une session *x* envoie une requête *REQ*() à un processus P_j dans un quorum déjà construit. Il attend un message d'accord du processus P_j dans le quorum. Ensuite, P_i accède à la session *x* avec les membres de son groupe s'ils n'ont pas demandé une autre session $x'(\neq x)$.
2. Quand le processus P_i libère une session, il envoie un message de libération *REL*() à $Pred_i$.
3. Supposons que quand une requête de P_i arrive à P_j, P_j envoie un message *OK*()

à un processus P_k. Si la priorité de P_i est plus grande que celle de P_k, c'est-à-dire $\langle P_i, H_i\rangle < \langle P_j, H_j\rangle$, P_j envoie un message d'investigation **Interrogation**() afin de préempter le message *OK*() envoyé à P_k. Si P_k est dans la file d'attente, le message *OK*() de P_j est préempté, et P_k envoie un message d'abandon **Relaxe**() à P_j. Si le processus P_k est dans l'un des deux etats, *SC* ou *NSC*, alors le message est ignoré parce que P_j a éventuellement reçu un message de libération *REL*() du processus P_k.

4. Quand un processus P_j reçoit une requête pour la session *x* d'un processus P_i, il y a plusieurs cas à étudier :
   a. aucun processus n'est satisfait par P_j : P_j envoie un message *OK*() à P_i, et la session courante devient *x*.
   b. un certain processus est satisfait par P_j : P_j est satisfait si l'égalité $x = x_j$ a lieu et qu'il n'y ait aucune requête dans la file d'attente *WS* dont la priorité est plus petite que celle du processus P_j.
   c. un certain processus est satisfait par P_j, sa priorité est plus petite que celle de P_i et $x \neq x_j$. Dans ce cas un blocage est possible et nous utilisons Maekawa [Mae85].
   d. sinon, la requête de P_i est différée et gardée dans la file d'attente *WS*.
5. Quand un processus P_j reçoit un message *REL*() d'un processus P_i pour la session *x*, s'il n'y a plus de processus accordés par P_i alors une requête pour une autre session peut être satisfaite.

3.2.2.1 Variables de l'algorithme

Les variables utilisées dans l'algorithme pour un processus P_i sont données.
status : indique si un processus P_i est dans l'état *Wait*, *SC*, ou *NSC*. Initialement, *status=NSC*.
H_i *:* une estampille pour la requête courante. Initialement $H_i = 0$.
x_i *:* maintient le numéro de la session courante.
$g(P_i)$ *:* désigne le groupe courant qui contient P_i.
$Pred_i$ *:* donne l'autorisation d'accès à la session.
nOK : le nombre de messages *OK*() reçus par un processus.
Nrel : désigne le nombre de message de libération *REL*() reçu par un processus.
WS : un ensemble de requêtes en attente. Initialement *WS* $= \emptyset$.
OKset : un ensemble de pairs de processus ayant reçu des message *OK*() et leurs estampilles. Initialement $OKset = \emptyset$.
Head() : retourne le premier noeud de de la file d'attente.
REQ(P_i, x, H_i) : message envoyé par un processus P_i participer à une session avant d'entrer en section critique.
OK() : message d'accord à un processus.
REQ(P_i, x, H_i) : signifie que le processus P_i avec l'estampille H_i a fermé la session *x*.

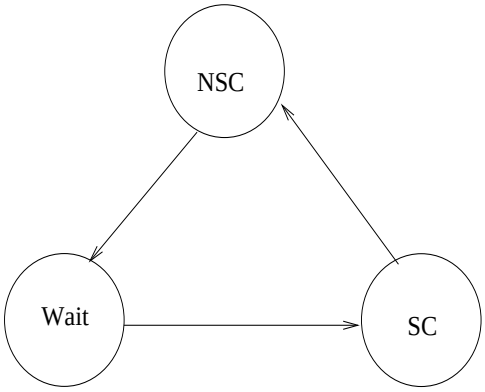

FIG. 3.4 – Etats des processus.

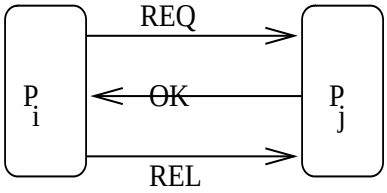

FIG. 3.5 – Messages échangés par les processus.

3.2.3 Description de l'algorithme

L'algorithme distribué est basé sur les règles suivantes : les règles de processus d'application et les règles de gestion de processus.

3.2.3.1 Règles de processus d'application

Algorithme 3.9 When a process P_i wants to enter **SC**

1.68 *status* $\leftarrow$ *Wait*
1.69 *group*$\leftarrow g(P_i)$
1.70 $H_i \leftarrow H_i + 1$
1.71 **Send REQ**(P_i, x, H_i) **To** all $q \in g$
1.72 *nOK*$\leftarrow |g(P_i)|-1$

Algorithme 3.10 When a process P_i receives **OK**(P_j, x, H_j)from P_j

1.73 $H_i \leftarrow \max(H_i, H_j)$
1.74 *nOk* $\leftarrow$ *nOK* $- 1$
1.75 **If** ($nOK = 0$) **Then**
1.76 *status* $\leftarrow$ **SC**
1.77 **EndIf**
1.78 **For all** $P_k \in g(P_i)$ **Send** Θ(*x*) **To** P_k.
1.79 $Nrel \leftarrow |g(P_i)| - 1$

Algorithme 3.11 When a process P_i exits **SC**

1.80 *status*$\leftarrow$ **SC**
1.81 **If** ($Pred_i =$ `Nil`) **Then**
1.82 **If** ($Nrel = 0$) **Then**
1.83 **Send Ok**() **To** *Head*(*WS*)
1.84 **EndIf**
1.85 **Else**
1.86 **Send REL**() **To** $Pred_i$
1.87 $Pred_i \leftarrow$ `Nil`
1.88 **EndIf**

3.2.3.2 Règles de gestion des processus

Algorithme 3.12 When REQ($P_i x, H_i$) is received at process P_j from P_i

1.89 **Do**
1.90 **If** ($OKset = 0$) **Then**
1.91 $OKset \leftarrow \{(P_i, x, H_i)\}$
1.92 $x_j \leftarrow x$
1.93 **Send OK**() **To** P_i
1.94 **Else**
1.95 **If** (*OKset*$\neq \emptyset$)$\wedge$ ($x = x_j$) **Then**
1.96 **If** (*WS* $\neq \emptyset$) **Then**
1.97 $(P'_k, H'_k) \leftarrow \min($*WS*$)$
1.98 **If** $($*WS* $\neq \emptyset) \wedge (P'_k, H'_k) < (P_j, H_j)$ **Then**
1.99 *WS* $\leftarrow$ *WS* $\cup \{(P_i, x, H_i)\}$
1.100 **Else**
1.101 *OKset*$\leftarrow$ *OKset*$\cup\{(P_i, x, H_i)\}$
1.102 **Send OK**() **To** P_i
1.103 **Else** /* (*OKset*$\neq \emptyset$)$\wedge$(*x*$\neq x_j$) */
1.104 $(P''_k, H''_k) \leftarrow \min($*OKset*$)$
1.105 **If** $(P''_k, H''_k) < (P_i, H_i)$ **Then**
1.106 *WS* $\leftarrow$ *WS* $\cup \{(P_i, x, H_i)\}$
1.107 **Else** /* $($*OKset* $\neq \emptyset) \wedge ($*x* $\neq x_j) \wedge (P_i, H_i) < (P''_k, H''_k)$ */
1.108 Voir Maekawa [Mae85].
1.109 **EndIf**
1.110 **EndIf**
1.111 *OKset*$\leftarrow \emptyset$
1.112 $x_j \leftarrow$ *x*
1.113 **For all** $(P_l, x_l, H_l) \in$ *WS* **s.t.** $x_l = x_j$ **do**
1.114 *OKset*$\leftarrow$ *OKset*$\cup\{(P_l, x_l, H_l)\}$
1.115 *WS*$\leftarrow$ *WS*$\backslash\{(P_l, x_l, H_l)\}$
1.116 **Send OK**() **To** P_l
1.117 **EndIf**
1.118 **EndIf**
1.119 **EndIf**
1.120 **EndDo**

Le cas $(OKset \neq \emptyset) \wedge (x \neq x_j) \wedge (P_i, H_i) < (P''_k, H''_k)$ peut provoquer un blocage. Spécialement, si un processus P_j reçoit un message *REQ*() d'un autre processus P_i après avoir envoyé un message *OK*() à P_k et la priorité de P_i est plus grande que celle de P_k, alors P_j envoie un message d'investigation **Interrogation** à P_k.

Algorithme 3.13 When a message **REL**(P_i, x, H_i) is received at process P_j from P_i

1.121 **Do**
1.122 *OKset*← *OKset*\{(P_i, x, H_i)}
1.123 **If** (*OKset* = ∅)∧(*WS* ≠ ∅) **Then**
1.124 $(P'_k, x'_k, H'_k) \leftarrow \min(WS)$
1.125 $x_j \leftarrow x'_k$
1.126 **For all** $(P_l, x_l, H_l) \in$ *WS* **s.t.** $x_l = x_j$ **do**
1.127 *OKset*← *OKset*∪{(P_l, x_l, H_l)}
1.128 *WS*← *WS*\{(P_l, x_l, H_l)}
1.129 **Send** *OK*() **To** P_l
1.130 **EndIf**
1.131 **EndDo**

3.2.4 Exemple

Considérons le plan projectif fini d'ordre 2 composé de 7 éléments, consistant en un sous ensemble *V* tel que tout sous ensemble a exactement 3 éléments. Chacun des éléments est contenu exactement dans 3 sous ensembles. Deux sous ensembles quelconques ont une intersection d'un seul élément. Les processus sont labellisés $1, 2, \cdots, n$. Les groupes sont :
$g(P_1) = \{1, 2, 3\},\ g(P_2) = \{2, 4, 6\},\ g(P_3) = \{3, 5, 6\}$
$g(P_4) = \{4, 1, 5\},\ g(P_5) = \{5, 2, 7\},\ g(P_6) = \{6, 1, 7\}$
$g(P_7) = \{7, 4, 3\}$.

Nous supposons que nous avons deux sessions x_1 et x_2. Maintenant nous illustrons notre algorithme par le scénario suivant :

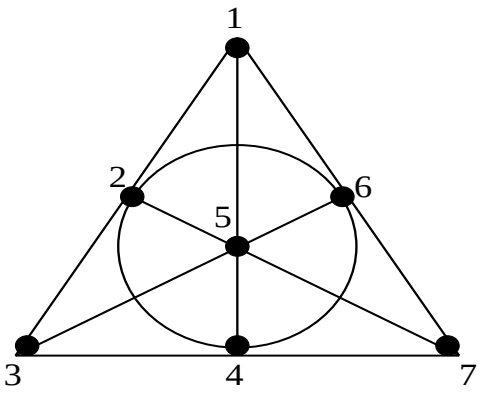

FIG. 3.6 – Le plan projectif fini d'ordre 2 (*Fano plane*).

E_1 : Les processus 1, 2 et 3 $\in g(P_1)$.
E_2 : 1 envoie un message *REQ*$(1, x_1, H_1)$ aux processus 2 et 3.
E_3 : 1 attend un message *OK*() des membres de son groupe, c'est-à-dire 2 et 3.
E_4 : Les processus 2 et 3 reçoivent un message *REQ*(P_i, x, H_i) du processus *1*.
E_5 : Si les processus 2 et 3 n'ont pas demandé à participer à une autre session x_2, ils peuvent envoyer un message *OK*() au processus 1.
E_6 : Le processus 1 a reçu tous les messages *OK*() ($nOK = |g(P_i)| - 1$) des membres de son groupe.
E_7 : Le processus 1 envoie un message $\mathbf{\Theta(x_1)}$ aux processus 2 et 3 afin d'accéder à la session x_1.

Nous pouvons aussi donner un autre exemple pour une grille de quorum de 9 éléments. Un quorum ici est l'union des éléments d'une ligne pleine et un élément de chaque ligne en dessous de la ligne pleine.

1	**2**	**3**	**4**
5	**6**	**7**	**8**
9	**10**	**11**	**12**
13	**14**	**15**	**16**

FIG. 3.7 – Une grille de quorum de 16 éléments.

3.2.5 Preuve de l'algorithme

3.2.5.1 Exclusion mutuelle

Lemme 3.1 *Soit x_i $(1 \leq i \leq m)$ une session et supposons que chaque requête est adressée à la session x_i. Alors, chaque processus voulant accéder à la session x_i entrera éventuellement en section critique.*

Preuve.

Un processus $P_k \in g(P_k)$ pour la session x_k envoie un message *OK*() à tout processus $P_i \in g(P_i)$ demandant à participer à la session x_i si $x_k = x_i$ et qu'il n'y ait pas de demande pour les autres sessions.

Lemme 3.2 *Au plus une seule session peut être ouverte à la fois.*

Preuve.

Si deux processus P_i et P_j sont en section critique *SC*, nous avons deux cas de figure :

i) P_i et P_j ont demandé la même session.

ii) P_i et P_j ont demandé deux sessions différentes x_i et x_j.

Si la priorité de P_i est plus haute que celle de P_j, nous avons :

$$\langle P_i, H_i \rangle < \langle P_j, H_j \rangle \Leftrightarrow H_i < H_j \text{ ou } (H_i = H_j) \wedge (P_i < P_j). \tag{3.3}$$

Si la priorité de P_j est plus haute que celle de P_i aussi, nous avons :

$$\langle P_j, H_j \rangle < \langle P_i, H_i \rangle \Leftrightarrow H_j < H_i \text{ ou } (H_j = H_i) \wedge (P_j < P_i). \tag{3.4}$$

Par les relations (3.3) et (3.4), nous avons $(P_i = P_j)$ et P_i et P_j sont nécessairement dans la même session.

À cause de la propriété d'intersection non vide, nous avons $g(P_i) \cap g(P_j) \neq \emptyset$ et par conséquent P_i et P_j ne peuvent pas être satisfaits simultanément. Ceci est une contradiction avec la spécification de l'algorithme permettant seulement à une session d'être ouverte en un moment.

Théorème 3.2 *L'exclusion mutuelle de groupe est satisfaite par l'algorithme proposé.*

Preuve.

La preuve de ce théorème découle directement des lemmes 3.1 et 3.2.

3.2.5.2 Absence de famine

La famine pour un processus P_i a lieu quand d'autres requêtes précédentes sont continuellement en attente d'une permission d'un membre de *g(P_i)*. Supposons que la famine existe. Considérons un processus $P_i \in$ *g(P_i)* ayant la plus haute priorité de sa requête qui ne peut jamais entrer en section critique. Avec notre hypothèse de supposition, P_i a une plus haute priorité de requête qui ne puisse pas entrer en section critique en un temps H_i. Soit *H* le temps durant lequel la requête de P_i arrive aux membres de *g(P_i)*. Considérons l'état du système après le temps $T = \max(H, H_i)$. Chaque processus $P_k \in$ *g(P_i)* $(k \neq i)$ doit essayer d'envoyer un message *OK*() à P_i parce que P_i a la plus haute priorité. Si P_k n'a pas envoyé un message *OK*() à aucun processus, évidemment il enverra un message *OK*() à P_i. Autrement, P_k peut envoyer un message *OK*() à un autre processus, appelons le P_l. Si P_l n'est pas entré en section critique, P_k pourra envoyer un message *OK*() à P_i. Si P_l est déjà en section critique, il en sortira éventuellement.
Après *T*, P_k n'envoie pas de message *OK*() à aucun des autres processus.
Par conséquent, P_k peut envoyer un message *OK*() au processus P_i et on a une absence de famine.

3.2.6 Résumé du chapitre

Dans ce chapitre, un algorithme distribué basé sur le quorum pour le problème de l'exclusion mutuelle de groupe a été présenté. Cet algorithme appartient à la catégorie des algorithmes répartis fondés sur les permissions qui échangent des messages pour déterminer quel est le prochain processus ou groupe de processus qui pourra exécuter la section critique. Les systèmes de quorums sont des techniques bien connues, conçues pour augmenter la performance des systèmes répartis, comme pour réduire le coût d'accès par opération, pour équilibrer la charge, et pour améliorer la scalabilité du système. Cet algorithme est une modification de l'algorithme de Maekawa [Mae85]. Dans notre algorithme, nous avons fait la différence entre les processus et les sessions contrairement à Joung [Jou01b]. Cet algorithme est substanciellement plus simple que celui de Joung [Jou01b]. Notons aussi que le nombre de processus pouvant entrer en section critique n'est pas limité. L'algorithme proposé a une complexité de $(\sqrt{n} + |Q|)$, où *n* est le nombre de processus dans le réseau et $|Q|$ la taille du quorum. Il faut aussi dire que cet algorithme réduit le nombre de messages par rapport à l'algorithme Maekawa_M donné par Joung dans [Jou01b]. Dans l'algorithme, un processus peut entrer en section critique seulement avec les membres de son groupe. Cet algorithme est optimal en terme de nombre de messages utilisés pour l'exclusion mutuelle de groupe.
Nous avons en outre présenté un autre algorithme d'exclusion mutuelle de groupe basé sur les quorums. Nous avons proposé un algorithme pour les accès concurrents de sessions basé sur les quorums. Dans cet algorithme, tout ensemble fini de ressources est supporté et tout

groupe de processus peuvent partager une ressource simultanément. C'est aussi une modification de l'algorithme de Maekawa [Mae85]. L'idée dans cet algorithme a été de considérer un noeud principal jouant le rôle de coordonnateur, se chargeant ainsi de gérer l'accès d'un groupe de processus à une ressource. Une session sera ouverte seulement si elle reçoit l'autorisation de toutes les autres sessions de son groupe.

CHAPITRE 4

Exclusion mutuelle de groupe basée sur le modèle client-serveur

DAns ce chapitre, nous présentons un algorithme d'exclusion mutuelle de groupe basée sur le modèle client-serveur. Nous présentons un algorithme pour le problème de l'exclusion mutuelle de groupe fonctionnant sur une topologie en arbre. L'algorithme assure qu'à tout moment, au plus une seule session est ouverte, et toute session demandée sera ouverte au bout d'un temps fini. Le nombre de messages est entre `0` et *m*, où *m* est le nombre de sessions dans le réseau. $O(log(m))$ messages sont nécessaires pour l'ouverture d'une session. Le degré maximum de concurrence est *n*, où *n* est le nombre total de processus dans le réseau.

4.1 Modèle Client-Serveur

4.1.1 Définition

La notion de *client-serveur* est fondamentale pour comprendre le fonctionnement des systèmes d'exploitation modernes. Bien qu'elle relève, en toute rigueur, d'un cours sur les communications, il est difficile de ne pas l'aborder indépendamment tant elle intervient aujourd'hui dans le fonctionnement des ordinateurs même en mode non connecté, sans aucune liaison à un réseau.

Pour comprendre le schéma du modèle *client-serveur* on se limitera à imaginer deux processus qui s'exécutent sur une ou deux machines différentes. Ces processus peuvent communiquer entre eux au travers d'interfaces logiciels spécifiques que l'on appelle ports ou

sockets. Ces interfaces sont analogues aux structures employées dans les ordres de lecture et d'écriture et mises en oeuvre par une déclaration d'ouverture de fichier fopen en langage C. La différence fondamentale est que ces deux interfaces de communication ne résident pas nécessairement sur la même machine et peuvent se trouver sur deux ordinateurs différents à des milliers de kilomètres. La façon dont la liaison est établie entre ces deux interfaces n'entre pas dans notre propos. Cela relève d'un cours sur les communications. En toute rigueur on peut imaginer un mécanisme *client-serveur* qui fonctionnerait en employant un autre moyen de communication que les sockets. Des pipes, par exemple, pourraient convenir. La seule restriction serait alors que le *client* et le *serveur* ne pourraient résider que sur la même machine, ce qui est trop restrictif pour la généralité de ce mécanisme.

4.1.2 Modèle client-serveur

Le modèle de base pour la structuration des systèmes répartis met en jeu un processus *client*, qui demande l'exécution d'un service, et un processus *serveur*, qui réalise ce service. *Client* et *Serveur* sont localisés sur deux machines reliées par un réseau de communication. Le modèle client-serveur garantit la protection mutuelle du client et du serveur par la séparation de leurs espaces d'adressage. Il permet également de localiser dans un serveur une fonction partagée par plusieurs clients.

Nous avons supposé que le serveur peut exécuter plusieurs types de services, identifiés par un nom service_id différent (par exemple, pour un serveur de fichier : lire_fichier, écrire_fichier, imprimer_catalogue, etc.). Lorsque le serveur peut servir plusieurs clients, il est intéressant de l'organiser comme une famille de processus coopérant pour permettre l'exécution concurrente de plusieurs requêtes et exploiter ainsi un multi-processus ou des entrées-sorties simultanées. Le schéma classique comporte un processus cyclique, le veilleur (deamon), qui attend les demandes des clients. Lorsqu'une demande arrive, le veilleur active un processus exécutant qui réalise le travail demandé. Les exécutants peuvent être créés à l'avance et constituer un pool fixe, ou être créés à la demande par le veilleur. ce dernier schéma est illustré ci-après :

Algorithme 4.1 Schéma classique du modèle client-serveur

Processus veilleur

1.132 **while** `True` **do**
1.133 **Begin**
1.134 $receive(client, message)$
1.135 $extract(message, service\_id, < params >)$
1.136 $p \leftarrow create\_processus(client, service\_id, < params >)$
1.137 **End**

Processus exécutant

1.138 processus *p*
1.139 **Begin**
1.140 **Do**
1.141 service [service_id] (<params>, results)
1.142 **Send** (client, results)
1.143 **EndDo**
1.144 exit
1.145 autodestruction
1.146 **End**

Notons qu'il n'y a pas identité entre la notion de serveur et la notion de site. Plusieurs serveurs peuvent coexister sur un même site. Un serveur peut être réparti sur plusieurs sites, pour augmenter sa disponibilité ou ses performances.

4.2 Préliminaires

Le problème de l'exclusion mutuelle de groupe a été introduit par Joung [Jou01a] sous le nom du *problème des philosophes parlant d'une même voix*. Dans [Jou01a] un autre type d'exclusion mutuelle appelée *exclusion mutuelle de groupe* (*GME*) est présenté. Dans le problème de GME, chaque accès à une section critique est associé à un groupe. Les sections critiques qui appartiennent à un même groupe peuvent être exécutées simultanément. Cependant les sections critiques de groupes différents doivent être exécutées dans un ordre séquentiel avec le principe de l'exclusion mutuelle. Un exemple pour le problème *GME*, décrit dans [Jou98], est le *CD-ROM* juke-box partagé par plusieurs processus comme on l'a vu dans le chapitre consacré à l'état de l"art. Tout nombre de processus peut simultanément accéder au *CD-ROM* en cours d'utilisation, mais les processus désirant accéder à un autre

CD-ROM doivent attendre. Dans ce cas les sessions sont les *CD-ROM* dans le juke-box. Joung [Jou01b] a introduit le concept de m-groupes système de quorums. Plusieurs solutions pour le problème *GME* ont été proposées sans partage de mémoire et d'horloge globale, où les processus communiquent en échangeant des messages. J. Beauquier et al. [BCDP03] ont présenté de nouvelles solutions pour *GME* : deux solutions basées sur l'arbre couvrant statique (*static spanning tree*). Dans [AM05] la notion de sorrogate-quorum a été utilisé pour résoudre le problème *GME*, et requiert une complexité basse en terme de messages, un délai de synchronisation minimum faible et un degré de concurrence élevé. La compléxité en messages est $O(q)$ par demande d'accès en section critique, où q est la taille maximale d'un quorum. Le degré maximum de concurrence est n, ce qui implique qu'il est possible pour tous les processus appartenant au même groupe d'exécuter leurs sections critiques simultanément. Dans ce chapitre, nous proposons une nouvelle solution distribuée pour le problème *GME* basée sur le chemin inverse (reverse path). Le problème est de concevoir une solution pour les algorithmes distribués de *GME* satisfaisant les propriétés suivantes :

- **Exclusion mutuelle (Surêté)** : A tout moment, deux sessions de différents groupes ne peuvent être ouvertes simultanément.
- **Absence de blocage (Vivacité)** : Une session désirant être ouverte le sera au bout d'un temps fini.
- **Entrée concurrente (Efficacité)** : Si un groupe g de processus demande une session et aucun autre groupe de processus n'est intéressé par une session différente, alors les processus du groupe g peuvent accéder en section critique de façon concurrente [Jou98, KM01, Had01].

4.3 Exclusion mutuelle distribuée

4.3.1 Modèle de système distribué

Un système distribué est organisé comme un ensemble de processus ou noeuds qui s'exécutent sur des sites reliés par un réseau de communication et qui communiquent par envoi de message. Sur chaque site, il est possible de définir un état local, qui est modifié par l'exécution des processus du site. Les messages font un temps fini mais arbitraire pour atteindre les processus qui les reçoivent. L'ordre des messages à travers les liens de communication fiable du réseau sont *FIFO*. L'algorithme présenté dans ce chapitre partage dans sa conception certaines conditions pour l'environnement du système distribué. A chaque processus dans le système distribué, il lui est associé une identification unique d'un entier naturel. Il y a exactement un seul processus en demande d'exécution sur chaque noeud. Les processus compétissent pour une seule ressource. A tout moment, chaque processus initie au plus une

requête sortante pour l'exclusion mutuelle.

4.3.2 Algorithme proposé

Notre contribution a été ici de proposer une solution distribuée pour l'exclusion mutuelle de groupe basée sur le modèle client-serveur.

Nous avons utilisé les mêmes principes pour résoudre le problème de l'exclusion mutuelle de groupe dans un système distribué. Un processus peut plusieurs fois entrer en section critique, tandis qu'une seule session peut être ouverte à la fois. La racine de l'arbre (leader) contrôle un ou plusieurs sessions, le jeton est contrôlé par la racine.

Quand les processus désirent entrer en section critique de manière concurrente, la sélection ne peut être retardée indéfiniment. Un processus demandant à entrer en section critique ne peut être empêché par un autre au bout d'un délai fini.
Avec un graphe connecté, nous construisons un arbre couvrant. Chaque noeud *y* appartient à l'arbre couvrant (*spanning tree*). L'arbre couvrant est utilisé afin de minimiser le nombre de messages échangé, et élimine les messages dupliqués. Initialement, un jeton est associé à un noeud.
Quand un noeud *x* demande à entrer en section critique, deux cas sont possibles :

1. Le noeud *x* a le jeton, dans ce cas, il entre immédiatement en *section critique*, sans envoyer un message de requête.
2. Le noeud *x* ne possède pas le jeton, il envoie une requête à son successeur dans l'arbre couvrant, et attend le jeton.

Quand un noeud *y* reçoit un message requête envoyé par un noeud *x*, plusieurs cas sont possibles :

1. Le noeud *y* possède le jeton et ne l'utilise pas, alors dans ce cas il l'envoie immédiatement au noeud *x*.
2. Le noeud *y* est en section critique alors dans ce cas il garde le message dans sa file d'attente.
3. Dans les autres cas, le noeud *y* envoie la requête à ses successeurs dans l'arbre couvrant sauf le noeud pour lequel il a reçu la demande.

4.3.3 Principe de l'algorithme

Initialement, toutes les sessions dans le réseau sont connectées de façon logique à un arbre enraciné (*rooted tree*) à une session (*Leader*). Nous supposons que la session racine possède le jeton.

Quand une session *X* reçoit une requête afin d'ouvrir la session *X* d'un processus P_i, plusieurs cas sont possibles : *i)* la session a le jeton et il n'existe aucun autre processus dans la file d'attente, dans ce cas, la session *X* envoie immédiatement une autorisation à P_i. Sinon, *ii)* s'il existe une autre session dans la file d'attente, la requête du processus P_i est ajoutée à la file d'attente. Maintenant, nous examinons la situation où une session *X* reçoit une requête d'un processus P_i et ne possède pas le jeton. Dans ce cas, *X* envoie immédiatement un message requête au *Leader* afin d'obtenir le jeton de lui. La session *X* attend le *jeton*. Quand une session *X* reçoit une requête d'obtention du *jeton* d'une autre session *Y*. La session *X* envoie un message à tous les processus qui sont dans la file d'attente, et attend d'eux tous les messages de libération.

Une fois que tous les messages de libération sont reçus par la session *X*, il envoie le jeton à la prochaine session. Quand une session *X* reçoit le *jeton* d'une autre session *Y*, il envoie un message d'autorisation aux processus se trouvant dans la file d'attente. Chaque processus *P* se comporte comme un client. En effet, quand le processus *P* veut participer à la session *X*, il envoie un message *OPEN* à la session *X* et attend son accord.
Une fois que sa participation à la session *X* est finie, il envoie un message de libération *REL* à la session *X*.

4.3.4 Messages de l'algorithme

Nous considérons deux types de messages : les messages échangés entre les sessions et les messages échangés entre les processus et les sessions.

1. **Messages échangés entre les sessions**
   REQ(*x*) : message envoyé pour obtenir le *jeton*, ce message est envoyé au *Leader*.
   TOKEN(*x*) : message désignant la permission d'ouvrir la session *x*.
2. **Messages échangés entre les processus et les sessions**
   OPEN(*x*) : requête envoyée par un processus pour ouvrir la session *x*.
   OK(*x*) : autorisation au processus P de participer à la session *x*.
   REL(*x*) : message envoyé par un processus *P* à la session *x*, signifiant que le processus *P* a fermé la session *x*.

4.3.5 Variables locales à la session *x*

Les variables locales utilisées pour la session *x* sont :
$Leader_x$: pointeur désigant un chemin d'une session à la session racine de l'arbre. Initialement, $Leader_x =$ `Nil` si *x* est la racine, et $Leader_x \neq$ `Nil` sinon.
$Next_x$: pointeur qui indique la prochaine session à partir de laquelle le *jeton* sera transmis. Initialement $Next_x =$ `Nil`.

HT_x : booléen, `True` si la session *x* possède le *jeton*, `False` sinon. Initialement, $HT_x =$ `True` si la session *x* est la racine, sinon, $HT_x =$ `False`.
RS_x : groupe de processus en attente de participer à la session *x*. Initialement, $RS_x = \emptyset$ pour toute session *x*.
$Nrel_x$: représente le nombre de message de libération que la session *x* attend de son groupe. Initialement, $Nrel_x = 0$ pour toute session *x*.

4.3.6 Variables locales à chaque processus *P*

Les variables locales utilisées pour un processus *P* sont :
X : où $X = \{x, y, z, \cdots\}$ est un ensemble dynamique de *m* sessions dans le réseau.
$Open\_Session$ *:* ensemble booléen égal à `True` si une session demandée par le processus P_i est ouverte, `False` sinon. Initialement, $Open\_Session =$ `False` pour tous les processus.
CS_i : représente la session courante demandée par le processus P_i. Initialement, CS_i=`Nil`.

4.3.7 Description de l'algorithme

L'algorithme distribué est basé sur les règles suivantes :

4.3.7.1 Règles sur les processus

Algorithme 4.2 When a process P_i wants to open a session *x*

1.147 **Do**
1.148 $CS_i \leftarrow x$
1.149 **Send** OPEN(P_i) **To** *x*
1.150 **EndDo**

Algorithme 4.3 When a process receives **OK**(*x*)

1.151 **Do**
1.152 $Open\_Session \leftarrow$ `True`
1.153 **EndDo**

Algorithme 4.4 When a process P_i releases session *x*

1.154 **Do**
1.155 **Send** *REL*(P_i) **To** *x*
1.156 $CS_i \leftarrow$ `Nil`
1.157 $Open\_Session \leftarrow$ `False`
1.158 **EndDo**

Pour ouvrir une session *x*, chaque processus P_i doit exécuter les étapes suivantes : algorithme 4.2-**Wait**(*Open_Session*)-algorithme 4.4.

4.3.7.2 Règles sur les sessions

Algorithme 4.5 When session *x* receives *OPEN*(P_i)

1.159 **Do**
1.160 **If** ((HT_x) $\wedge$ ($Next_x =$ `Nil`)) **Then**
1.161 **Send** *OK*(*x*) **To** $Leader_x$
1.162 $Leader_x \leftarrow$ `Nil`
1.163 **Else**
1.164 **If** (($Next_x =$ `Nil`) $\wedge$ ($RS_x = \emptyset$))
1.165 **Send** *REQ*() **To** $Leader_x$
1.166 $Leader_x \leftarrow$ `Nil`
1.167 **EndIf**
1.168 $RS_x \leftarrow RS_x \cup \{P_i\}$
1.169 **EndIf**
1.170 **EndDo**

Algorithme 4.6 When session *x* receives *REQ*(*y*)

1.171 **Do**
1.172 **If** ($Leader_x = \texttt{Nil}$) **Then**
1.173 **If** ((HT_x) $\wedge$ ($Nrel_x = 0$)) **Then**
1.174 **Send** *TOKEN*() **To** *y*
1.175 $HT_x \leftarrow \texttt{False}$
1.176 **Else**
1.177 $Next_x =\leftarrow y$
1.178 **EndIf**
1.179 **Else Send** *REQ*(*y*) to $Leader_x$
1.180 **EndIf**
1.181 $Leader_x \leftarrow y$
1.182 **EndDo**

Algorithme 4.7 When session *x* receives *REL*(P_i)

1.183 **Do**
1.184 $Nrel_x \leftarrow Nrel_x - 1$
1.185 **If** (($Nrel_x = 0$) $\wedge$ ($Next_x \neq \texttt{Nil}$)) **Then**
1.186 **Send** *TOKEN*() **To** $Next_x$
1.187 $HT_x \leftarrow \texttt{False}$
1.188 $Next_x \leftarrow \texttt{Nil}$
1.189 **If** ($RS_x \neq \emptyset$) **Then**
1.190 **Send** *REQ*(x) **To** $Leader_x$
1.191 $Leader_x \leftarrow \texttt{Nil}$
1.192 **EndIf**
1.193 **EndIf**
1.194 **EndDo**

Algorithme 4.8 When session *x* receives *TOKEN*()

1.195 **Do**
1.196 $HT_x \leftarrow$ `True`
1.197 **For all** $P_i \in RS_x$ **Send** *OK*() **To** P_i
1.198 **Send** *TOKEN*() **To** $Next_x$
1.199 $Nrel_x \leftarrow |RS_x|$
1.200 $RS_x \leftarrow \emptyset$
1.201 **EndDo**

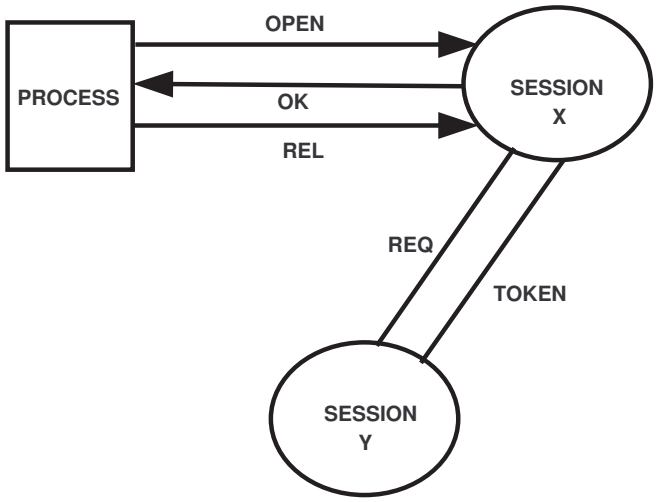

FIG. 4.1 – Messages échangés entre les processus et les sessions

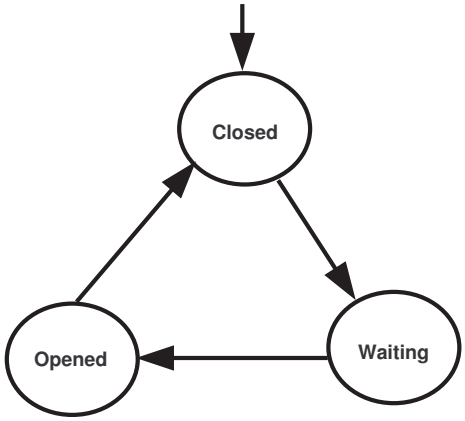

FIG. 4.2 – Etats des Sessions

Initialement, nous construisons un arbre enraciné (*rooted tree*) du réseau, où la racine est une session possédant le *jeton* et appelé *Leader*.

Un processus envoie directement sa requête à une session et attend une autorisation. Chaque session manage un groupe de processus demandeur.
Et un processus ouvre seulement une seule session à la fois si elle est la racine d'un arbre d'un réseau donné, et manage toutes les sessions. Quand un processus *p* veut ouvrir une session *k*, plusieurs cas sont possibles :

1. la session *k* est ouverte, dans ce cas le processus *p* peut accéder immédiatement à la section critique *SC*.
2. la session *k* est fermée :
   a. *x* est la racine, si toutes les sessions sont fermées, *x* ouvre la session *k*.
   b. *x* n'est pas la racine, il envoie une requête à la racine et attend le *jeton* (voir figures 4.3 et 4.4).

4.4 Exemple

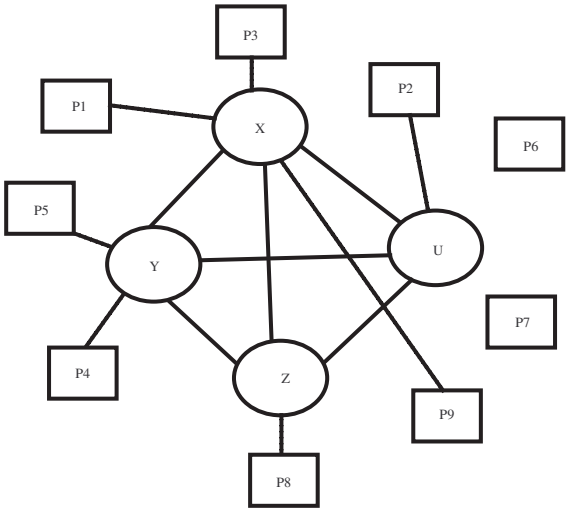

FIG. 4.3 – Graphe des sessions et processus.

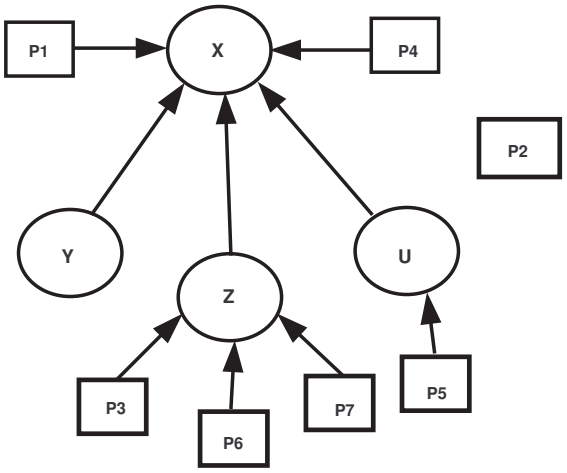

FIG. 4.4 – Arbre enraciné initial

Initialement, l'état global du système distribué est donné par :

Variable	*u*	*x*	*y*	*z*
Leader	*x*	*Nil*	*x*	*x*
Next	*Nil*	*Nil*	*Nil*	*Nil*
HT	*False*	*True*	*False*	*False*
RS	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
Nrel	0	0	0	0

TAB. 4.1 – État initial du système

Maintenant, nous allons illustrer l'algorithme par le scénario suivant :

E_1 : Les processus P_1 et P_4 veulent participer à la session *x*, et envoient un message *OPEN*() à la session *x*.

E_2 : Le processus P_5 veut ouvrir la session *u*, et envoie un message *OPEN*() à la session *u*.

E_3 : Les processus P_3, P_6 et P_7 veulent participer à la session *z* ; ils envoient un message *OPEN*() à la session *z*.

E_4 : La session *u* reçoit le message *OPEN*() du processus P_5, elle n'est pas leader, il envoie un message *REQ*() à la session *x*.

E_5 : La session *x* reçoit un message *OPEN*() du processus P_4, *x* envoie un message *OK*() à P_1.

E_6 : La session *z* reçoit un message *OPEN*() du processus P_6, il envoie un message *REQ*() à la session *x*.

E_7 : La session *x* reçoit un message *REQ*() de la session *z*. La session *z* devient la prochaine session à laquelle la session *x* doit envoyer le jeton.

E_8 : La session *x* reçoit un message *REQ*() de la session *u*, il le transmet au nouveau leader *z*.

E_9 : La session *z* reçoit les messages *OPEN*() des processus P_3 et P_7. Les processus P_3, P_6 et P_7 sont dans la file d'attente RS_z.

E_{10} : La session *x* reçoit un message *OPEN*() du processus P_1, la file d'attente RS_x contient maintenant le processus P_1.

E_{11} : La sesion *z* reçoit la requête de la session *u* venant de la session P_1.

L'état global du système distribué est donné par le tableau suivant :

Variable	*u*	*x*	*y*	*z*
Leader	*Nil*	*u*	*x*	*u*
Next	*Nil*	*z*	*Nil*	*u*
HT	*False*	*True*	*False*	*False*
RS	$\{P_5\}$	$\{P_4\}$	$\emptyset$	$\{P_3, P_6, P_7\}$
Nrel	0	1	0	0

TAB. 4.2 – État global du système

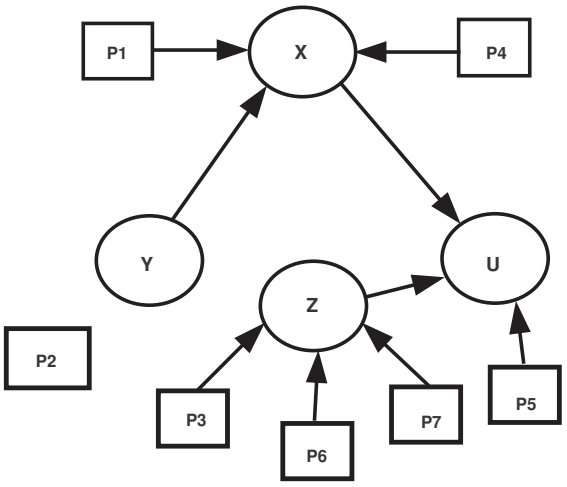

FIG. 4.5 – Nouvel arbre enraciné.

4.5 Preuve

Soient *x*, *y* et *z* les sessions dans le réseau.

Lemme 4.1 *Quelque soient les sessions x et y,* $(HT_x \wedge HT_y) =$ `False` *est un invariant.*

Preuve.

Initialement la proposition est *vraie*. Quand une session *x* transmet le *jeton* à une autre session demandeur *y*, elle donne à sa variable logique HT_x la valeur *faux* (algorithmes 4.5 et 4.6) dans l'algorithme.
Quand le jeton est en possession d'une session demandeuse, nous avons $HT_x =$ `False` pour toute session *x* dans le réseau. Lors de la réception du *jeton*, la session *x* donne à la variable HT_y la valeur *vraie*.

Théorème 4.1 *L'algorithme assure l'exclusion mutuelle (au plus une seule session est ouverte).*

Preuve.

À tout moment durant l'exécution de l'algorithme, un seul *jeton* est possédé par une session ou en transit entre deux sessions dans le réseau (Lemme 4.1). Donc, l'exclusion mutuelle est toujours garantie par le protocole.

Lemme 4.2 *A tout moment, si nous traversons le long d'une quelconque session x à la chaîne de variable pointeur $Leader_x$, alors nous atteindrons une session y qui est la racine de l'arbre.*

Théorème 4.2 *L'algorithme est exempt de famine.*

Preuve.

La famine survient quand une session doit attendre indéfiniment son ouverture même bien que quand les sessions sont ouvertes.

Considérons une session *x*, et examinons la réception d'un message *OPEN*() d'un processus *P*. Si la session *x* est le leader ($Leader_x = \texttt{Nil}$), elle attend le *jeton*. Sinon, la requête de la session *x* est transmise, par arcs correspondants au leader, à la session *y* telle que $Leader_y = \texttt{Nil}$. Si *y* désire entrer en *section critique*, la session x devient le successeur de la session *y* ; sinon, *y* envoie immédiatement le *jeton* à la session *x*.
Regardons les deux cas suivants :

1. La requête de la session *x* est transmise à la session *y* telle que $Leader_y = \texttt{Nil}$ dans un délai fini. Cela est réalisé par le lemme 4.2 dans lequel nous utilisons le fait qu'il n'y ait pas de circuits (Lemme 4.1).
2. Si la session *y* a demandé la section critique, la session *x* devient son successeur, et ce fait permettra à la session *x* de posséder le jeton au bout d'un délai fini. Ceci est prouvé par le lemme 4.7 et nous utiliserons le fichier de structures de données (lemme 4.4 et lemme 4.5).

Lemme 4.3 *Les propriétés suivantes sont satisfaites :*

1. *Le tracé Leader constitue un ensemble d'arbres enracinés (une forêt).*
2. *L'ensemble des arbres enracinés est réduit à un arbre enraciné s'il n'y a pas de messages requête en transit entre deux sessions.*

Preuve.

Initialement 1. et 2. sont satisfaites.
Supposons maintenant qu'elles sont vraies à un instant. Nous considérons que la session *x* désire entrer en *section critique* et supposons l'utilisation de l'algorithme 4.2. Si *x* est un leader, il n'y a pas de changement fait sur l'ensemble des arbres enracinés ; sinon un arbre enraciné est déconnecté du reste. Le nombre d'arbres enracinés est incrémenté de `1`. Examinons la réception d'une requête.
Quand une requête est reçue par une session leader *y*, la nouvelle valeur du pointeur leader devient *x*. La session *y* est enlevée de l'arbre dans lequel elle était et est reliée à l'arbre de la session demandeuse. Nous avons une nouvelle forêt. Le nombre d'arbres enracinés reste inchangé.
Quand une session leader *y* reçoit une requête, *y* est connecté à la session *x*. Le nombre d'arbres enracinés est décrémenté de `1`.

Lemme 4.4 *Un message de requête est transmis à une session pour laquelle* $Leader = \texttt{Nil}$ *au bout d'un temps fini.*

Preuve.

Supposons que la session *x* demande à entrer en *section critique* sans la possession du jeton. Une requête est dont envoyée de la session *x* au leader de l'arbre.

Considérons un instant durant la transmission de cette requête, quand il est en transit entre les sessions *y* et *z*. L'arc de *y* à *z* a été supprimé et la forêt est partitionnée en deux parties : une partie *A* dans laquelle le message arrive, et une partie *B* durant laquelle le message part. Aucune autre requête ne peut passer entre *A* et *B* car aucun chemin ne peut être crée avant que la requête soit arrivée à la session *z*.
Quand la requête arrive à *z*, si la session *z* n'est pas le leader, la requête est envoyé de la session *z* à la session *y*. La partie *A* est augmentée et la partie *B* est diminuée et il existe toujours une coupe entre *A* et *B*.

Donc, la requête ne peut jamais atteindre encore une session de *A*. Nous avons prouvé qu'une requête ne peut être reçue deux fois par une même session ; c'est-à-dire que le nombre de sessions par lesquelles la requête passe est inférieure à *n*.
Sinon, les délais de transmission sont finis. Nous avons prouvé que la requête atteindrait le leader au bout d'un délai fini.

Lemme 4.5 $((Leader_x = Nil) \Rightarrow (Next_x = Nil))$ *est un invariant.*

Preuve.

Initialement *vrai*, et reste *vrai* pour toutes les actions de l'algorithme.

Lemme 4.6 $(Leader_x = Nil) \wedge (RS_x = \emptyset) \Rightarrow (HT_x = True)$.

Preuve.

Initialement *vrai*, seule la session racine possède le *jeton*, et pour laquelle $(Leader_x = \texttt{Nil})$. Une session perd le *jeton* quand elle reçoit une requête d'une autre session, dans ce cas $(Leader_x \neq \texttt{Nil})$.

Lemme 4.7 $(Next_x \neq \texttt{Nil}) \Leftrightarrow (Leader_x \neq \texttt{Nil}) \wedge (RS_x \neq \emptyset)$.

Preuve.

Initialement *vrai* et reste *vrai* pour toutes les actions de l'algorithme.

4.6 Performances

La performance de l'algorithme d'exclusion mutuelle distribué peut être évaluée en terme du nombre de métriques. La complexité de messages et le délai de synchronisation sont deux paramètres qui peuvent être utilisés pour comparer avec la performance d'algorithmes d'exclusion mutuelle variés.

La complexité de messages d'un algorithme d'exclusion mutuelle distribué est le nombre de messages échangé par un processus par accès à la *section critique*.
Le délai de synchronisation est le délai moyen accordant la *section critique*.

La tolérance de fautes d'un algorithme d'exclusion mutuelle distribuée est le nombre maximal de noeuds pouvant tomber en panne avant qu'il ne soit impossible pour un noeud d'entrer en *section critique*.
La disponibilité est la probabilité que la section critique peut entrer en présence de panne. En fait, la disponibilité d'un algorithme d'exclusion mutuelle distribuée est une mesure de la tolérance de fautes.

Lemme 4.8 *Le nombre de requêtes envoyé par une session demandeur est borné par (m-1) où m est le nombre de sessions dans un réseau donné.*

Preuve.

Quand une session *x* possède le *jeton*, il n'envoie aucune requête ; sinon, la session *x* envoie la requête à la racine courante, et attend le *jeton*. Avec le lemme 4.4, aucune session ne reçoit la même requête deux fois.
Soit *h* la hauteur de la session *x* dans l'arbre enraciné. Nous avons $0 \leq h \leq (m-1)$.

Lemme 4.9 *Le nombre de messages nécessaire à la transmission du jeton d'une session x à une autre est 1.*

Preuve.

Le *jeton* est transmis directement d'une session à une autre.

Théorème 4.3 *L'algorithme requiert m messages par accès à la section crtitique dans le pire des cas.*

Preuve.

Par les lemmes 4.5 et 4.7, une topologie complète où chaque noeud *x* a (*n*-1) voisins et le rayon est égal à 1. Le nombre de requêtes est (*n*-1). Dans un arbre maximal chaque requête est envoyée exactement une fois à chaque noeud, et le nombre de messages est égal à (*n*-1).

4.7 Résumé du chapitre

Dans ce chapitre, nous avons présenté un algorithme distribué d' *exclusion mutuelle de groupe basée sur le modèle Client/Server* et qui fonctionne sur une topologie en arbre. Cet algorithme est basé sur une circulation de jeton contrairement à l'algorithme d'exclusion mutuelle de groupe que nous avons présenté dans le chapitre précédent et qui appartient à la classe des algorithmes basés sur les permissions. L'approche de la circulation de jeton repose sur la fiabilité de la communication. Cet algorithme a été présenté dans le but de réduire le nombre de messages échangés pour une entrée en section critique. Comme avec les deux algorithmes proposés pour le problème de l'exclusion mutuelle de groupe basés sur les quorums, ici aussi on fait une différence entre les sessions et les processus. L'algorithme assure qu'à tout moment une seule session est ouverte, et toute session demandée sera ouverte au bout d'un temps fini. Le nombre de messages nécessaire pour la satisfation de chaque requête se situe entre 0 et *m* dans le meilleur et pire des cas respectivement, où *m* est le nombre de sessions dans le système distribué. $O(log(m))$ messages sont nécessaires pour l'ouverture d'une session. Le degré maximum de concurrence est *n*, où *n* est le nombre total de processus dans le réseau. Si certains processus sont intéressés à une participation d'une session ouverte, et aucun autre processus n'est intéressé par une différente session, alors le processus peut être présent à la session de façon concurrente.

CHAPITRE 5

Exclusion mutuelle de groupe dans les réseaux mobiles ad hoc

DAns ce chapitre, nous présentons un algorithme d'exclusion mutuelle de groupe pour les réseaux mobiles ad hoc. C'est une adaptation de l'algorithme *RL* (Reverse Link) utilisé dans [WWV98].

5.1 Introduction

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

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

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

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

Dans un réseau ad hoc les hôtes mobiles doivent former, d'une manière ad hoc, une sorte d'architecture globale qui peut être utilisée comme infrastrcuture du système. Les applications des réseaux ad hoc sont nombreuses, on cite l'exemple classique de leur applications de secours et les missions d'expérience.

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

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

5.2 Les environnements mobiles

Un environnement mobile est un système composé de sites mobiles et qui permet à ses utilisateurs, d'accéder à l'information indépendemment de leurs positions géographiques.

Le modèle de système intégrant des sites mobiles et qui a tendance à se généraliser, est

composé de deux ensembles d'entités distinctes : les *sites fixes* d'un réseau de communication filaire classique (wired network), et les *sites mobiles* (wireless network) [IB94]. Certains sites fixes, appelés stations support mobile (Mobile Support Station) ou *station de base* (*SB*) sont munis d'une interface de communication sans fil pour la communication directe avec les sites ou unités mobiles (*UM*), localisés dans une zone géographique limitée, appelée *cellule* (voir figure 5.1).

À chaque station de base correspond une cellule à partir de laquelle des unités mobiles peuvent émettre et recevoir des messages. Alors que les sites fixes sont interconnectés entre eux à travers un réseau de communication filaire, généralement fiable et d'un débit élevé. Les liaisons sans fil ont une bande passante limitée qui réduit sévèrement le volume des informations échangées [DR92].

Dans ce modèle, une unité mobile ne peut être, à un instant donné, directement connectée qu'à une seule station de base. Elle peut communiquer avec les autres sites à travers la station à laquelle elle est directement rattachée. L'autonomie réduite de sa source d'énergie, lui occasionne de fréquentes déconnexions du réseau ; sa reconnexion peut alors se faire dans un environnement nouveau voire dans une nouvelle localisation.

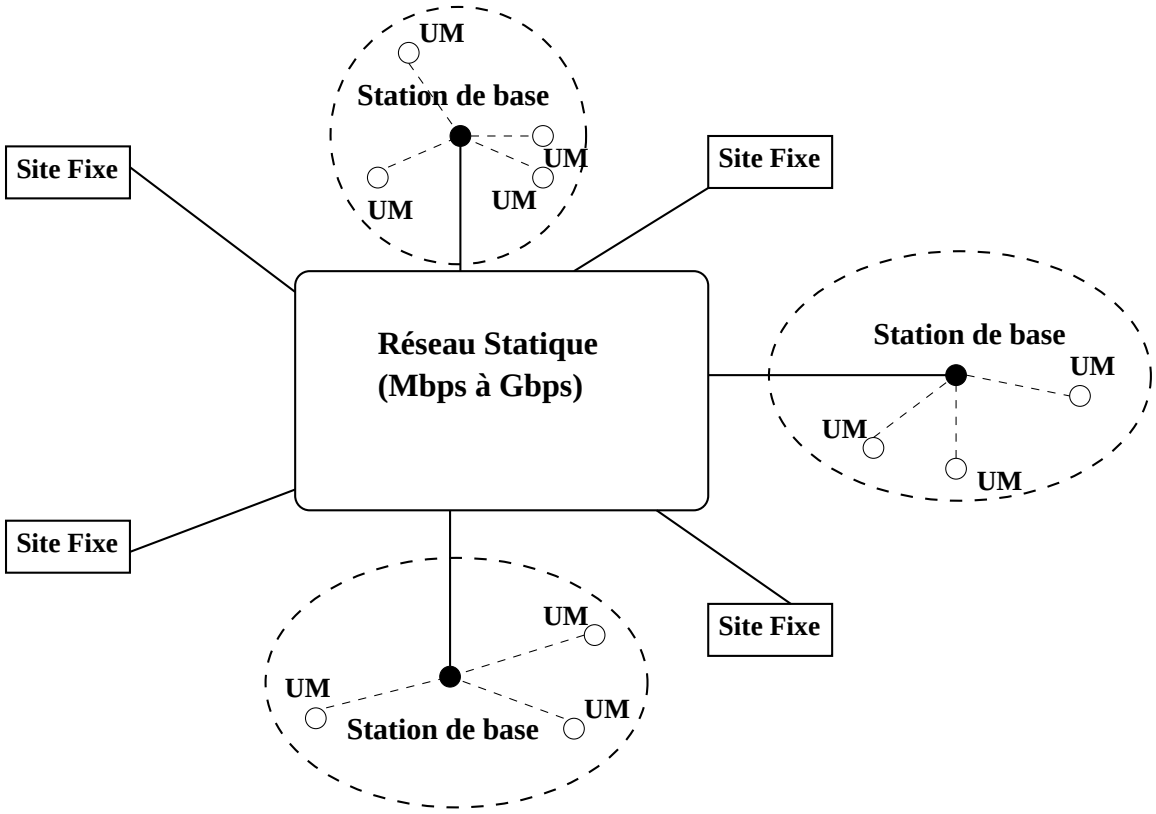

FIG. 5.1 – Le modèle des réseaux mobiles avec infrastructure

Le modèle de réseau sans infrastructure préexistante ne comporte pas l'entité *site fixe*, tous les sites du réseau sont mobiles et se communiquent d'une manière directe en utilisant leurs interfaces de communication sans fil (voir figure 5.2). L'absence de l'infrastructure ou du réseau filaire des stations de base, oblige les untiés mobiles à se comporter comme des routeurs qui participent à la découverte et la maintenance des chemins pour les autres hôtes du réseau.

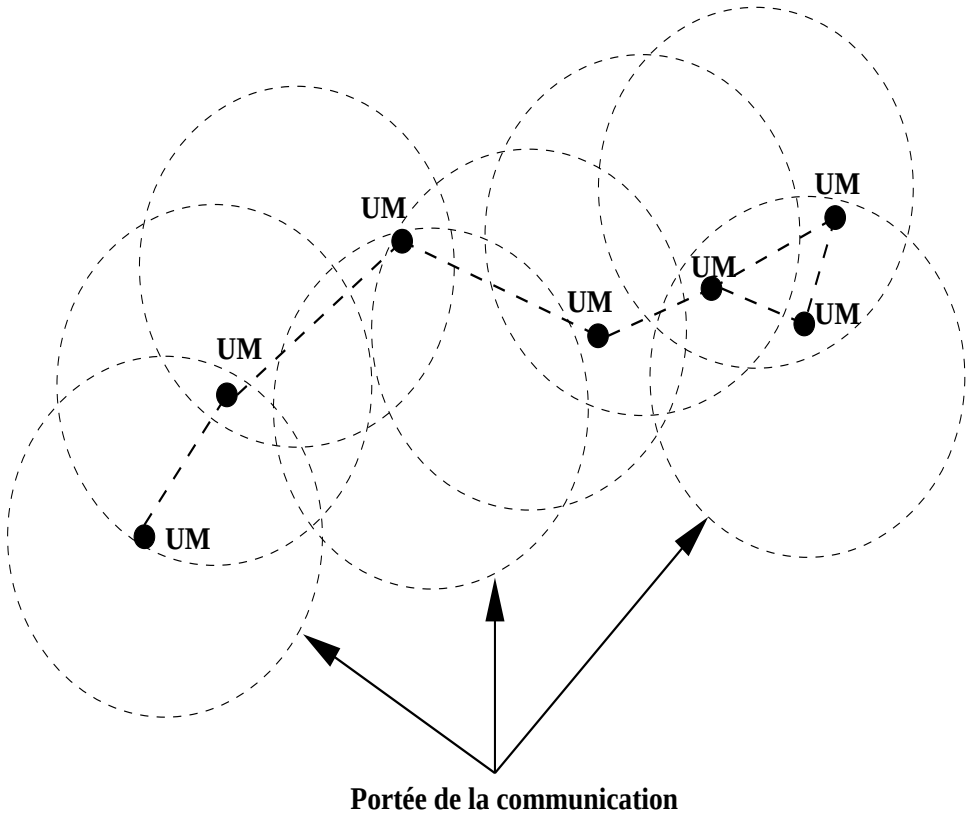

FIG. 5.2 – Le modèle des réseaux mobiles sans infrastructure

La mobilité et la portabilité offertes par les environnements mobiles, permettront le développement de nouvelles classes d'application : services d'informations avec accès à diverses bases de données en tout lieu et à tout temps (pages jaunes, distribution, spectacles, etc.) et des applications dites verticales relevant de domaines spécifiques : compagnie de location, localisation d'employés dans une entreprise [Nad98], etc.

La messagerie électronique connaîtra un développement spectaculaire ; les usagers munis de communicateurs pourront envoyer et recevoir des messages de n'importe où et les nouvelles électroniques leur seront délivrées en fonction de leurs profils respectifs [IB92]. La permanence de la connexion des usagers aux réseaux d'information, indépendemment de leurs positions géographiques contribuera au développement des applications coopératives

[DBCF92, DBCF92, DBC+93, FZ94].

5.2.1 Caractéristiques physiques des unités mobiles

Il est à prévoir que l'émergence d'un marché massif du calcul mobile, que la plupart des auteurs situent autour de la fin de cette décénie, sera basée sur des applications orientées vers des services d'information et de messagerie, et verra le développement de diverses configurations d'unités mobiles plus ou moins évoluées.

Les configurations existantes, bien que diverses, se décomposent essentiellement en deux classes : les ordinateurs de poche (*palmtops*, avec une féquence d'horloge qui oscille entre *8* et *20 Mhz*, une *RAM* de *1* Moctets et une *ROM* de *512* Koctets à *2* Moctets, sont généralement comparables à celle d'un ordinateur personnel (*PC*) de bureau avec une capacité mémoire de *2* à *8* Moctets et une fréquence d'horloge de *15* à *20 Mhz* [PB93].

5.2.2 La fiabilité de la communication sans fil

La communication sans fil est moins fiable que la communication dans les réseaux filaires. La propagation du signal subit des perturbations (erreurs de transfert, micro-coupure, timeout) dues à l'environnements, qui altèrent l'information transférée. Il s'ensuit alors, un accroissement du délai de transit de messages à cause de l'augmentation du nombre de retransmissions. La connexion peut aussi être rompue ou altérée par la mobilité des sites.

Un usager peut sortir de la zone de réception ou entrer dans une zone de haute interférence. Le nombre d'unités mobiles dans une même cellule (dans le cas des réseaux cellulaires), par exemple lors d'un rassemblement populaire, peut entraîner une surcharge du réseau.

L'une aussi des limites de la communication sans fil vient de la relative faiblesse de la bande passante des technologies utilisées. On distingue les réseaux utilisant l'infrarouge avec un débit de *1Mbps*, la communication radio avec *2 Mbps* et le téléphone cellulaire avec *9* à *14 Kbps*. La bande passante est évidemment partagée entre les utilisateurs d'une même cellule. Pour augmenter la capacité de service d'un réseau, deux techniques sont utilisées : la technique de recouvrement des cellules sur différentes longueurs d'ondes et celle qui réduit la portée du signal pour avoir plus de cellules mais de rayon moindre couvrant une région donnée. Chaque cellule est généralement subdivisée en sept cellules dont le rayon r est égal au tiers de celui de la cellule de départ. Deux cellules peuvent utiliser la même fréquence *fi*, si la distance d qui les sépare est au moins égale à trois fois le rayon r de la cellule (voir figure 5.3). Cette dernière technique est généralement plus utilisée à cause de sa faible

consommation d'énergie et une meilleure qualité du signal.

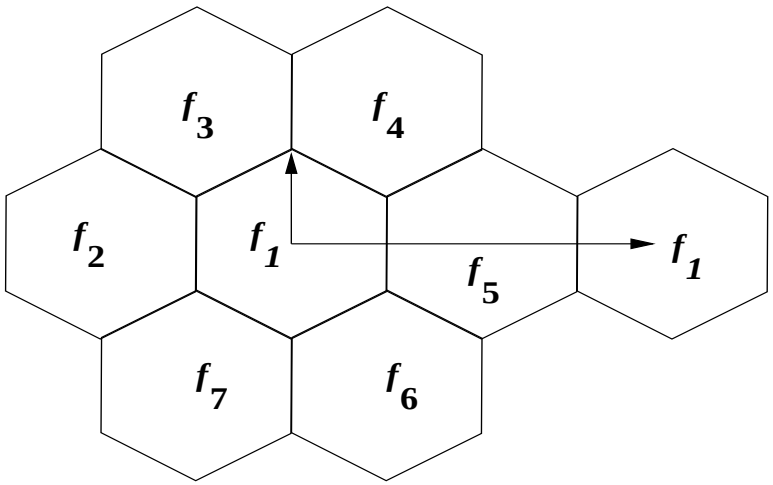

FIG. 5.3 – Le principe de réutilisation de fréquence

5.2.3 Quelques éléments de l'infrastructure sans fil

Les réseaux informationnels de demain dits *PCN* (Personal Communication Network) intégreront une large variété de services (voix, données, multimédia, etc.) offerts aux usagers indépendamment de leur position géographique. L'architecture générale de ces réseaux, bien qu'encore en débat, sera construite autour des infrastructures déjà existantes telles que :
Les réseaux téléphoniques cellulaires (à l'avenir microcellulaire) reliés au réseau téléphonique public.
Les réseaux locaux traditionnels tels Ethernet, étendus à la communication sans fil, et reliés à des réseaux plus étendus de type *LAN, WLAN, Internet*, etc.
Les architectures orientées vers des services spécialisés fournit par diffusion sur des portions d'ondes radio en modulation de fréquence ou par des satellites à des usagers munis de terminaux spéciaux [PB93, IB94].

La même unité mobile peut, en principe, interagir avec les trois types d'infrastructures à différents moments, par exemple, en se déplaçant de l'intérieur d'un bâtiment où elle interagit avec un réseau local pourvu d'une interface de communication sans fil, à l'extérieur du bâtiment où elle interagit avec le réseau téléphonique cellulaire.

5.3 Les réseaux mobiles ad-hoc

Les systèmes de communication cellulaire sont basés essentiellement sur l'utilisation des réseaux filaires (tels que *Internet* ou *ATM*) et la présence des stations de base qui couvrent

les différentes unités mobiles du système. Les réseaux mobile *ad hoc* sont à l'inverse, des réseaux qui s'organisent automatiquement de façon à être déployable rapidement, sans infrastructure fixe, et qui doivent pouvoir s'adapter aux conditions de propagation, aux trafics et aux différents mouvements pouvant intervenir au sein des noeuds mobiles.

Les réseaux mobiles présentent une architecture originale. En effet, l'atténuation des signaux avec la distance, fait que le médium peut être réutilisé simultanément en plusieurs endroits différents sans pour autant provoquer de collisions, ce phénomène est appelé *la réutilisation spatiale* (Spatial Reuse) [Nad98] et il sert de base au concept de la communication cellulaire.

Le concept des *réseaux mobile ad hoc* essaie d'étendre les notions de la mobilité à toutes les composantes de l'environnement. Ici, contrairement aux réseaux basés sur la communication cellulaire, aucune administration centralisée n'est disponible, ce sont les hôtes mobiles elles-mêmes qui forment, d'une manière *ad hoc*, une infrastructure du réseau. Aucune supposition ou limitation n'est faite sur la taille du réseau ad hoc, le réseau peut contenir des centaines ou des milliers d'unités mobiles.

Les réseaux ad hoc sont idéals pour les applications caractérisées par une absence (ou la non-fiabilité) d'une infrastructure préexistante, telle que les applications militaires et les autres applications de tactique comme les opérations de secours (incendies, tremblement de terre...) et les missions d'exploration.

Définition 5.1 *Un réseau mobile ad hoc, appelé généralement MANET (Mobile Ad hoc NETwork), consiste en une grande population, relativement dense, d'unités mobiles qui se déplacent dans un territoire quelconque et dont le seul moyen de communication est l'utilisation des interfaces sans fil, sans l'aide d'une infrastructure préexistante ou administration centralisée. Un réseau ad hoc peut être modélisé par un graphe* $G_t = (V_t, E_t)$ *où* V_t *représente l'ensemble des noeuds (c'est-à-dire les unités ou les hôtes mobiles) du réseau et* E_t *modélise l'ensemble des connections qui existent entre ces noeuds (voir figure 5.4). Si* $e = (u, v) \in E_t$*, cela veut dire que les noeuds u et v sont en mesure de communiquer directement à l'instant t.*

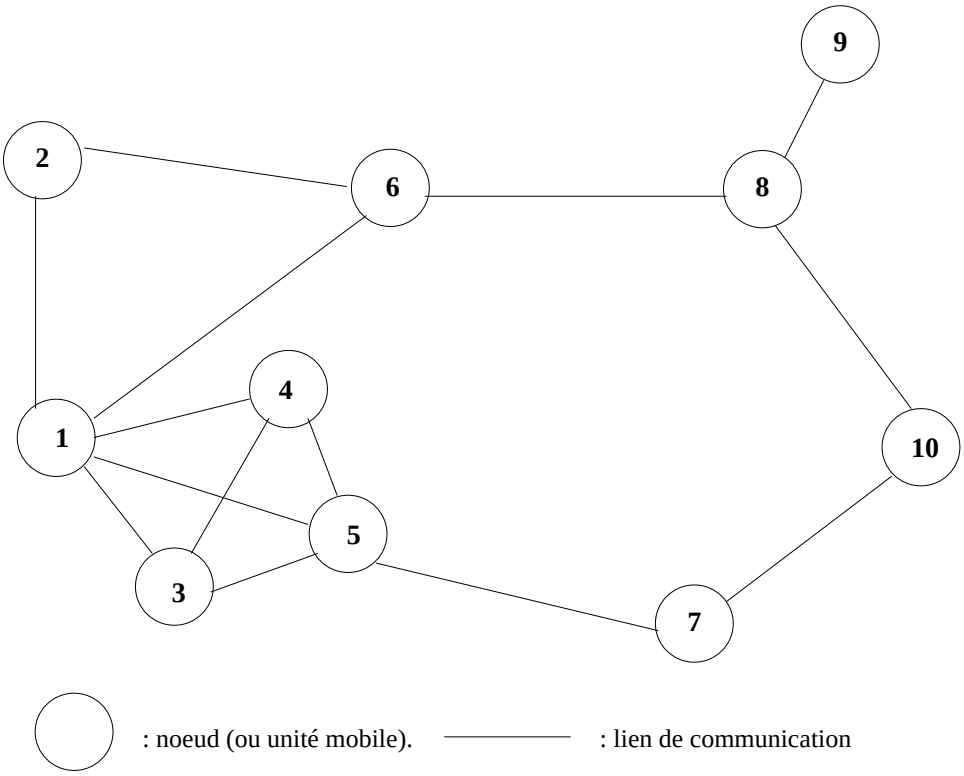

FIG. 5.4 – La modélisation d'un réseau ad hoc

La topologie du réseau peut changer à tout moment (voir figure 5.5), elle est donc dynamique et imprévisible ce qui fait que la déconnexion des unités soit très fréquente.

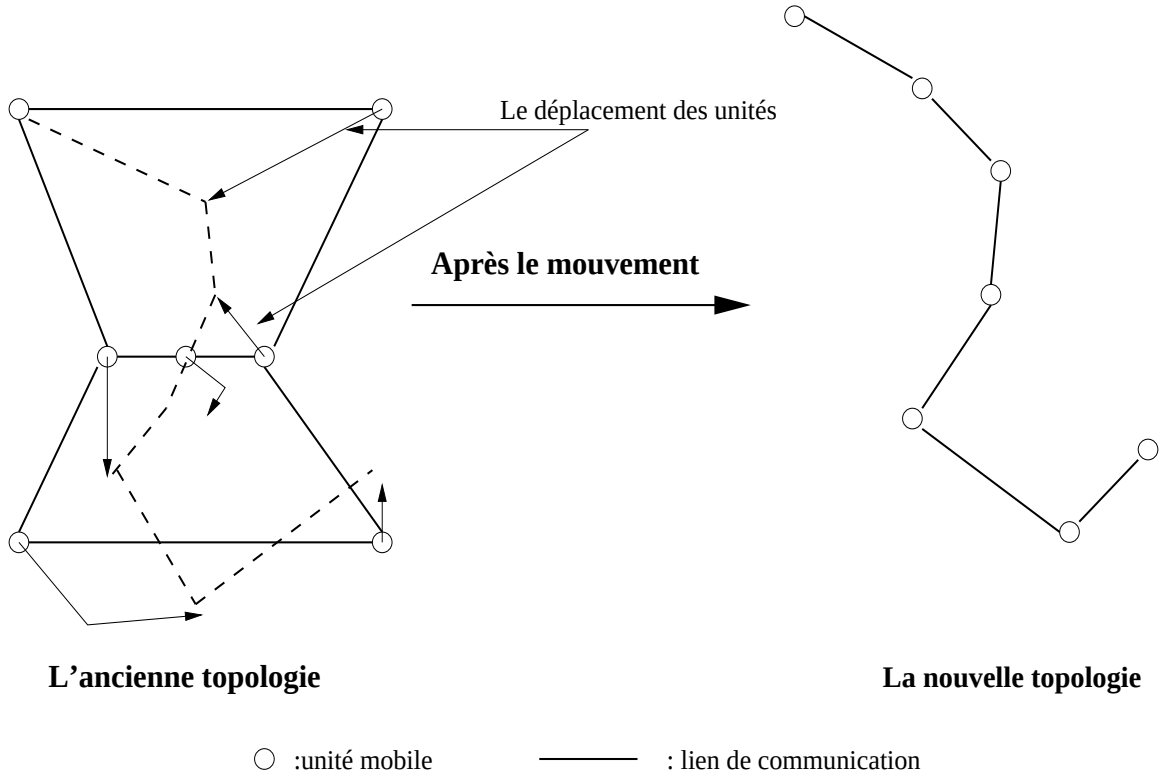

FIG. 5.5 – Le changement de la topologie des réseaux ad hoc

Un exemple d'un réseau ad hoc : un groupe d'unités portables reliés par des cartes *HIPERLAN*. Les réseaux appelés *GSM* ne représentent pas des réseaux ad hoc, car la communication entre les unités passe obligatoirement par des stations de base du réseau filaire.

5.3.1 Les applications des réseaux mobiles ad hoc

Les applications ayant recours aux réseaux ad hoc couvrent un très large spectre, incluant les applications militaires et de tactique, les bases de données parallèles, l'enseignement à distance, les systèmes de fichiers répartis, la simulation distribuée interactive et plus simplement les applications de calcul distribué ou méta-computing.

D'une façon générale, les réseaux ad hoc sont utilisés dans toute application où le déploiement d'une infrastructure réseau filaire est trop contraignant, soit parce que difficile à mettre en place, soit parce que la durée d'installation du réseau ne justifie pas de câblage demeure.

5.3.2 Les caractéristiques des réseaux ad hoc

Les réseaux mobiles ad hoc sont caractérisés par ce qui suit :

Une topologie dynamique : Les unités mobiles du réseau se déplacent d'une façon libre et arbitraire. Par conséquent la topologie du réseau peut changer à des instants imprévisibles, d'une manière rapide et aléatoire. Les liens de la topologie peuvent être unis ou bidirectionnels.

Une bande passante limitée : Une des caractéristiques primordiales des réseaux basés sur la communication sans fil est l'utilisation d'un médium de communication partagé. Ce partage fait que la bande passante réservée à un hôte soit modeste.

Des contraintes d'énergie : Les hôtes sont alimentés par des sources d'énergie autonomes comme les batteries ou les autres sources consommables. Le paramètre d'énergie doit être pris en considération dans tout contrôle fait par le système.

Une sécurité physique limitée : Les réseaux ad hoc sont plus touchés par le paramètre de sécurité, que les réseaux filaires classiques. Cela se justifie par les contraintes et limitations physiques qui font que le contrôle des données transférées doit être minimisé.

L'absence d'infrastructure : Les réseaux ad hoc se distinguent des autres réseaux mobiles par la propriété d'absence d'infrastructure préexistante et de tout genre d'administration centralisée. Les hôtes mobiles sont responsables d'établir et de maintenir la connectivité du réseau d'une manière continue.

5.4 Préliminaires

Dans [MM05], R. Mellier et J.F. Myoupo ont présenté un protocole pour l'exclusion mutuelle (MUTEX) pour les réseaux mobiles ad hoc multi-sauts basé sur sur la structure de *clusterisation* offerte par les techniques de partitionnement du réseau. Dans leur travail, les noeuds sont partitionnés en *clusters* qui est fondamental dans le cas des réseaux mobiles ad hoc. Une fois ces groupes ou clusters établis, des noeuds sont choisis afin d'agir comme des coordonnateurs du processus de clusterisation : ces noeuds sont appelés *clusterheads* ou leaders de clusters. Ainsi un cluster est formé en associant un clusterhead avec quelques noeuds qui lui sont voisins (c'est-à-dire des noueds qui sont dans sn rayon de transmission). Le choix du clusterhead est basé sur un système de poids. A chaque noeud est associé un poids qui est un nombre réel positif. Plus le poids d'un noeud est grand, plus ce noeud est meilleur dans le rôle de clusterhead.

Dans [BBBAN04], un état de l'art des algorithmes d'exclusion mutuelle sur les réseaux mobiles ad hoc a été présenté. La majorité des algorithmes d'exclusion et de k-exclusion mutuelle pour les réseaux mobiles ad hoc sont basés sur une circulation du jeton.

Dans [MM06], R. Mellier et J.F. Myoupo ont apporté une grande contribution en proposant un algorithme non basé sur une circulation du jeton et qui est tolérant aux fautes avec une option d'économie de l'énergie. Cet algorithme résout le problème de l'exclusion mutuelle et de la k-exclusion mututelle dans les réseaux moiles ad hoc single hop. Ils ont supposé dans [MM06] que les noeuds ou liens du réseau ne tombent pas en panne. Le principal avantage de leur contribution est la tolérance aux fautes et le fait que les noeuds peuvent rester anonymes durant tout l'algorithme et leur nombre n est inconnu.

Dans [WWV98], un algorithme d'exclusion mutuelle basé sur le jeton, appelé *RL* (Reverse Link), pour un réseau mobile ad hoc a été proposé. Le système contient un ensemble de n noeuds et de m ressources partagées, communiquant par un modèle à passage de messages sur un réseau sans fil. L'algorithme *RL* considère les propriétés suivantes sur les noeuds mobiles et le réseau :

1. les noeuds ont des identificateurs uniques,
2. les noeuds ne tombent pas en panne,
3. les communications de liens sont bidirectionnelles et *FIFO*,
4. un protocole de niveau de lien (link-level) assure que chaque noeud est averti de l'ensemble des noeuds avec lequel il peut directement communiquer en fournissant des indications de formations ou de pannes de liens,
5. les pannes de liens naissantes sont détectées, fournissant ainsi une communication fiable,
6. le réseau n'est pas partitionné,
7. les délais de messages obeissent à une inégalité triangulaire (c'est-à-dire que les messages qui traversent un saut seront reçus avant les messages envoyés en même temps traversant plus d'un saut).

L'algorithme *RL* suppose aussi qu'il existe un unique jeton initialement et utilise la technique d'inversion partielle dans [GB81] afin de maintenir un graphe acyclique direct (*DAG*). Dans l'algorithme *RL*, quand un noeud veut accéder à une ressource partagée, il envoie une requête le long d'un lien de communication. Chaque noeud maintient une file d'attente contenant les identificateurs des noeuds voisins à partir desquels il a reçu des requêtes pour le jeton. L'algorithme *RL* utilise une relation d'ordre total sur les noeuds de sorte que le noeud de plus petit ordre soit toujours celui qui possède le jeton. Chaque noeud choisit dynamiquement son noeud voisin de plus petit ordre comme son lien sortant du détenteur du jeton. Quand un noeud détecte une panne sur un lien sortant et qu'il n'est pas le dernier sortant,

il reroute la requête. S'il est le dernier lien sortant, s'il n'y a pas de chemin au détenteur du jeton, ainsi, il organise un réarrangement partiel du graphe acyclique direct afin de trouver une nouvelle route. Quand un nouveau lien est détecté, les deux noeuds concernés avec ce fait échangent leurs messages pour réaliser un changement nécessaire de leurs liens entrants et sortants et reroutent éventuellement leurs requêtes. Ainsi, le partiel réarrangement est appelé. Cet algorithme garantit les propriétés de sûreté et de vivacité ([WK97] pour la preuve).

Maintenant nous présentons le scénario du problème d'exclusion mutuelle de groupe. Considérons un réseau mobile ad hoc consistant en *n* noeuds et *m* ressources partagées. Chaque noeud est dans l'un des états suivant : une non section critique (*NSC*), une section d'attente (*Trying*), et une section critique (*SC*). Un noeud *i* peut accéder à une ressource partagée seulement dans la section critique. A chaque fois qu'un noeud *i* veut accéder à une ressource partagée S_i, il se déplace des états *NSC* à *Trying*, et attend d'entrer dans l'état *SC*. Les problème d'exclusion mutuelle [Jou98] est concerné par la façon de concevoir un algorithme satisfaisant les propriétés suivantes :

- **Exclusion mutuelle (Surêté)** : Si deux noeuds distincts, *i* et *j* exécutent simultanément leur section critique, alors $S_i = S_j$.
- **Absence de blocage (Vivacité)** : Si un noeud *i* est dans son état *Trying*, alors il entrera en section critique éventuellement.
- **Entrée concurrente (Efficacité)** : Si des noeuds veulent accéder à une ressource, et qu'aucun autre noeud ne veut accéder à une autre ressource différente, alors les noeuds peuvent y entrer de façon concurrente.

Notons que cette dernière propriété est une conséquence triviale de la seconde. Maintenant regardons le cas où toutes les requêtes sont pour un même noeud. La propriété d'entrée concurrente de Joung (dans de telles exécutions) est que les noeuds devraient pouvoir occuper non seulement de façon concurrente la section critique mais aussi d'y entrer sans une nécessaire synchronisation. Cela signifie que (dans de telles exécutions) les noeuds ne devraient pas retarder un autre pendant qu'ils essaient d'entrer en section critique. L'entrée concurrente assure qu'un noeud *i* essayant d'entrer en section critique ne soit pas retardé par d'autres noeuds déja entrés en section critique. Il n'empêche cependant pas au noeud *i* d'être retardé (pour un temps arbitraire) par d'autres noeuds qui tentent simultanément d'entrer en section critique.

5.5 Algorithme proposé

Un graphe acyclique direct est maintenu sur les liens physiques du réseau dans toute l'exécution d'algorithme comme résultat d'un triplet, représentant la taille des noeuds, comme

dans [GB81]. Les liens sont considérés directs des noeuds de plus grande hauteur ou taille vers les noeuds de plus petite taille, basé sur un ordre lexicographique du 3-uplet. Un lien entre deux noeuds est *sortant* au noeud de plus grande taille et *entrant* au noeud de plus petite taille. Une relation d'ordre total sur la taille des noeuds dans le réseau est assurée parce que le dernier entier du triplet est l'identificateur unique du noeud. Par exemple, si la taille du noeud 1 est (1,2,3) et la taille du noeud 2 est (2,2,2), alors le lien entre ces deux noeuds sera direct du noeud 1 au noeud 2. Intitialement au noeud 0, la taille est (0,0,0) et, pour tout $i \neq 0$, la taille de i est initialisée de sorte que les liens directs forment un graphe acyclique direct dans lequel chaque noeud non détenteur du jeton a un chemin direct au noeud possédant le jeton. Le noeud de plus petite taille est toujours celui qui possède le jeton, faisant ainsi de lui une centrale à partir de laquelle toutes les requêtes sont envoyées. Dans cette section, nous proposons un algorithme distribué pour résoudre le problème d'exclusion mutuelle de groupe pour un réseau mobile ad hoc.

Notre algorithme est une adaption de l'algorithme *RL* étudié dans [WWV98]. Notre contribution dans cette partie a été d'ajouter des variables comme par exemple *SubToken*() qui est un message permettant d'informer aux noeuds qu'ils peuvent accéder de façon concurrente à une ressource *S*. Ainsi cela nous a permis de résoudre la propriété d'entrée concurrente, condition très importe dans l'exclusion mutuelle de groupe. Notons aussi que d'autres variables supplémentaires ont été utilisées dans cet algorithme, nous permettant ainsi d'apporter une contribution au problème de l'exclusion mutuelle de groupe pour les réseaux ad hoc.

5.5.1 Structures de données

Cet algorithme est exécuté dans un système de *n* noeuds et de *m* ressources partagées. Les noeuds sont labélisés $0, 1, \cdots, n-1$ et les ressources $0, 1, \cdots, m-1$. Nous supposons en plus qu'il y a seul jeton dans le réseau initialement possédé par le noeud *0*. Les variables utilisées dans l'algorithme pour un noeud *i* sont :
status : indique si un noeud est dans l'un des états suivants : *Trying*, *SC*, ou *NSC*. Initialement, *status=NSC*.
N : l'ensemble de tous les noeuds en contact physique avec le noeud *i*. Initialement, *N* contient tous les noeuds voisins de *i*.
Num : compte le nombre de noeuds en section critique.
height : un *3*-uplet (h_1, h_2, i) représentant la taille du noeud *i*. Les liens sont considérés directs du noeud de plus grande taille vers le noeud de plus petite taille, basé sur un ordre léxicographique. Initialement au noeud *0*, $height_0$=(*0,0,0*) et, pour tout $i \neq 0$, $height_i$ est initialisé de sorte que les liens directs forment un graphe acyclique direct où chaque noeud a un chemin direct au noeud *0*.

Vect : un tableau de tuples représentant la vue de la taille du noeud *i*, $i \in N$. Initialement, *Vect*[*i*]=taille du noeud *i*. D'un point de vue du noeud *i*, le lien entre *i* et *j* est *entrant* au noeud *i* si *Vect*$[j] > height_i$, et *sortant* du noeud *i* si *Vect*$[j] < height_i$.
Leader : un drapeau mis à `True` si un noeud possède le jeton et mis à `False` autrement. Initialement, *Leader*=`True` si $i = 0$, et *Leader*=`False` sinon.
next : indique la position du jeton par rapport à la vue de *i*. Quand un noeud *i* possède le jeton, *next* $=$ *i*, sinon *next* est un noeud d'un lien *sortant*. Initialement, *next* $= 0$ si $i = 0$, et *next* est noeud voisin *sortant* sinon.
Q : une file d'attente contenant les requêtes des noeuds voisins. Les opérations dans *Q* inclusent *Enqueue*(), qui met en file d'attente un noeud seulement s'il n'est pas déjà dans *Q*, *Dequeue*() avec une sémantique usuelle *FIFO*, et *Delete*(), qui enlève un noeud de *Q*, sans se soucier de sa position. Initialement, $Q = \emptyset$.
receivedLI : tableau de booléens indiquant si le message portant la taille *LinkInfo* est reçu du noeud *j*, à partir duquel un message *Token* a été recemment envoyé. Aucune information sur la taille reçue du noeud *i* au noeud *j* pour lequel *receivedLI*[*j*] est faux ne sera gardée dans *Vect*[*j*]. Initialement, *receivedLI*[*j*]=`True` pour tout $j \in N$.
forming[*j*] : tableau de booléens mis à `True` si une formation de lien au noeud *j* a été détectée et mis à `False` si le premier message *LinkInfo* arrive du noeud *j*. Initialement, *forming*[*j*]=`False` pour tout $j \in N$.
formHeight[*j*] : un tableau gardant les valeurs des tailles du noeud *j*, $j \in N$, quand un nouveau lien à *j* est détecté d'abord. Initialement, *formHeight*[*j*]=*height* pour tout $j \in N$.

5.5.2 Messages de l'algorithme

Les messages utilisés dans l'algorithme sont données ci-dessous. Il faut noter que chaque message est attaché à la valeur *height*, notée *h*, du noeud envoyant le message :
Request(*i*,*S*) : quand un noeud *i* veut entrer en section critique *SC* pour accéder à une ressource *S*, il envoie un message *Request*() au noeud voisin indiqué par la variable *next*.
SubToken() : un message pour informer les noeuds qu'ils peuvent accéder de façon concurrente à une ressource *S*. Il peut y avoir plusieurs messages *SubToken*s dans le système simultanément.
Token() : un message pour entrer en section critique *SC*. Le noeud possédant le jeton est appelé *Leader*.
Rel() : un message du noeud *i* libérant la ressource S_i, il envoie un message *Rel*() à un des noeuds voisins.
LinkInfo() : un message utilisé par les noeuds pour échanger les valeurs de leurs tailles avec les noeuds voisins.

5.5.3 Description de l'algorithme

Comme l'algorithme *RL*, l'algorithme proposé est basé sur les événements. Un événement d'un noeud *i* consiste en une réception de message d'un autre noeud $j \neq i$, ou en une indication de formation ou de panne de lien de communication de la couche de lien. Chaque événement déclenche une procédure qui est exécutée atomiquement. Ci-dessous, nous présentons une vue d'ensemble de ces procédures d'événements :

- **Demande d'une ressource *S*** : Quand un noeud *i* veut entrer en section critique pour accéder à la ressource *S*, il met dans sa file *Q* le message *Request*() et met son état à *Trying*. Si le noeud *i* ne possède pas le jeton et qu'il a un seul élément dans sa file, il appelle la procédure *SendRequest*() afin d'envoyer une requête *Request*(*i*,*S*). Si le noeud *i* a le jeton, il enlève la requête *Request*(*i*,*S*) et définit son état à *SC*. Après il envoie le message *SubToken*() à tous ses noeuds voisins de sa file *Q*. A sa sortie de la section critique, il fait appel à la procédure *SendTokenToNext*().
- **Libération d'une ressource *S*** : Quand un noeud *i* ne possédant pas le jeton quitte la section critique pour libérer la ressource *S*, il appelle la procédure *SendRel*() pour envoyer un message *Rel*() à un de ses noeuds voisins et définit son nouveau état à *NSC*. S'il n'a pas le jeton il appelle la procédure *SendTokenToNext*().
- **Réception d'une requête** : Quand une requête *Request*(*j*,*S*) envoyée par un noeud voisin *j* est reçu par le noeud *i*, *i* ignorera cette requête si *receivedLI*[*j*] est faux. Sinon, *i* change *Vect*[*j*] et met la requête dans *Q* si le lien entre *i* et *j* est *entrant* au noeud *i*. Si *Q* est non vide et que le status est *NSC*, *i* appelle *SendTokenToNext*(). Si *i* n'a pas le jeton, il fait appel à *RaiseHeight*() si le lien à *j* est *entrant* et que *i* n'a pas de liens *sortants* ou *i* appelle la procédure *SendRequest*() si $Q = [j]$ ou si *Q* est non vide et ainsi le lien à *next* est inversé.
- **Réception d'un message *Rel*()** : Supposons que *i* possède le jeton. Quand un message *Rel*() envoyé par *j* est reçu par *i*, *i* met son statut à *NSC*. Ainsi, *i* appelle *SendTokenToNext*() pour passer le jeton. Si *i* n'a pas le jeton, ensuite quand il reçoit *Rel*(), il appelle juste *SendRel*() afin d'expédier le message de libération.
- **Réception du jeton** : Quand un noeud *i* reçoit un message *Token*() d'un certain noeud *j*, *i* donne à la valeur *Leader* vraie. Ensuite *i* diminue sa taille afin qu'elle soit plus petite que celle du dernier noeud détenteur du jeton, soit *j*, informe tous ses voisins de sa nouvelle taille en leur envoyer un message *LinkInfo*(), et appelle la procédure *SendTokenToNext*().
- **Réception d'un message *SubToken*()** : Quand un noeud *i* reçoit un message *SubToken*() d'un certain noeud *j*, il envoie ce message à tous ses voisins qui ont leurs requêtes dans la file *Q*. Si la requête de *i* pour accéder à la ressource *S* est dans *Q*, *i* peut entrer en section critique et accéder à *S*. Plus précisément si la requête de *i* est la seule dans *Q* et $S \neq R$, alors *i* envoie un message *Rel*() en faisant appel à *SendRel*().

- **Réception d'un message d'information *LinkInfo*()** : Quand un message d'information de lien *LinkInfo*() d'un noeud *j* est reçu par un noeud *i*, la taille de *j* est gardée dans *Vect*[*j*]. Si *receivedLI*[*j*] est faux, alors *i* regarde si la taille de *j* est la même que lorsqu'il lui envoyait le jeton. Si c'est le cas, *receivedLI*[*j*] devient vrai. Si *forming*[*j*] est vrai, la valeur courante de *height* est comparée à celle de *height* quand le lien à *j* a été détecté avant, *formHeight*[*j*] sont différentes, ensuite *LinkInfo*() est envoyé au noeud *j*. L'identificateur de *j* est ajouté à *N* et *forming*[*j*] est mis à faux. Si $j \in Q$ et que *j* est un noeud d'un lien *sortant*, alors *j* est enlevé de *Q*. Si le noeud *i* n'a pas de liens *sortants* et qu'il n'a pas le jeton, il fait appel à *RaiseHeight*(), ainsi un lien *sortant* sera formé. Sinon, si *Q* est non vide, et que le lien à *next* est inversé, *i* appelle *SendRequest*() puisqu'il doit envoyer une autre requête pour le jeton.
- **Panne ou échec d'un lien** : Quand un noeud *i* sent une panne de lien à un noeud voisin *j*, il enlève *j* de *N*, met *receivedLI*[*j*] à vrai, et si $j \in Q$, enlève *j* de *Q*. Ensuite, si *i* ne possède pas le jeton et n'a pas de liens *sortants*, il appelle *RaiseHeight*(). Si *i* ne possède pas le jeton, $Q \neq \emptyset$, et le lien à *next* échoue, *i* appelle *SendRequest*() puisqu'il doit envoyer une autre requête pour le jeton.
- **Formation de lien** : Quand un noeud *i* détecte un nouveau lien au noeud *j*, il envoie un message *LinkInfo*() à *j*, met *forming*[*j*] à vrai et fait *formingHeight*[*j*]=*height*.
- **Procédure *SendTokenToNext*()** : Le noeud *i* retire de la file d'attente d'abord la première requête, *Request*(*j*,*S*) de *Q* et donne à *next* la valeur *j*. Si $next = i$, *i* entre en section critique.
  Si $next \neq i$, *i* diminue *Vect*[*next*] à $(height.h_1, height.h_2 - 1, next)$, donne à *Leader* la valeur vrai, *receivedLI*[*next*] la valeur faux, et ensuite envoie un message *Token*() à *next*. Si *Q* est non vide après avoir envoyé un message à *next*, une requête est envoyée à *next* immédiatement en suivant le message *Token*() ainsi jeton sera éventuellement retourné à *i*.
- **Procédure *RaiseHeight*()** : Cette procédure est appelée à chaque fois que *i* perd son dernier lien *sortant*. Le noeud *i* augmente sa taille en utilisant une méthode partielle inversée (partial reversal method) de [GB81] et informe à tous ses voisins de son changement de taille avec des messages *LinkInfo*(). Tous les noeuds dans *Q* pour lesquels les liens sont maintenant *sortants* sont enlevés de *Q* ; Si *Q* est non vide, *SendRequest*() est appelé puisque *i* doit envoyer une autre requête pour le jeton.
- **Procédure *SendRequest*()** : Elle consiste à la sélection de *next* comme étant le noeud voisin de plus petite taille de *i* et ensuite à l'envoi d'une requête à *next*.
- **Procédure *SendRel*()** : Un noeud *i* ne possédant pas le jeton appelle *RaiseHeight*() quand il perd son dernier lien *sortant*. Après avoir appelé *RaiseHeight*(), *i* sélectionne *next* comme étant le noeud voisin de plus petite taille et envoie un message *Rel*() à *next*. La procédure *SendRel*() n'est jamais appelée par un noeud ayant le jeton.

5.5.4 Pseudocode de l'algorithme

Algorithme 5.1 When a node *i* requests access to the *SC*

1.202 *status*← *Trying*
1.203 *Enqueue*(*Q*,*i*)
1.204 **If** (not *Leader*) **Then**
1.205 **If** ($|Q| = 1$) **Then**
1.206 *SendRequest*()
1.207 **EndIf**
1.208 **Else**
1.209 **Send** *SubToken*(S_i) **to all** $v \in Q$
1.210 **EndIf**

Algorithme 5.2 When a node *i* releases *SC*

1.211 *status*← *NSC*
1.212 **If** (not *Leader*) **Then**
1.213 *SendRel*()
1.214 **Else**
1.215 **If** ($|Q| > 0$) **Then**
1.216 *SendTokenToNext*()
1.217 **EndIf**
1.218 **EndIf**

Algorithme 5.3 When *Request*(*h*) received at node *i* from node *j*

/* *h* désigne la taille du noeud *j* quand le message est envoyé */

1.219 **If** (*ReceivedLI*[*j*]) **Then**
1.220 $Vect[j] \leftarrow h$
1.221 **If** ($height < Vect$[*j*]) **Then**
1.222 *Enqueue*(*Q*,*j*)
1.223 **If** (*Leader*) **Then**
1.224 **If** ((*status* $= NSC$) $\wedge$ ($|Q| > 0$)) **Then**
1.225 *SendTokenToNext*()
1.226 **Else**
1.227 **If** (*height* $<$ *Vect*$[k], \forall k \in N$) **Then**
1.228 *RaiseHeight*()
1.229 **Else**
1.230 **If** (($Q = [j] \vee ((|Q| > 0) \wedge$ (*height* $<$ *Vect*[*next*]))) **Then**
1.231 *SendRequest*()
1.232 **EndIf**
1.233 **EndIf**
1.234 **EndIf**
1.235 **EndIf**
1.236 **EndIf**
1.237 **EndIf**

Algorithme 5.4 When *Rel*() is received at node *i* from node *j*

1.238 **If** (*leader*) **Then**
1.239 **If** *status=NSC* **Then**
1.240 *SendTokenToNext*()
1.241 **Else**
1.242 *SendRel*()
1.243 **EndIf**
1.244 **EndIf**

Algorithme 5.5 When *Token*(*h*) received at node *i* from node *j*

1.245 *Leader*← `True`
1.246 *Vect*[j]← h
1.247 *height*.$h_1 \leftarrow h.h_1$
1.248 *height*.$h_2 \leftarrow h.h_2 - 1$
1.249 **Send** *LinkInfo*($h.h_1, h.h_2, i$) **to all** $k \in N$
1.250 **If** ($|Q| > 0$) **Then**
1.251 *SendTokenToNext*()

Algorithme 5.6 When *SubToken*() received at node *i* from node *j*

1.252 **Send** *SubToken*(S_j) **to all** $v \in Q$
1.253 **If** $((i \in Q) \wedge (Resource = S_j))$ **Then**
1.254 $Num \leftarrow Num + 1$
1.255 $status \leftarrow SC$
1.256 **Else**
1.257 **If** $((Q = [i]) \wedge (Resource \neq S_j))$ **Then**
1.258 $Resource \leftarrow S_j$
1.259 $Num \leftarrow 1$
1.260 **EndIf**
1.261 **EndIf**

/* la variable *Resource* est l'identificateur de la ressource courante */

Algorithme 5.7 When *LinkInfo*(*h*) received at node *i* from node *j*

1.262 $N \leftarrow N \cup \{j\}$
1.263 **If** ((*forming*$[j]$) $\wedge$ (*height* $\neq$ *formHeight*$[j]$)) **Then**
1.264 **Send** *LinkInfo*(*height*) **to** *j*
1.265 *forming*[*j*]$\leftarrow$ `False`
1.266 **If** (*receivedLI*[*j*]) **Then**
1.267 *Vect*[j]$\leftarrow h$
1.268 **Else**
1.269 **If** (*Vect*$[j] = h$) **Then**
1.270 *receivedLI*$[j] =$ `True`
1.271 **If** ((*j* $\in$ *Q*) and (*height* $>$ *Vect*$[j]$)) **Then**
1.272 *Delete*(*Q*,*j*)
1.273 **If** (*Leader*) **Then**
1.274 **If** ((*height* $<$ *Vect*$[k], \forall k \in N) \wedge$ (not *Leader*)) **Then**
1.275 *RaiseHeight*()
1.276 **Else**
1.277 **If** ((|Q| > 0) $\wedge$ (*height* $<$ *Vect*[*next*])) **Then**
1.278 *SendRequest*()
1.279 **EndIf**
1.280 **EndIf**

Algorithme 5.8 When failure of link to *j* is detected at node *i*

1.281 $N \leftarrow N - \{j\}$
1.282 $receivedLI[j] \leftarrow$ `True`
1.283 **If** ($j \in Q$) **Then**
1.284 *Delete*(*Q*,*j*)
1.285 **If** (not *Leader*) **Then**
1.286 **If** ($height < Vect[j], \forall k \in N$) **Then**
1.287 *RaiseHeight*()
1.288 **Else**
1.289 **If** (($|Q| > 0$) $\wedge$ ($next \notin N$)) **Then**
1.290 *SendRequest*()
1.291 **EndIf**
1.292 **EndIf**
1.293 **EndIf**
1.294 **EndIf**

Algorithme 5.9 When formation of link to *j* detected to node *i*

1.295 **Send** *LinkInfo*(*height*) **to** *j*
1.296 *forming*[*j*]$\leftarrow$ `True`
1.297 *formHeight*[*j*]$\leftarrow$*height*

Algorithme 5.10 Procedure *SendRequest*()

1.298 $next \leftarrow l \in N : Vect[l] \leq Vect[j]\ \forall j \in N$
1.299 **Send** *Request*(*height*) **to** *next*

Algorithme 5.11 Procedure *SendTokenToNext*()

1.300 *next*← *Dequeue*(*Q*)
1.301 **If** (*next* $\neq i$) **Then**
1.302 *Leader*← `False`
1.303 *Vect*[*next*]← (*height*.h_1, *height*.$h_2 - 1$, *next*)
1.304 *receivedLI*[*next*]← `False`
1.305 **Send** *Token*(*height*) **to** *next*
1.306 **EndIf**
1.307 **If** ($|Q| > 0$) **Then**
1.308 **Send** *Request*(*height*) **to** *next*
1.309 **Else**
1.310 *status* $\rightarrow SC$
1.311 **Send** $|Q|$ *SubTokens* **to all** $v \in N \cap Q$
1.312 **EndIf**

Algorithme 5.12 Procedure *RaiseHeight*()

1.313 *height*.$h_1 \leftarrow 1 + \min_{k \in N}\{$*Vect*$[k].h_1\}$
1.314 $S \leftarrow \{l \in N :$ *Vect*$[l].h_1 =$ *height*$.h_1\}$
1.315 **If** ($S \neq \emptyset$) **Then**
1.316 **Send** *linkInfo*(*height*) **to all** $k \in N$
1.317 **For all** $k \in N$ **such that** *height* $>$ *Vect*$[k]$) **do** *Delete*(*Q*,*k*)
1.318 **EndIf**
1.319 **If** ($|Q| > 0$) **Then**
1.320 **EndIf**

Algorithme 5.13 Procedure *SendRel*()

```
1.321  If (not Leader) Then
1.322      If (height < Vect[k], ∀k ∈ N) Then
1.323          RaiseHeight()
1.324          next← l ∈ N : Vect[l] ≤ Vect[j] ∀j ∈ N
1.325          Send Rel() to next
1.326      EndIf
1.327  EndIf
```

5.6 Preuve de l'algorithme

Dans cette section, nous prouvons que l'algorithme proposé satisfait l'exclusion mutuelle, l'absence de famine et l'entrée concurrente. Nous prouvons d'abord que l'algorithme proposé satisfait la propriété d'exclusion mutuelle avec le théorème 5.1.

Théorème 5.1 *L'algorithme assure la propriété d'exclusion mutuelle.*

Preuve.

Quand un noeud possède le jeton, il peut entrer en section critique et ensuite il envoie des message *SubToken*s aux noeuds voisins demandeurs. Quand un noeud reçoit un message *SubToken*(), il peut entrer en section critique s'il demande la même ressource que celle du leader c'est-à-dire le noeud possédant le jeton. Puisqu'il n'existe qu'un seul jeton dans le réseau, tous les noeuds en section critique doivent accéder à la même ressource. Ainsi, la propriété d'exclusion mutuelle est garantie.

Ci-dessous, nous prouvons que l'algorithme proposé satisfait la propriété d'entrée concurrente dans le théorème 5.2.

Théorème 5.2 *L'algorithme proposé assure la propriété d'entrée concurrente.*

Preuve.

Quand un noeud possède le jeton, il peut entrer en section critique et envoyer ensuite des messages *SubToken*s aux noeuds voisins qui le souhaitent. Quand un noeud reçoit un message *SubToken*(), il peut entrer en section critique s'il demande à la même ressource que celle du leader c'est-à-dire le noeud possédant le jeton. En résumé, tous les noeuds peuvent accéder à une même ressource comme le détenteur du jeton. Ainsi, la propriété d'entrée

concurrente est garantie.

Ci-dessous, nous prouvons que la propriété d'absence de famine en montrant qu'un noeud demandeur possédera le jeton éventuellement. Puisque les valeurs des tailles des noeuds sont totalement ordonnées, le graphe logique dont les arcs sont supposés avoir la direction du noeud de plus grande taille au noeud de plus petite ne contient pas de cycle, et ainsi il est un graphe acyclique direct. Nous voulons prouver que le graphe acyclique direct est orienté selon le jeton, c'est-à-dire que pour chaque noeud *i*, il existe un chemin direct dont l'origine est le noeud *i* et la destination est le noeud qui possède le jeton. Nous présentons le lemme 5.2, qui est le véritable lemme 4 de [WWV98].

Lemme 5.1 *A chaque état de chaque exécution, le graphe acyclique direct est orienté selon le jeton si et seulement si il n'y a pas de noeuds médiateurs (sink en anglais).*

Preuve.

La condition de suffisance est triviale par la définition même d'un graphe acyclique direct orienté selon le jeton. La condition de nécessité est prouvée par contradiction. Supposons le contraire c'est-à-dire qu'il existe un noeud *i* dans un certain état tel que $Leader_i = \texttt{False}$ et pour lequel il n'existe pas de chemin direct dont l'origine est *i* et la destination est le noeud détenteur de jeton. Puisqu'il n'y a pas de noeuds médiateurs, *i* avoir un plus un lien *sortant* qui est *entrant* d'un autre noeud. Puisque i) le nombre de noeuds est fini, ii) le réseau est connecté, et iii) toutes les arêtes sont logiquement dirigées telles que aucun chemin logique ne puisse former un cycle, il doit exister un chemin direct de *i* au noeud ayant le jeton, ceci est une contradiction.

Lemme 5.2 *Si les changements de liens cessent, le graphe logique dont les arcs sont supposés avoir la direction du noeud de plus grande taille au noeud de plus petite, est un graphe acyclique direct orientée selon le jeton.*

Sur la base du lemme 5.2, nous pouvons prouver qu'un noeud demandant le jeton, le possédera éventuellement.

Théorème 5.3 *Si les changements cessent, alors un noeud demandant le jeton, le possédera éventuellement.*

Preuve.

Quand un noeud *i* possèdant le jeton est dans l'état *NSC*, il passe le jeton au noeud *j* dont la requête est à la tête de la file. Le noeud *i* enlève la requête de *j* de la file après avoir passé le jeton. Ainsi, chaque requête de noeud sera éventuellement à la tête de la file pour avoir l'opportunité d'avoir le jeton. Par le lemme 5.2, chaque requête d'un noeud a un chemin menant au noeud possédant le jeton. Donc, un noeud demandeur possédera le jeton éventuellement.

5.7 Résumé du chapitre

Dans ce chapitre nous avons proposé un algorithme basé sur le jeton pour résoudre le problème d'exclusion mutuelle de groupe dans un réseau mobile. Cet algorithme est adapté à la mobilité des noeuds. Cet algorithme combine plusieurs idées venant de différents articles. La technique d'inversion partielle [GB81], utilisée pour maintenir un graphe acyclique direct orienté à l'arrivée dans le réseau quand la destinaton est statique, est utilisée dans cet algorithme afin de maintenir un graphe acyclique direct orienté selon le jeton avec une destination dynamique. Comme dans les algorithmes [Ray89a, DK94], chaque noeud dans l'algorithme maintient une queue de requêtes contenant les identificateurs des noeuds voisins pour lesquels il a reçu des requêtes pour le jeton. Comme dans [DK94], les noeuds sont totallement ordonnés. Nous avons supposé en outre que le réseau n'est pas partitionné pour une certaine simplicité. L'algorithme proposé est adapté de l'algorithme *RL* dans [WWV98]. Nous avons prouvé que cet algorithme satisfait les propriétés d'exclusion mutuelle, d'absence de famine et d'entrée concurrente. Cet algorithme est sensible aux formations et coupures de liens et est ainsi approprié pour les réseaux mobiles ad hoc.

CONCLUSION

LES systèmes distribués, ou répartis, tiennent une place de plus en plus importante dans le monde actuel, en particulier avec l'arrivée d'Internet. Informellement, ils représentent une abstraction dans laquelle un ensemble d'entités coopèrent entre elles afin d'effectuer une tâche ou d'offir un service donné.

Les travaux menés dans cette thèse concernent l'allocation de ressources dans les groupes. Nous avons présenté quelques algorithmes pour la résolution de ce problème. Ce problème a été présenté pour la première fois par Joung [Jou98]. Ce problème permet à un ensemble de processus demandant la même ressource d'accéder à cette ressource de façon concurrente, le but étant d'assurer l'exclusion mutuelle entre les ressources. Les seules solutions existantes dans les systèmes distribués étaient ceux fondées sur l'horloge de Lamport [Lam78b]. La mise en oeuvre de ce mécanisme interdit que la taille des messages (et par conséquent, la mémoire des processus) soit bornée.

Dans la première partie de la thèse, nous avons présenté le contexte général de notre travail de thèse.

Dans le premier chapitre, un état de l'art des problèmes d'allocation de ressources dans le cadre des systèmes distribués a été présenté. Le problème de l'exclusion mutuelle de groupe a aussi été présenté dans ce chapitre ainsi que des utilisations et un état de l'art.

Dans le chapitre deux, nous avons présenté les définitions formelles des structures de quorums, incluant les ensembles de quorums, coteries et bicoteries. Ensuite, quelques propriétés qui appartiennent à des travaux précédents ont été présentées.

Dans ce contexte, nous avons présenté dans le troisième chapitre un algorithme pour le

	Maekawa	$Maekawa\_M$	$Maekawa\_S$	Algorithme proposé
Nombre de messages	$6 \times \sqrt{\frac{2n(m-1)}{m}}$	$O(n \times \min(n, m))$	$O(\sqrt{n})$	$(\sqrt{n} + \|Q\|)$

TAB. 5.1 – Nombre de messages des algorithmes *GME* basés sur les quorums

problème de l'exclusion mutuelle de groupe basée sur les quorums. Il est de la catégorie des algorithmes répartis fondés sur les permissions. Cette solution est basée sur une modification de l'algorithme de Maekawa [Mae85] et ici on a mis une différence entre les processus et les sessions. Dans cet algorithme, nous avons mis une différence entre les processus et les sessions. On a utilisé les systèmes ordinaires de quorums pour l'exclusion mutuelle du fait que les quorums pour des processus de même groupe n'ont pas besoin d'avoir une intersection. Cette généralisation permet aux processus de sélectionner des quorums simultanément, et ainsi leur permet d'entrer en section critique de façon concurrente. Notons aussi que le nombre de processus pouvant entrer en section critique n'est pas limité. L'algorithme proposé a une complexité de $(\sqrt{n} + |Q|)$, où *n* est le nombre de processus dans le réseau et $|Q|$ la taille du quorum. Il faut aussi dire que cet algorithme réduit le nombre de messages par rapport à l'algorithme Maekawa_M donné par Joung dans [Jou01b]. Dans l'algorithme, un processus peut entrer en section critique seulement avec les membres de son groupe. Cet algorithme est optimal en terme de nombre de messages utilisés pour l'exclusion mutuelle de groupe. Dans ce même chapitre, nous avons présenté un autre algorithme pour le problème de l'exclusion mutuelle de groupe toujours basé sur les quorums qui met en évidence le problème des accès concurrents entre les sessions. L'idée dans cet algorithme a consisté à considérer dans un groupe de processus, un noeud jouant le rôle de coordonnateur, se chargeant ainsi de gérer l'accès à une ressource.

Dans le chapitre quatre, nous avons présenté un nouvel algorithme d'exclusion mutuelle basé sur le modèle client-serveur. Cet algorithme appartient à la classe des algorithmes répartis fondés sur une circulation de jeton. Cela suppose qu'il existe un jeton unique qui circule dans le jeton. Cet algorithme utilise une structure de données dynamique. Cette solution utilise une topologie en arbre et est basée sur une circulation du jeton. Dans cette solution, une différence est faite entre les processus et les sessions. Plusieurs processus (*clients*) peuvent accéder simultanément à une même session ouverte (*serveur*). Cet algorithme assure qu'à tout instant, au plus une seule session est ouverte, et toute session demandée sera ouverte au bout d'un temps fini. Le nombre de messages est entre 0 et *m*, où *m* est le nombres de sessions dans le réseau. Le nombre de messages nécessaires à l'ouverture d'une session est de $O(\log m)$. Le degré maximum de concurrence est *n*, où *n* est le nombre de processus dans le réseau. Dans cet algorithme, un processus n'interroge pas les autres processus lorsqu'il

veut participer à une session. Le jeton sert juste à ouvrir et fermer les sessions. Avant d'ouvrir une nouvelle session, un processus devra initialiser une phase de fermeture de la session courante, afin de s'assurer qu'aucun processus ne se trouve encore en section critique.

Dans le chapitre cinq, nous avons proposé une solution pour le problème de l'exclusion mutuelle de groupe au niveau des réseaux mobiles ad hoc. Pour le moment pratiquement pas de travaux ne sont pas encore faits dans le cadre de l'exclusion mutuelle de groupe dans les réseaux mobiles ad hoc à part ceux effectués dans les cadre de l'exclusion mutuelle dans [WWV98, BV01, MWV00, MM05] et dans le cadre de la k-exclusion mutuelle dans [WCM01, MM06]. Donc ce problème présente de belles perspectives de recherches dans la suite. Cette solution a été adaptée à l'algorithme *RL* (Reverse Link) présenté par J. Walter et al. dans [WWV98]. Cet algorithme est adapté à la mobilité des noeuds. Cet algorithme combine plusieurs idées venant de différents articles. La technique d'inversion partielle [GB81], utilisée pour maintenir un graphe acyclique direct orienté à l'arrivée dans le réseau quand la destinaton est statique, est utilisée dans cet algorithme afin de maintenir un graphe acyclique direct orienté selon le jeton avec une destination dynamique. Comme dans les algorithmes [Ray89a, DK94], chaque noeud dans l'algorithme maintient une queue de requêtes contenant les identificateurs des noeuds voisins pour lesquels il a reçu des requêtes pour le jeton. Comme dans [DK94], les noeuds sont totallement ordonnés. Cet algorithme est sensible aux formations et coupures de liens et est ainsi approprié pour les réseaux mobiles ad hoc.

5.8 Perspectives

Les solutions proposées dans cette thèse pour la résolution du problème de l'exclusion mutuelle de groupe sont basées d'une part sur les structures de quorums, et d'autre part sur une circulation de jeton. Il existe une littérature considérable sur les système de quorums ordinaires. Les quorums de surface ont été présentés par Joung afin de résoudre le problème de l'exclusion mutuelle de groupe. Plusieurs structures ont été explorées, incluant les plans projectifs finis, le vote majoritaire, les grilles, les arbres, les graphes planaires etc. Pour de futurs travaux, il serait intéressant de voir comment les autres structures peuvent être utilisées dans les systèmes de groupe de quorums. Il serait intéressant dans la suite de se lancer en profondeur dans la complexité de ces algorithmes en terme du nombre de message et dans la simulation en vue de les comparer aux quelques algorithmes d'exclusion mutuelle de groupe existants.

Dans l'algorithme d'exclusion mutuelle de groupe basé sur le modèle client-serveur, il serait intéressant de l'examiner dans le cadre de la mobilité c'est-à-dire s'il y a des changements de topologie. Nous sommes en train de faire des simulations aussi pour ce problème. On peut aussi essayer de réduire le nombre de messages échangés pour une entrée en section

critique. Pour cela une idée serait d'organiser logiquement les noeuds. Il faut noter cependant que la circulation de jeton entre les processus consomme des ressources même lorsque les processus ne sont pas demandeurs. Pour pallier ce défaut, il faudra utiliser des algorithmes pour bloquer le jeton sur un site s'il n'existe pas de requête en attente.

Pour la solution proposée concernant l'exclusion mutuelle de groupe pour les réseaux mobile ad hoc, des travaux futurs peuvent être entrepris aussi. Il serait intéressant d'essayer d'adapter l'algorithme de clusterisation exposé dans [MM05] par rapport à nos travaux. On pourra aussi voir le problème d'économie d'énergie des algorithmes d'exclusion mutuelle et d'exclusion mutuelle de groupe dans les réseaux mobiles ad hoc. Il est possible par des simulations de faire une comparaison avec d'autres approches utilisant un protocole de routage ad hoc. Avec les simulations que nous avons envisageons de faire dans le futur, nous comptons comparer la complexité en nombre de messages par entrée en section critique avec le coût des messages encouru par un algorithme d'exclusion mutuelle de groupe exécuté sur un algorithme de routage ad hoc, où les messages de l'algorithme de l'exclusion mutuelle de groupe et de l'algorithme de routage sont comptés. Une étude complète et théorique d'analyse et de performance doit être entreprise pour étuder les algorithmes les plus avancés. Le modèle de système devrait être plus réaliste en considérant des situations plus compliquées de pannes de liens. Ces questions de recherche avancées sont dans notre futur plan de travail.

Dans le cadre des perspectives liées aux solutions basées sur une circulation du jeton, il serait très intéressant de faire des simulations concernant le délai d'attente et le nombre de messages par entrée en section critique.

Publications issues de cette thèse

CEtte thèse a fait l'objet des publications suivantes :

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

2. O. THIARE, M. NAIMI, A. GUEROUI, Distributed Groups Mutual Exclusion Based on Clients/Servers Model, 7th IEEE International Conference on Parallel and Distributed Computing Applications and Technologies (PDCAT'06), pages 67-73, 4-6 December 2006 Taipei, Taiwan (2006).

3. O. THIARE, M. NAIMI, M. GUEROUI, Access Concurrents Sessions Based on Quorums, IEEE 2nd International Conference on Systems, Computing Sciences and Software Engineering (SCS^2'06) CISSE 2006 December 4-14 Bridgeport, USA (2006).

4. O. THIARE, M. NAIMI, A Quorum-Based Algorithm for Group Mutual Exclusion, ISCA 19th International Conference on Parallel and Distributed Computing Systems (PDCS'06) pages 63-69, September 20-22 San Francisco California, USA (2006).

5. O. THIARE, M. NAIMI, A. GUEROUI, Distributed Groups Mutual Exclusion Based on Clients/Servers Model, Version Journal à soumettre prochainement.

BIBLIOGRAPHIE

[AD76] P.A. Alsberg and J.D. Day. A principle for resilient sharing of distributed resources. In *Proceedings of the 2nd International Conference on Software Engineering.*, 1976.

[ADHK01] U. Abraham, S. Dolev, T. Herman, and I. Koll. Self-stabilization over unreliable *l*-exclusion. *Theoretical Computer Science*, 266 :653–692, 2001.

[AEA90] D. Agrawala and A. El Abbadi. Exploiting logical structures in replicated databases. *The Computer Journal*, 33 :71–78, 1990.

[AEA91] D. Agrawala and A. El Abbadi. An efficient and fault tolerant solution for mutual exclusion. *ACM Transactions on Computer Systems*, 9 :1–20, 1991.

[AEA97] D. Agrawal, $\ddot{O}$. E$\breve{g}$ecio$\breve{g}$lu, and A.El. Abbadi. Billard quorums on the grid. *Information Processing Letter*, 64 :9–16, 1997.

[AGR03] E. Anceaume, M. Gradinariu, and M. Roy. Self-organizing systems. case study : peer-to-peer systems. *In Brief Announcement at the 17th Int. Symposium on Distributed Computing (DISC-03)*, 2003.

[AM05] R. Atreya and N. Mittal. A dynamic group mutual exclusion algorithme using surrogate quorums. *In proceedings of the IEEE Conference on Distributed Computing Systems (ICDCS)*, 2005.

[AR98] J. Almond and M. Romberg. The unicore project : Uniform access to supercomputing over the web. *In Proceedings of the 40th Cray User Group meeting*, 1998.

[AV99] K. Alagarsamy and K. Vidyasankar. Elegant solutions for group mutual exclusion problem. *Journal of Information Science and Engineering*, 19 :415–423, 1999.

[Baz00] R.A. Bazzi. Planar quorum. *Theoretical Computer Science*, 243 :243–268, 2000.

[BB90] J. Beauquier and B. Bérard. *Systèmes d'exploitation*. Mc Graw-Hill, 1990.

[BBBAN04] M. Benchaiba, A. Bouabdallah, N. Badache, and M. Ahmed-Nacer. Distributed mutual exclusion algorithms in mobile ad hoc networks : an overview. *ACM SIGOPS Operating Systems Review*, 38 :74–89, 2004.

[BC96] S. Banerjee and P.K. Chrysanthis. A new token passing distributed mutual exclusion algorithm. *16th International Conference on Distributed Computing Systems (ICDCS)*, 1996.

[BCDP03] J. Beauquier, S. Cantarell, A.K. Datta, and F. Petit. Group mutual exclusion in tree networks. *Journal of Information Science and Engineering*, 19 :415–423, 2003.

[Ber73] C. Berge. *Graphs and hypergraphs*. American Elsevier, 1973.

[BG81] P.A. Bernstein and N. Goodman. Concurrency control and recovery in database systems. *ACM Computing Surveys*, 13 :185–221, 1981.

[BGM86] D. Barbara and H. Garcia-Molina. Mutual exclusion in partitioned distributed systems. *Distributed Computing*, 1 :119–132, 1986.

[BGW89] G.M. Brown, M.G. Gouda, and C.L. Wu. Token systems that self-stabilize. *IEEE Transactions on Computers*, 38 :845–852, 1989.

[BL00] A. Bouabdallah and C. Laforest. A distributed token-based algorithm for the dynamic allocation problem. *ACM SIGOPS Operating Systems Review*, 34 :60–68, 2000.

[BP85] J.E. Burns and K.K. Pachl. Uniform self-stabilizing rings. *ACM on Transaction on Programming Languages and Systems*, 11 :330–344, 1985.

[BV01] R. Baldoni and A. Virgillito. A token-based mutual exclusion algorithm distributed mutual exclusion for mobile ad hoc networks. Technical Report 28-01, Departimento di Informatica, Univ. di Roma, 2001.

[CAA90] S.Y. Cheung, M. Ammar, and M. Ahamad. The grid protocol : a high performance scheme for maintaining replicated data. *In IEEE 6th International Conference on Data Engineering*, 32 :824–840, 1990.

[Can03] S. Cantarell. Exclusion mutuelle de groupe et auto-stabilisation. *Thèse de Doctorat en Informatique Orsay N° d'ordre : 7403*, 2003.

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

[CF99] F. Cristian and C. Fetzer. The timed asynchronous distributed system model. *In IEEE Transactions on Parallel and Distributed Systems*, 1999.

[Cha92a] Y.I. Chang. A correct o($\sqrt{n}$) distributed mutual exclusion algorithm. *Proc. 15th International Conference on Parallel and Distributed Computing and Systems*, 1992.

[Cha92b] Y.I. Chang. Notes on maekawa's o($\sqrt{n}$) distributed mutual exclusion algorithm. *Proc. Symposium on Parallel and Distributed Systems*, 1992.

[Cha96] Y.I. Chang. A dynamic request set based algorithm for mutual exclusion in distributed systems. *ACM Operating System Review*, 30 :52–62, 1996.

[CHP71] P.J. Courtois, F. Heymans, and D.L. Parnas. Concurrent control with readers and writers. *Communications of the Association of the Computing Machinery*, 14 :667–668, 1971.

[CM84] K.M. Chandy and J. Misra. The drinking philosophers problem on mutual exclusion in computer networks. *ACM TOPLAS*, 6 :632–646, 1984.

[CP00] S. Cantarell and F. Petit. Self-stabilizing group mutual exclusion for asynchronous rings. *4th International Conference on Principles of Distributed Systems (ICDCS 2001)*, 2000.

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

[CS01] G. Cao and M. Singhal. A delay optimal quorum based mutual exclusion algorithm for distributed systems. *IEEE Transactions on Parallel and Distributed Systems*, 12 :1256–1268, 2001.

[CSL90] Y.I. Chang, M. Singhal, and M.T. Liu. A hybrid approach to mutual exclusion for distributed systems. *Proc 1990 Annual Inter. Comp. Soft. And Appli. Conf. IEEE Computer Soc*, 1990.

[CSL91] Y.I. Chang, M. Singhal, and T. Liu. A dynamic token-based distributed mutual exclusion algorithm. *10th International Conference on Computers and Communications, Scottsdale, Arizona, USA*, 1991.

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

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

[Dij65] E.W. Dijkstra. Solution of a problem in concurrent programming control. *Communications of the ACM*, 8 :569, 1965.

[Dij74] E.W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Communications of the Association of the Computing Machinery*, 17 :643–644, 1974.

[Dij78] E.W. Dijkstra. Two starvation-free of a general mutual exclusion problem. Technical Report 625, EWD, Plataanstraat 5, 5671, AL Nuenen, The Netherlands, 1978.

[DJPV00] A.K. Datta, C. Johnen, F. Petit, and V. Villain. Self-stabilizing depth-first token circulation in arbitrary rooted networks. *Distributed Computing*, 13 :207–218, 2000.

[DK94] D.M. Dhamdhere and S.S. Kulkarni. A token based k-resilient mutual exclusion algorithm for distributed systems. *Information Processing Letters*, 50 :151–157, 1994.

[DR92] D. Duchamp and N.F. Reynolds. Mesure performance of wireless lan. *Technical Report, United States*, 1992.

[FK96] I. Foster and C. Kessalman. Globus : A metacomputing infrastructure toolkit. *In Proceedings of the Workshop on Environments and Tools for Parallel Scientific Computing*, 1996.

[FLBB79] M. Fisher, N.A. Lynch, J. Burns, and A. Borodin. Resource allocation with immunity to process failure. *In FOCS79 Proceedings of the 20th Annual IEEE Symposium on Foundations of Computer Science*, 1979.

[Fly96] M.J. Flynn. Very high-speed computing systems. *Proceedings of the IEEE 54*, 1996.

[Fu90] W.C.A. Fu. *Enhancing Concurrency and Availability for database systems*. PhD thesis, Simon fraser University, Burnaby, B.C., Canada, 1990.

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

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

[Gif79] D.K. Gifford. Weighted voting for replicated data. *In Proc. 7th ACM Symposium on Operating Systems Principles*, pages 150–162, 1979.

[GMB85] H. Garcia-Molina and D. Barbara. How to assign votes in a distributed system. *Journal of the ACM*, 32 :841–860, 1985.

[Had01] V. Hadzilacos. A note on group mutual exclusion. *Proceedings in the 20th annual ACM Symposium on Principles of Distributed Computing*, 2001.

[Had02a] R. Hadid. *Algorithmes auto-stabilisant d'allocation de ressources*. PhD thesis, LaRIA, Laboratoire de Recherche en Informatique d'Amiens, Université de Picardie Jules Verne, 2002.

[Had02b] R. Hadid. Space and time efficient self-stabilizing *l*-exclusion in tree network. *Journal of Parallel and distributed Computing*, 62 :843–864, 2002.

[Hil85] W. Hillis. The connecting machine. *MIT Press*, 1985.

[HLNPRP03] M. Hurfin, J.P. Le Narzul, J. Pley, and P. Raîpin Parvédy. A fault-tolerant protocol for ressources allocation in a grid dedicated to genomic applications. *In Proc. of the Fifth International Conference on Parallel Processing and Applied Mathematics, Special Session on Parallel and Distrbuted Bioinformatic Applications 5PPAM-03), LNCS. Springer Verlag*, 2003.

[HT93] V. Hadzilacos and S. Toueg. Reliable broadcast and related problems. *In Distributed systems, ACM Press*, 1993.

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

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

[IK90] T. Ibaraki and T. Kameda. Theory of coteries. Technical Report TR90-09, Technical Report CSS/LCCR, Kyoto University, Kyoto, JAPAN, 1990.

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

[Jou00] Y.J. Joung. On generalized quorum systems. Technical report, National Taiwan University, Taipei, Taiwan, Department of Information Management, 2000.

[Jou01a] Y.J. Joung. The congenial talking philosophers problem in computer networks. *Distributed Computing*, 15 :155–175, 2001.

[Jou01b] Y.J. Joung. Quorum-based algorithms for group mutual exclusion. *IEEE Transaction on Parallel and Distributed Systems*, 14 :463–476, 2001.

[KC91] A. Kumar and S.Y. Cheung. A high availability in a hierarchical grid algorithm for replicated data. *Information Processing Letters*, 49 :213–248, 1991.

[KFYA94] H. Kakugawa, S. Fujita, M. Yamashita, and T. Ae. A distributed k-mutual exclusion algorithm using k-coterie. *Information Processing Letters*, 40 :311–316, 1994.

[KGJV83] S. Kirkpatrick, C.D. Gelatt Jr, and M.P. Vecchi. Optimization by simulated annealing. *Science*, 220 :671–680, 1983.

[KHI98] A. KHIAT. L'allocation répartie des ressources dans k groupes. *Thèse de Doctorat en Informatique Université de Franche-Compté N° d'ordre : 700*, 1998.

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

[Knu66] D.E. Knuth. Addition comments on a problem in concurrent programming control. *Communications of the ACM*, 9 :321–322, 1966.

[Kum90] A. Kumar. Performance analysis of a hierarchical quorum consensus algorithm for replicated objects. *In IEEE 10th International Conference on Distributed Computing Systems*, 1990.

[Kum91] A. Kumar. Hierarchical quorum consensus : a new algorithm for managing replicated data. *IEEE Transactions on Computers*, 40 :996–1004, 1991.

[Lam74] L. Lamport. A new solution of dijkstra's concurrent programming problem. *Communications of the ACM*, 17 :453–455, 1974.

[Lam78a] L. Lamport. The implementation of reliable distributed multiprocess systems. *Computer Networks*, 2 :95–114, 1978.

[Lam78b] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21 :558–565, 1978.

[Lam86] L. Lamport. The mutual exclusion problem : Parts i & ii. *Journal of the ACM*, 33 :313–348, 1986.

[Lap90] J.C. Laprie. Dependability : Basic concepts and associated terminology. Technical Report 3092, LAAS-CNRS, ESPRIT Project, 1990.

[LeL77] G. LeLann. Distributed systems : towards a formal approach. *Information Proceeding'77*, 1977.

[LL89] L. Lamport and N. Lynch. Chapter on distributed computing. Technical Report MIT/LCS/TM-384, Massachusetts Institute of Technology, Cambridge, Massachusetts, 1989.

[Lov73] L. Lov$\acute{a}$sz. Coverings and colorings of hypergraphs. *In Proceedings of the 4th Southeastern Conference on Combinatorics, Graph Theory, and Computing*, 1973.

[Mae85] M. Maekawa. A $\sqrt{n}$ algorithm for mutual exclusion in decentralized systems. *ACM Transactions onComputer Systems*, 3 :145–159, 1985.

[MM05] R. Mellier and J.F. Myoupo. A clustering mutual exclusion protocol for multihop mobile ad hoc networks. *IEEE International Conference on Networks (ICON'05), IEEE Press*, 2005.

[MM06] R. Mellier and J.F. Myoupo. Fault-tolerant mutual and k-mutual exclusion algorithms for single-hop mobile ad hoc networks. *International Journal of Ad Hoc and Ubiquitous Computing (IJAHUC)*, 1, 2006.

[MNK96] M. Mizuno, M. Nesterenko, and H. Kakuga. Lock-based self-stabilizing distributed mutual exclusion algorithms. *In Proceedings of the 16th International Conference on Distributed Computing Systems (ICDCS'91)*, 1996.

[MNR91a] M. Mizuno, M.L. Neilsen, and R. Rao. A token based distributed mutual exclusion algorithm based on quorum agreements. *11th International Conference on Distributed Computing*, 20 :361–368, 1991.

[MNR91b] M. Mizuno, M.L. Neilsen, and R. Rao. A token based distributed mutual exclusion algorithm based on quorum agreements. *In Proceedings of the 11th International Conference on Distributed Computing Systems (ICDCS'91)*, 1991.

[MS90] S. Mishra and P. Srimani. Fault tolerant mutual exclusion algorithms. *Journal of systems Software*, 11 :111–129, 1990.

[MT99] Y. Manabe and N. Tajima. (h,k)-arbiter for h-out of-k mutual exclusion problem. *In Proceedings of the 13th International Symposium on Distributed Computing (DISC'99)*, 1999.

[MWV00] N. Malpani, J.L. Welch, and N.H. Vaidya. Leader election algorithms for mobile ad hoc networks. *Proceedings of the 4th International Workshop on Discrete Algorithms and Methods for Mobile Computing and Communications (PODC'99)*, 2000.

[Nad98] B. Nadjib. La mobilité dans les systèmes répartis. *Technical Report, France*, 1998.

[Nai87] M. Naimi. *Une structure arborescente pour une classe d'algorithmes distribués d'exclusion mutuelle*. PhD thesis, University of Franche Comté, 1987.

[Nai93] M. Naimi. Distributed algorithm for k-entries to critical section based on directed graphs. *ACM Operating Systems Review*, 27 :67–75, 1993.

[NM91] M.L. Neilsen and M. Mizuno. A dag-based algorithm for distributed mutual exclusion. *In IEEE 11th International Conference on Distributed Computing Systems*, 1991.

[NM94] M.L. Neilson and M. Mizuno. Non dominated k-coteries for multiple mutual exclusion. *Information Processing Letters*, 50 :247–252, 1994.

[NM02] M. Nesterenko and M. Muzino. A quorum-based self-stabilizing distributed mutual exclusion algorithm. *Journal of Parallel and Distributed Computing*, 62 :284–305, 2002.

[NT87a] M. Naimi and M. Trehel. A distributed algorithm for mutual exclusion based on data structures and fault tolerance. *In IEEE Phoenix Conference on Computers and Communications*, 1987.

[NT87b] M. Naimi and M. Trehel. How to detect a failure and regenerate the token in the log(n) distributed algorithm for mutual exclusion. *Lecture Notes in Computer Science*, 312 :155–166, 1987.

[NT87c] M. Naimi and M. Trehel. An improvement of the $\log(n)$ distributed algorithm for mutual exclusion. *In the 7th IEEE International Conference on Distribued Computing Systems proceedings*, 1987.

[NTA96] M. Naimi, M. Trehel, and A. Arnold. A log(n) distributed mutual exclusion algorithm based on path reversal. *Journal of Parallel and Distributed Computing*, 34 :1–13, 1996.

[OvG89] R. Otten and L. van Ginnekin. *The annealing Algorithm*. Kluwer Academic Publishers, 1989.

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

[Pet81] G.L. Peterson. Myths about the mutual exclusion problem. *Information Processing Letters*, 12 :115–116, 1981.

[RA81] G. Ricart and A.K. Agrawala. An optimal algorithm for mutual exclusion in computer networks. *Communications of the ACM (CACM)*, 24 :9–17, 1981.

[RA83] G. Ricart and A.K. Agrawala. Author's response to on mutual exclusion in computer networks. *Communications of ACM*, 26 :146–148, 1983.

[Ran92] S. Rangarajan. Fault tolerant algorithms for replicated data management. *In IEEE international Conference on Data Engineering*, 1992.

[Ray85a] M. Raynal. *Algorithmes distribués et protocoles*. Editions Eyrolles, 140 pages, 1985.

[Ray85b] M. Raynal. *Algorithmique du parallélisme, le problème de l'exclusion mutuelle*. DUNOD Informatique, 1985.

[Ray86] M. Raynal. Algorithms for mutual exclusion. *MIT Press*, 1986.

[Ray89a] K. Raymond. A tree based algorithm for distributed mutual exclusion. *ACM Transactions on Computer Systems*, 7 :61–77, 1989.

[Ray89b] K. Raymond. A tree-based algorithm for distributed mutual exclusion. *ACM Transaction on Computer Systems*, 7 :61–77, 1989.

[Ray91a] M. Raynal. A distributed algorithm for the *k*-out of-*m* ressources allocations problem. *Proceedings of the first Conference on Computing and Informations, Sppringer-Verlag LNCS*, 497 :599–609, 1991.

[Ray91b] M. Raynal. A simple taxonomy for distributed mutual exclusion algorithms. *ACM Operating Systems Review*, 1991 :47–49, 1991.

[Ray92a] M. Raynal. *Gestion des données réparties : Problèmes et Protocoles*. Editions Eyrolles, 193 pages, 1992.

[Ray92b] M. Raynal. *Synchronisation et état global dans les systèmes répartis*. Eyrolles, Collection de la direction des Ètudes et Recherches d'Électricité de France, 1992.

[San87] B.A. Sanders. The informations structure of mutual exclusion algorithms. *ACM Transactions on Computer Systems*, 5 :284–299, 1987.

[Sin85] M. Singhal. A class of deadlock free maekawa type algortihms for mutual exclusion in distributed systems. *Distributed Computing*, 4 :145–159, 1985.

[Sin89] M. Singhal. A heuristically-aided algorithm for mutual exclusion in distributed systems. *IEEE Transactions on Computers*, 38 :651–662, 1989.

[Sin92] M. Singhal. A dynamic information structure mutual exclusion algorithm for distributed systems. *In IEEE Transactions on Parallel and Distribued Systems*, 3 :121–125, 1992.

[Sin93] M. Singhal. A taxonomy of distributed mutual exclusion. *Journal of Parallel and Distributed Computing*, 18 :94–101, 1993.

[SK85] I. Suzuki and T. Kasami. A distributed mutual exclusion algorithm. *In ACM Symposium on Principles of Distributed Computing (PODC)*, 1985.

[SK93] I. Suzuki and T. Kasami. An optimality theory for mutual exclusion networks. *Proc. of the 3th Conference on Disributed Computing Systems, Miami*, 1993.

[SM97] M. Singhal and D. Manivannan. A distributed mutual exclusion algorithm for mobile computing environments. *IEEE International Conference on Intelligent Information Systems, IASTED*, 1997.

[Spe28] E. Sperner. Ein satz uber untermengen einer endlichen menge. *Math. Z.*, 27 :544–548, 1928.

[SR92] P.K. Srimani and R.L.N. Reddy. Another distributed algorithm for multiples entries to a critical section. *Informations Processing Letters*, 41 :51–57, 1992.

[Tho79] R.W. Thomas. A majority consensus approach to concurrency control for multiple copy databases. *ACM Transactions on Databases systems*, 4 :180–209, 1979.

[TK88] Z. Tong and R.Y. Kain. Vote assignments in weighted voting mechanism. *In Proc. of the 7th Symposium on reliable Distributed Systems*, 1988.

[TN87a] M. Trehel and M. Naimi. A distributed algorithm for mutual exclusion based on data structures and fault tolerance. *In IEEE 6th International Conference on Computers and Communications*, 1987.

[TN87b] M. Trehel and M. Naimi. Un algorithme distribué d'exclusion mutuelle en log(n). *TSI*, 6 :141–150, 1987.

[TN90] J. Tang and N. Natarajan. A static pessimistic scheme for managing replicated databases. Technical Report CS-90-07, Pennsylvania State University, University Park, Pennsylvania, 1990.

[TN06] O. Thiare and M. Naimi. A quorum-based algorithm for group mutual exclusion. *ISCA 19th International Conference on Parallel and Distributed Computing Systems (PDCS'06), San Francisco California, USA*, 2006.

[TNG06a] O. Thiare, M. Naimi, and M. Gueroui. Acces concurrents sessions based on quorums. *IEEE International Conference on Systems, Computing Sciences and Software Engineering (CISSE 2006), Bridgeport, USA*, 2006.

[TNG06b] O. Thiare, M. Naimi, and M. Gueroui. Distributed groups mutual exclusion based on client/server model. *IEEE 7th International Conference on Parallel and Distributed Computing Applications and Technologies, 4-6 December, Taipei, Taiwan*, 2006.

[TNG06c] O. Thiare, M. Naimi, and M. Gueroui. A group mutual exclusion algorithm for mobile ad hoc networks. *IEEE International Conference on Systems, Computing Sciences and Software Engineering (CISSE 2006), Bridgeport, USA*, 2006.

[Tse95] Y.C. Tseng. Distincting termination by weight-throwing in a faulty distributed system. *Journal of Parallel and Distributed Computing*, 25 :7–15, 1995.

[Vil99] V. Villain. A key tool for optimally in the state model. *In DIMACS Workshop on Distributed Data and Structures, Carleton University Press*, 1999.

[WCM01] J. Walter, G. Cao, and M. Mohanty. A k-mutual exclusion algorithm for ad hoc wireless networks. *Proceedings of the first annual Workshop on Principles of Mobile Computing (POMC 2001)*, 2001.

[WJ99a] K.P. Wu and Y.J. Joung. Asynchronous group mutual exclusion algorithm in ring networks. *13th International Parallel Processing Symposium and 10th Symposium on Parallel and distributed Processing (IPPS/SPDP'99)*, 1999.

[WJ99b] K.P. Wu and Y.J. Joung. Asynchronous group mutual exclusion in ring networks. Technical Report 28-01, Department of Information Management, National Taiwan University, Taipei, Taiwan, 1999.

[WK97] J.E. Walter and S. Kini. Mutual exclusion on multihop wireless networks. *Texas A&M Univ., College Station, TX 77 Journal of Parallel and Distributed Computing*, 25 :7–15, 1997.

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