Algorithmique distribuée d'exclusion mutuelle : vers une gestion efficace des ressources

Jonathan Lejeune

▶ To cite this version:

Jonathan Lejeune. Algorithmique distribuée d'exclusion mutuelle : vers une gestion efficace des ressources. Autre [cs.OH]. Université Pierre et Marie Curie - Paris VI, 2014. Français. NNT : 2014PA066174 . tel-01077962

HAL Id: tel-01077962
https://theses.hal.science/tel-01077962

Submitted on 27 Oct 2014

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

École doctorale EDITE de Paris

Algorithmique distribuée d'exclusion mutuelle : vers une gestion efficace des ressources

THÈSE

présentée et soutenue publiquement le 19 septembre 2014

pour l'obtention du

Doctorat de l'Université Pierre et Marie Curie

(mention informatique)

par

Jonathan Lejeune

Composition du jury

Président :	Franck Petit	Professeur UPMC
Rapporteurs :	Abdelmadjid Bouabdallah	Professeur UTC
	Christine Morin	Directrice de recherche Inria
Examinateurs :	Achour Mostefaoui	Professeur Université de Nantes
	Mohamed Naimi	Professeur Université de Cergy-Pontoise
Encadrants :	Luciana Arantes	Maître de conférences UPMC
	Julien Sopena	Maître de conférences UPMC
Directeur de thèse :	Pierre Sens	Professeur UPMC

Laboratoire d'Informatique de Paris 6 — UMR 7606

Mis en page avec la classe thesul.

Cette thèse est dédiée à :
ma mère
mon père
pépère Alexandre
mémère Antoinette
pépère Michel
mémère Mireille
et Rebecca

Remerciements

Une thèse marque la fin de longues années d'études. Sa rédaction permet de se rendre compte qu'un travail d'expertise approfondi a été accompli et donne l'agréable sensation d'avoir apporté une pierre à l'édifice. Je tiens dans un premier temps à remercier toute personne qui a contribué d'une façon ou d'une autre à la réalisation et à l'aboutissement de cette thèse.

Je n'aurais jamais pu imaginer, il y a 8 ans, lors de mes premiers pas dans l'enseignement supérieur, dans ma petite université de Picardie, que je serai allé découvrir la capitale pour préparer un doctorat en informatique. Je n'étais d'ailleurs pas destiné initialement à l'informatique croyant à tort qu'il fallait être un "geek" pour y prendre goût. J'ai donc un premier remerciement pour M. Gwenaël Richomme qui a contribué grandement à mon changement d'orientation grâce à son cours passionnant d'introduction à l'algorithmique et à la programmation. Lors de ma venue à Paris en septembre 2009 pour préparer le master SAR de l'UPMC, je pensais encore être destiné à l'industrie. Le changement de cap, je le dois à Olivier Marin que je remercie pour m'avoir aider à faire le premier pas dans le domaine de la recherche grâce à son sujet de projet sur le Map/Reduce. Le projet a ensuite débouché sur un stage au LIP6 où j'ai fait la connaissance de Julien Sopena et de Luciana Arantes. C'est sans nul doute grâce à ce stage ayant abouti sur deux publications et un voyage à Lisbonne que l'idée de faire une thèse commençait à germer.

Cette thèse fait suite à mon stage de fin d'études qui m'a été proposé par Pierre Sens, Julien Sopena et Luciana Arantes. Ce stage a été très bénéfique pour moi car il m'a donné 6 précieux mois de plus et a sans nul doute contribué à terminer cette thèse dans les temps. Je tiens donc à remercier chaleureusement mon directeur de thèse et mes encadrants pour la confiance qu'ils m'ont accordée lors de ce recrutement ainsi que l'aide et conseils précieux qu'ils m'ont apportés durant ces trois ans.

J'adresse un grand merci à mes collègues doctorants et ex-doctorants de l'équipe REGAL (en particulier Florian David qui a été aussi un très bon colocataire) et de l'équipe Move.

Je remercie particulièrement Sébastien Monnet qui a participé quasiment à toutes mes répétitions de présentation et de soutenance de cette thèse.

Je remercie le projet ANR MyCloud qui a été à l'origine du sujet ainsi que ses membres avec qui j'ai eu d'enrichissantes conversations scientifiques.

Je remercie Mme Christine Morin et M. Abdelmadjid Bouabdallah d'avoir accepté de rapporter cette thèse ainsi que M. Mohamed Naimi, M. Achour Mostefaoui et M. Franck Petit d'avoir accepté de faire partie du jury.

Je terminerai cette page en remerciant les personnes de ma famille qui ont très grandement contribué à mon éducation et sans qui je ne serai pas ce que je suis devenu : mes parents et mes grands-parents. Il me reste à remercier immensément ma chère Rebecca qui fait mon bonheur tous les jours et qui m'a sans cesse encouragé pendant ces trois ans.

Résumé

Les systèmes à grande échelle comme les Grilles ou les Nuages (Clouds) mettent à disposition pour les utilisateurs des ressources informatiques hétérogènes. Dans les Nuages, les accès aux ressources sont orchestrés par des contrats permettant de définir un niveau de qualité de service (temps de réponse, disponibilité ...) que le fournisseur doit respecter. Ma thèse a donc contribué à concevoir de nouveaux algorithmes distribués de verrouillage de ressources dans les systèmes large échelle en prenant en compte des notions de qualité de service. Dans un premier temps, mes travaux de thèse se portent sur des algorithmes distribués de verrouillage ayant des contraintes en termes de priorités et de temps. Deux algorithmes d'exclusion mutuelle ont été proposés : un algorithme prenant en compte les priorités des clients et un autre pour des requêtes avec des dates d'échéance. Dans un second temps, j'ai abordé le problème de l'exclusion mutuelle généralisée pour allouer de manière exclusive plusieurs types de ressources hétérogènes. J'ai proposé un nouvel algorithme qui réduit les coûts de synchronisation en limitant la communication entre processus non conflictuels. Tous ces algorithmes ont été implémentés et évalués sur la plateforme nationale Grid 5000. Les évaluations ont montré que nos algorithmes satisfaisaient bien les contraintes applicatives tout en améliorant de manière significative les performances en termes de taux d'utilisation et de temps de réponse.

Mots-clés: Algorithmique distribuée, exclusion mutuelle, QoS, expérimentation

Abstract

Distributed large-scale systems such as Grids or Clouds provide large amounts of heterogeneous computing resources. Clouds manage ressource access by contracts that allow to define a quality of service (response time, availability, ...) that the provider has to respect. My thesis focuses on designing new distributed locking algorithms for large scale systems that integrate notions of quality of service. At first, my thesis targets distributed locking algorithms with constraints in terms of priorities and response time. Two mutual exclusion algorithms are proposed : a first algorithm takes into account client-defined priorities and a second one associates requests with deadlines. I then move on to a generalized mutual exclusion problem in order to allocate several types of heterogeneous resources in a exclusive way. I propose a new algorithm that reduces the cost of synchronization by limiting communication between non-conflicting processes. All algorithms have been implemented and evaluated over the national platform Grid 5000. Evaluations show that our algorithms satisfy applicative constraints while improving performance significatively in terms of resources use rate and response time.

Keywords: Distributed algorithm, mutual exclusion, QoS, experiments

Sommaire

1 Introduction générale **1**
1.1 Présentation et contexte général de la thèse 1
1.2 Contributions 2
1.3 Organisation du manuscrit 3

Partie I Exclusion mutuelle distribuée : concepts, modèles et algorithmes

2 Exclusion mutuelle distribuée **7**
2.1 Introduction 7
2.2 Description et formalisation du système considéré 8
2.3 Définition de l'exclusion mutuelle 9
2.3.1 États des processus 9
2.3.2 Propriétés 10
2.3.3 Ordonnancement 10
2.3.4 Métriques d'évaluation de performances 10
2.4 Taxonomie des algorithmes 11
2.4.1 Algorithmes à permissions 11
2.4.2 Algorithmes à jeton 12
2.4.3 Algorithme centralisé 14
2.4.4 Algorithmes hybrides et hiérarchiques 14
2.5 Algorithmes de bases des contributions de la thèse 15
2.5.1 L'algorithme de Raymond 15
2.5.2 Les algorithmes de Naimi-Tréhel 18
2.6 Extensions de l'exclusion mutuelle 20
2.7 Conclusion 22

3 Exclusion mutuelle à priorités 25
3.1 Introduction 25
3.2 Extension du modèle du système considéré 26
3.3 Taxonomie des algorithmes à priorité 26
3.3.1 Priorités statiques 26
3.3.2 Priorités dynamiques 27
3.4 Description des algorithmes principaux 27
3.4.1 Algorithme de Mueller 27
3.4.2 Algorithme de Chang 28
3.4.3 Algorithme de Kanrar-Chaki 29
3.5 Conclusion 29

Partie II Exclusion mutuelle à priorité et exclusion mutuelle à contraintes temporelles 31

4 Temps d'attente et inversions de priorité dans l'exclusion mutuelle 33
4.1 Introduction 34
4.2 Définition des inversions 34
4.3 Réduction des inversions 36
4.3.1 Corps de l'algorithme 36
4.3.2 Amélioration des communications 38
4.3.3 Retard d'incrémentation de priorité 39
4.3.4 Prise en compte de la topologie 39
4.4 Évaluation du mécanisme de réduction d'inversions 40
4.4.1 Protocole d'évaluation 40
4.4.2 Résultats en charge constante 41
4.4.3 Résultats en charge dynamique 49
4.4.4 Étude en charge constante avec priorité constante 51
4.4.5 Synthèse de l'évaluation 55
4.5 Réduction du temps d'attente 55
4.5.1 Un équilibre sur deux objectifs contradictoires 56
4.5.2 Principes de l'algorithme *Awareness* 56
4.5.3 Description de l'algorithme *Awareness* 57

4.6 Évaluation des performances de l'algorithme *Awareness* 61
4.6.1 Protocole d'évaluation 62
4.6.2 Résultats pour une fonction de palier donnée 62
4.6.3 Impact de la fonction de palier sur les inversions et le temps de réponse 65
4.7 Conclusion 66

5 Exclusion mutuelle avec dates d'échéance 69
5.1 Introduction 69
5.2 Motivations 70
5.2.1 Cloud Computing 70
5.2.2 Service Level Agreement 71
5.3 Description de l'algorithme 71
5.3.1 Description générale 71
5.3.2 Contrôle d'admission 74
5.3.3 Mécanisme de préemption 77
5.4 Évaluation des performances 81
5.4.1 Protocole d'évaluation 81
5.4.2 Métriques 82
5.4.3 Impact global 82
5.4.4 Impact de la préemption pour une charge donnée 84
5.4.5 Impact de la charge 86
5.5 Conclusion 86

Partie III Contribution dans la généralisation de l'exclusion mutuelle 89

6 Présentation de l'exclusion mutuelle généralisée 91
6.1 Introduction 91
6.2 Généralités et notations 92
6.3 Le modèle à une ressource en plusieurs exemplaires 92
6.3.1 Section critique à entrées multiples ou k-mutex 92
6.3.2 Plusieurs exemplaires par demande 93

6.4 Le modèle à plusieurs ressources en un seul exemplaire 94
6.4.1 Les conflits 94
6.4.2 Propriétés à respecter 95
6.4.3 Algorithmes incrémentaux. 95
6.4.4 Algorithmes simultanés 96
6.5 Le modèle à plusieurs ressources en plusieurs exemplaires 98
6.6 Conclusion et synthèse 99

7 Verrouiller efficacement les ressources sans connaissance préalable des conflits 101
7.1 Introduction 101
7.2 Objectifs 102
7.3 Suppression du verrou global 104
7.3.1 Mécanisme de compteurs 104
7.3.2 Ordonnancement total des requêtes 104
7.4 Évaluation 105
7.4.1 Protocole d'évaluation 105
7.4.2 Résultats sur le taux d'utilisation 107
7.4.3 Résultats sur le temps d'attente 109
7.5 Ordonnancement dynamique 109
7.5.1 Principes 109
7.5.2 Évaluation 111
7.6 Conclusion 112

Partie IV Conclusion générale 113

8 Conclusion générale 115
8.1 Conclusion 115
8.2 Perspectives 116
8.2.1 Perspectives spécifiques aux contributions 117
8.2.2 Perspectives globales 117

Bibliographie 119

Publications associées à cette thèse 125

Liste des notations **127**

Annexes **129**

A Implémentation distribuée de l'algorithme d'exclusion mutuelle généralisée **129**

A.1 Généralités . . . 129
A.2 Les messages . . . 130
A.2.1 Information véhiculée . . . 130
A.2.2 Mécanisme d'agrégation . . . 130
A.3 États des processus . . . 131
A.4 Variables locales . . . 131
A.5 Description . . . 132

Table des figures

2.1 Machine à états de l'exclusion mutuelle classique . . . 9
2.2 Algorithme de Raymond . . . 16
2.3 Exemple d'exécution de l'algorithme de Raymond . . . 17
2.4 Algorithme de Naimi-Tréhel avec file distribuée [NT87a] . . . 19
2.5 Algorithme de Naimi-Tréhel avec files locales [NT87b] . . . 19
2.6 Exemple d'exécution des deux versions de l'algorithme de Naimi-Tréhel . . 21
2.7 Schéma récapitulatif de l'état de l'art de l'exclusion mutuelle . . . 24

3.1 Exemple d'exécution de l'algorithme de Kanrar-Chaki . . . 30

4.1 Exemple de classe d'inversions de priorité . . . 35
4.2 Algorithme retard-distance . . . 37
4.3 Performances du mécanisme de retard avec une charge intermédiaire . . . 42
4.4 Analyse approfondie des inversions pour une charge intermédiaire . . . 44
4.5 Étude de l'impact de la charge sur le nombre de messages, le taux d'utilisation et le nombre d'inversions . . . 46
4.6 Impact de la charge sur les requêtes pénalisées et favorisées . . . 48
4.7 Rapport (nombre total d'inversions / nombre total de requêtes) avec une charge dynamique . . . 50
4.8 Schémas des distributions de priorités considérées dans l'arbre . . . 52
4.9 Temps de réponse moyen en fonction de la distribution des priorités dans l'arbre . . . 53
4.10 Temps de réponse moyen en fonction de la distribution des priorités dans l'arbre (tableau des valeurs) . . . 54
4.11 Différence de comportement entre les différents algorithmes . . . 58
4.12 Ordre de grandeur en nombre d'émissions de requêtes de priorité p, pour acheminer le jeton en zone de priorité $p' < p$ . . . 58
4.13 Algorithme Awareness . . . 59
4.14 Performances de l'algorithme "Awareness" en charge moyenne ($\rho = 0.5N$) et en charge haute ($\rho = 0.1N$) . . . 63
4.15 Étude de cinq familles de fonction de palier en charge moyenne ($\rho = 0.5N$) et en charge haute ($\rho = 0.1N$) . . . 67

5.1 Algorithme à dates d'échéances . . . 72
5.2 Contrôle d'admission avec acquittement . . . 76

5.3 Contrôle d'admission sans acquittement 78
5.4 Exemple pour comparer les deux stratégies de validation 78
5.5 Fonction de condition de préemption 79
5.6 Description des différentes politiques de préemption 80
5.7 Comparaison globale 83
5.8 Impact de la taille de préemption ψ 85
5.9 Impact de la charge pour $\psi = 4$ 86

6.1 Exemple de construction de graphe de conflit 94
6.2 Exemple illustrant l'effet domino 96
6.3 Schéma récapitulatif de la généralisation de l'exclusion mutuelle distribuée 99

7.1 Illustration de l'impact des objectifs sur le taux d'utilisation 103
7.2 Exemple d'exécution d'obtention des compteurs 105
7.3 Illustration du taux d'utilisation pour l'exclusion mutuelle généralisée . . . 107
7.4 Impact sur le taux d'utilisation 108
7.5 Impact sur le temps d'attente moyen 110
7.6 Évaluation du mécanisme de prêt en mémoire partagée 111

A.1 Structures véhiculées par les messages de l'implémentation distribuée . . . 131
A.2 Machine à états des processus 131
A.3 Implémentation distribuée : procédures d'initialisation, de demande et de libération de section critique 133
A.4 Implémentation distribuée : procédures de réception de requêtes 1 et 2 et de compteur 135
A.5 Implémentation distribuée : procédure de réception et d'envoi de jeton . . . 136

Chapitre 1

Introduction générale

Sommaire

1.1 **Présentation et contexte général de la thèse** 1
1.2 **Contributions** 2
1.3 **Organisation du manuscrit** 3

1.1 Présentation et contexte général de la thèse

Si nos sociétés modernes sont souvent décrites comme égoïstes, elles sont en réalité basées sur le partage des ressources : le guichet de bureau de poste, la caisse du supermarché, le médecin ou bien encore la bouteille de vin du repas dominical. Ce partage peut être inéquitable et résulte plus d'une somme d'intérêts individuels que d'une véritable générosité. Il ne serait en effet guère rentable d'avoir son propre médecin, son propre réseau postal, etc. Une mutualisation des ressources est donc indispensable et nous oblige à définir des règles, des conventions sociales pour l'organiser : prise de rendez-vous, émission de tickets, salles d'attente, politesse, ... Ces protocoles structurent notre société. D'un point de vue informatique, les programmes évoluent dans un système composé de ressources partagées pouvant aussi bien être matérielles (processeur, carte graphique, carte réseau, disque, etc.) que logicielles (variables, table de base de données, etc.). Il faut donc à l'instar de la société humaine définir des protocoles, des algorithmes pour synchroniser les accès des différents processus.

Ce problème de synchronisation est un des piliers de l'algorithmique. Identifié par Edsger Dijkstra [Dij65], il est connu sous le nom de l'**exclusion mutuelle**. Ce paradigme permet d'assurer que l'exécution d'une portion de code manipulant une ressource partagée (**section critique**) se fera toujours de manière exclusive (**propriété de sureté**) et que tout processus souhaitant l'utiliser y accédera en temps fini (**propriété de vivacité**).

Pour mettre en place un algorithme d'exclusion mutuelle, les processus doivent obligatoirement communiquer. Ils peuvent alors communiquer de deux façons : soit par l'intermédiaire d'une mémoire partagée (principalement rencontré dans le système d'exploitation d'une machine), soit par l'intermédiaire d'un réseau par passage de messages (principalement rencontré dans les systèmes répartis comme les clusters, les grilles ou les nuages).

Le sujet de cette thèse porte sur l'exclusion mutuelle par passage de messages. Dans ce mode de communication, de nombreuses solutions ont été apportées pour lesquelles on peut distinguer deux grandes classes : l'approche à permissions [Lam78, RA81, Mae85] où il est possible d'entrer en section critique après la réception de la permission d'un ensemble de processus et l'approche basée sur la circulation d'un jeton unique [Mar85, SK85, NT87a] où la possession du jeton donne le droit exclusif d'entrer en section critique. Le jeton se transmet alors généralement entre les processus sur une topologie logique comme un anneau [Mar85] ou un arbre [Ray89b, NT87a].

Les nombreux algorithmes distribués d'exclusion mutuelle ne sont pas forcément bien adaptés aux besoins spécifiques des systèmes distribués modernes. Ces systèmes tels que les Clouds mettent souvent à disposition un ensemble de ressources partagées hétérogènes et les applications s'exécutant sur ces systèmes peuvent avoir des contraintes différentes en termes de priorité, temps de réponse ou fiabilité. Plusieurs extensions de l'exclusion mutuelle ont été introduites ces dernières années pour prendre en compte les priorités des demandes d'accès, assurer un accès exclusif à plusieurs exemplaires de ressources (k-mutex), gérer plusieurs types de ressources, tolérer les pannes, etc. Cependant, la plupart de ces algorithmes se montrent inefficaces et/ou inadaptés aux grands systèmes actuels, à cause de coûts de synchronisation élevées, de temps d'attente trop important ou un non respect des priorités. Cette thèse a donc pour but de concevoir de nouveaux algorithmes d'exclusion mutuelle. Nous nous intéressons plus particulièrement à la prise en compte des requêtes à priorités différentes, à assurer l'accès à une ressource avant une date d'échéance requise et enfin à l'exclusion mutuelle généralisée pour gérer les requêtes nécessitant plusieurs ressources.

1.2 Contributions

Cette thèse apporte les trois contributions suivantes dans le domaine de l'exclusion mutuelle distribuée.

Exclusion mutuelle distribuée à priorités. Les algorithmes d'exclusion mutuelle classiques assurent que les requêtes soient satisfaites selon un ordre FIFO. Cependant, un tel ordre peut être incompatible avec les différents niveaux de requêtes exprimées par les clients. L'exclusion mutuelle à priorité permet de prendre en compte cette différence. Son but est de satisfaire les requêtes en respectant l'ordre des priorités. Cependant, un respect strict de cet ordre peut amener à des famines pour les requêtes de faibles priorités. Un mécanisme de priorités dynamiques que l'on peut trouver dans les algorithmes de Chang [Cha94] ou Kanrar-Chaki [KC10] est donc indispensable afin d'assurer que toute requête atteindra en temps fini la priorité maximale et ainsi préserver la vivacité. Mais une telle stratégie génère beaucoup d'inversions de priorités. Nous proposons donc un premier algorithme afin de ralentir l'incrémentation de priorité ([CCgrid12, Compas13]). Bien que ce ralentissement assure toujours la propriété de vivacité, les petites priorités peuvent avoir des temps d'attente énormes dans certaines configurations. Un second algorithme a donc été proposé dans [ICPP13]. Cet algorithme se base sur la circulation d'un jeton dans une topologie d'arbre statique et permet de réduire considérablement le temps d'attente

d'obtention de l'accès à la section critique pour un taux d'inversions donné.

Exclusion mutuelle distribuée à contrainte temporelles. Les algorithmes actuels n'intègrent pas les notions de qualité de service en termes de temps de réponse. Nous avons donc proposé une extension de l'exclusion mutuelle en prenant en compte les contraintes temporelles exprimées par les applications. Notre algorithme publié dans [CCgrid12, CCgrid13] se base sur la circulation d'un jeton dans un arbre statique et permet aux requêtes clientes de spécifier au moment de leur émission une date d'échéance de satisfaction. Un mécanisme de contrôle d'admission accepte ou refuse les requêtes en fonction de l'état actuel du système. Si une requête passe avec succès le contrôle, le système s'engage à satisfaire la requête avant la date d'échéance (exigence client). Afin de maximiser l'utilisation de la ressource partagée (exigence fournisseur), l'algorithme ordonnance les requêtes en fonction de leur localité dans la topologie tout en respectant les dates d'échéance des requêtes acceptées.

Exclusion mutuelle généralisée à plusieurs ressources. Il est possible dans un système à grande échelle qu'une section critique concerne plusieurs ressources. Il s'agit du paradigme du "Cocktail des philosophes" introduit par Chandy-Misra [CM84]. Le fait d'introduire plusieurs ressources dans une requête peut amener à des interblocages dans les requêtes conflictuelles (qui demandent des ressources communes) violant ainsi la vivacité. La plupart des algorithmes existants [Lyn81, SP88, CM84, GSA89, BL00] peuvent résoudre ce problème d'interblocage soit au prix d'une connaissance préalable sur les conflits entre les requêtes amenant une hypothèse très forte sur le système soit au prix d'un coût important de synchronisation impliquant une perte d'efficacité dans l'utilisation des ressources. Nous avons donc conçu un algorithme qui ne nécessite pas de connaître a priori les requêtes conflictuelles et qui réduit de manière significative les coûts de synchronisation. Cet algorithme limite entre autres les échanges entre les sites qui n'accèdent pas aux mêmes ressources. Cette réduction des coûts réduit le temps d'attente global des requêtes et améliore le taux d'utilisation des ressources. Cette contribution a donné lieu à une publication dans une conférence francophone [Compas14]. Une version internationale est en cours de soumission.

Tous les algorithmes produits dans cette thèse ont été implémentés avec OpenMPI [oMP] et évalués sur la plate-forme nationale académique Aladin Grid 5000 [g5k]. Ces évaluations montrent un gain très important de nos algorithmes par rapport aux algorithmes de l'état de l'art.

1.3 Organisation du manuscrit

Ce manuscrit s'articule autour de quatre parties. La première partie présente l'exclusion mutuelle de manière générale. Cette partie est composée de deux chapitres :

- le chapitre 2 introduit le problème de l'exclusion mutuelle, donne ses principales propriétés et présente une taxonomie d'algorithmes répartis. Les algorithmes de

bases sur lesquels nos travaux reposent (Raymond [Ray89b] et Naimi-Tréhel [NT87a, NT87b]) seront présentés plus en détails.
- le chapitre 3 présente un état de l'art de l'exclusion mutuelle à priorité. Il présente également les algorithmes sur lesquels nos travaux se baseront (Chang [Cha94], Kanrar-chaki [KC10] et Mueller [Mue99])

La deuxième partie présente les contributions relatives à la synchronisation autour d'une seule ressource :
- le chapitre 4 présente notre contribution sur l'exclusion mutuelle à priorités. On décrit dans un premier temps notre algorithme réduisant le nombre d'inversions de priorités avec une étude de performances approfondie. Dans un second temps nous présenterons l'algorithme amélioré permettant d'équilibrer le temps d'attente et le taux d'inversions indépendamment de la topologie considérée.
- le chapitre 5 présente la contribution sur les requêtes à contrainte de temps. Les différents mécanismes qui composent cet algorithme y sont détaillés et évalués.

La troisième partie traite du problème de l'exclusion mutuelle généralisée :
- le chapitre 6 présente un état de l'art des différents modèles de généralisation d'exclusion mutuelle. Il schématise également les liens entre ces modèles.
- le chapitre 7 présente les principes de notre algorithme d'exclusion mutuelle généralisée pour verrouiller plusieurs ressources hétérogènes. L'implémentation distribuée étant relativement complexe est décrite en annexe de ce manuscrit.

Enfin, la quatrième partie conclut ce manuscrit. Cette partie rappelle les contributions et présente en dernier lieu nos ouvertures et perspectives de recherche.

Première partie

Exclusion mutuelle distribuée : concepts, modèles et algorithmes

Chapitre 2

Exclusion mutuelle distribuée

Sommaire

2.1 Introduction **7**
2.2 Description et formalisation du système considéré **8**
2.3 Définition de l'exclusion mutuelle **9**
2.3.1 États des processus 9
2.3.2 Propriétés 10
2.3.3 Ordonnancement 10
2.3.4 Métriques d'évaluation de performances 10
2.4 Taxonomie des algorithmes **11**
2.4.1 Algorithmes à permissions 11
2.4.2 Algorithmes à jeton 12
2.4.3 Algorithme centralisé 14
2.4.4 Algorithmes hybrides et hiérarchiques 14
2.5 Algorithmes de bases des contributions de la thèse **15**
2.5.1 L'algorithme de Raymond 15
2.5.2 Les algorithmes de Naimi-Tréhel 18
2.6 Extensions de l'exclusion mutuelle **20**
2.7 Conclusion **22**

2.1 Introduction

Dans un système distribué, les processus sont susceptibles d'accéder à une ou plusieurs ressources partagées (variables, pages mémoires, fichiers, etc.). Pour éviter les incohérences dues aux accès concurrents des processus, il est indispensable que ceux-ci synchronisent leurs accès. Ce problème fut identifié par Edsger Dijkstra en 1965 [Dij65]. Il l'a formalisé au travers du paradigme de l'exclusion mutuelle qui est devenu un pilier de l'algorithmique répartie. Ce paradigme assure qu'à un instant donné, au plus un processus peut exécuter une partie d'un code concurrent, appelée **section critique**. Dans le cadre de cette thèse nous nous intéresserons à l'exclusion mutuelle distribuée où les processus communiquent

par passage de messages. Ce chapitre a pour but de présenter ce problème fondamental et se découpe de la manière suivante.

La section 2.2 décrit le système que nous considérons et introduit les notations qui serviront tout au long de ce manuscrit. La section 2.3 définit formellement l'exclusion mutuelle. Ensuite la section 2.4 donne une classification des algorithmes distribués que l'on trouve dans la littérature. Deux algorithmes d'exclusion mutuelle classique à jeton sur lesquels les contributions de cette thèse sont basées (les algorithmes de Raymond [Ray89b] et de Naimi-Tréhel [NT87a, NT87b]) seront présentés en section 2.5. Enfin la section 2.6 décrit brièvement les différentes extensions de l'exclusion mutuelle et la section 2.7 donne un schéma récapitulatif de cet état de l'art.

2.2 Description et formalisation du système considéré

Nous considérons un système distribué composé d'un ensemble $\Pi = \{s_1, s_2, ..., s_N\}$ de N nœuds. Ces nœuds n'exécutant qu'un seul processus, dans la suite les mots "nœuds", "processus" et "sites" sont interchangeables. Le système est considéré statique, autrement dit Π est constant. Les différentes hypothèses sur le système sont données ci dessous.

Hypothèse 1. *Tous les sites sont fiables, ne partagent pas de mémoire et communiquent uniquement par circulation de messages au travers de liens point à point.*

Hypothèse 2. *Les liens sont fiables (ni duplication ni perte de message) et FIFO.*

Hypothèse 3. *Le temps de communication entre deux nœuds est borné mais inconnu.*

Hypothèse 4. *Le graphe de communication est complet, i.e,. chaque processus peut communiquer avec n'importe quel autre processus.*

Pour demander un accès à la section critique, un processus doit faire appel à la fonction bloquante *Request_CS* de l'algorithme permettant d'assurer l'exclusivité d'exécution sur la section critique. Lorsque son exécution de section critique se termine il doit faire appel à la fonction *Release_CS* pour indiquer à d'autres processus que la section critique est libre.

Hypothèse 5. *Un processus peut demander la section critique en appelant la primitive Request_CS si et seulement si sa précédente requête a été satisfaite par une section critique qui s'est terminée par un appel à Release_CS.*

L'hypothèse 5 implique qu'il y a au plus N requêtes pendantes dans le système.

Hypothèse 6. *Le temps d'exécution de la section critique est fini.*

Dans la suite de ce rapport, les algorithmes seront spécifiés avec un pseudo-code événementiel. A chaque réception d'un message de type $type_m$ le site exécute la fonction $Receive_type_m$. Les primitives de l'algorithme se résument donc aux primitives *Request_CS*, *Release_CS* et l'ensemble des primitives *Receive_* correspondantes aux types de messages de l'algorithme. Enfin, nous supposons que les processus possèdent un

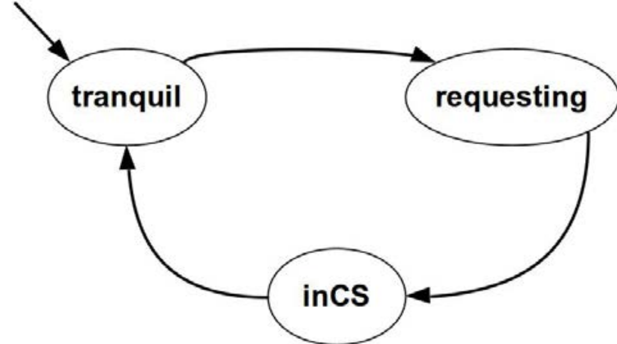


FIGURE 2.1 – Machine à états de l'exclusion mutuelle classique

verrou local permettant aux primitives de s'exécuter de manière exclusive afin d'éviter toute incohérence sur les variables locales partagées.

Nous notons le temps d'exécution du système T. T est discrétisé par rapport aux exécutions des primitives sur les différents sites et peut être vu comme un ensemble d'instants t. L'instant $t_0 \in T$ correspond au moment où le système s'initialise. Par conséquent il n'existe pas d'instant $t \neq t_0$ tel que $t < t_0$. L'instant t_{now} désigne l'instant courant.

2.3 Définition de l'exclusion mutuelle

Cette section présente les notions de bases qui serviront tout au long de ce manuscrit à savoir : la machine à états du problème, les propriétés fondamentales définissant l'exclusion mutuelle, l'ordonnancement des requêtes de section critique. Le lecteur pourra trouver une présentation complémentaire du problème de l'exclusion mutuelle dans le chapitre 2 de la thèse de J. Sopena [Sop08].

2.3.1 États des processus

Dans les modèles classiques de la littérature les processus ont trois états possibles :

- $tranquil_{s_i}^t$: le processus s_i ne demande pas la section critique à l'instant $t \in T$.
- $requesting_{s_i}^t$: le processus s_i est en attente de la section critique à l'instant $t \in T$.
- $inCS_{s_i}^t$: le processus s_i est en train d'exécuter la section critique à l'instant $t \in T$.

La machine à états est composée de trois transitions :

- de $tranquil_{s_i}$ à $requesting_{s_i}$: événement provoqué par le processus s_{s_i} lorsque celui-ci souhaite entrer en section critique en faisant appel à $Request_CS$.
- de $requesting_{s_i}$ à $inCS_{s_i}$: événement indiquant que le processus s_i peut entrer en section critique
- de $inCS_{s_i}$ à $tranquil_{s_i}$: événement provoqué par le processus s_i lorsque celui-ci sort de la section critique en faisant appel à $Release_CS$.

Le schéma de la machine à états est représenté dans la figure 2.1.

2.3.2 Propriétés

Les algorithmes d'exclusion mutuelle doivent respecter deux propriétés fondamentales :

- **la sûreté** : au plus un processus exécute la section critique.
  Formellement : $\forall t \in T \exists s_i \in \Pi$ *tel que* $inCS^t_{s_i} \Rightarrow \nexists s_j \in \Pi$ *tel que* $inCS^t_{s_j}$
- **la vivacité** : si le temps d'exécution de section critique est fini, chaque processus demandeur accédera à la section critique dans un temps fini (pas de famine).
  Formellement : $\forall s_i \in \Pi$, $\forall t \in T, \exists t' \in T$ tel que $t' > t$,

$$requesting^t_{s_i} \Rightarrow inCS^{t'}_{s_i}$$

2.3.3 Ordonnancement

Toute requête pendante doit pouvoir être différenciée d'une autre pour déterminer celle qui accédera à la section critique d'abord. Cette différenciation se fait selon un ordre total propre à l'algorithme qui définit sa politique d'ordonnancement. Pour éviter la famine toute requête pendante doit devenir en un temps fini la plus prioritaire de cet ordre. La politique d'ordonnancement de l'algorithme est primordiale pour assurer la vivacité.

L'exclusion mutuelle classique obéit à un ordonnancement du type "premier arrivé, premier servi", c'est à dire que les requêtes sont satisfaites en fonction de leur date d'enregistrement dans le système. Ce mécanisme de date peut s'implémenter entre autres par une horloge logique ou bien par un ajout dans une file d'attente gérée de manière distribuée ou centralisée. Cette politique d'ordonnancement permet d'assurer la vivacité : toute nouvelle requête q' plus récente que q sera satisfaite après q et comme nous considérons un système où le nombre de processus N est fini, ceci implique que toute requête q a un nombre fini de requêtes plus prioritaires. Ainsi toute requête deviendra en un temps fini la plus prioritaire.

2.3.4 Métriques d'évaluation de performances

En plus de leurs propriétés fondamentales, les algorithmes distribués d'exclusion mutuelle doivent être performants. D'après [Sin93], les performances d'un algorithme d'exclusion mutuelle se caractérisent par la minimisation des métriques suivantes :

- **le temps de réponse** ou **temps d'attente** : le temps entre le moment où un site envoie une requête et le moment où il est autorisé à entrer en section critique.
- **le temps de synchronisation** : le temps entre le moment où un site libère la section critique et le moment où le site suivant entre en section critique. Le temps de synchronisation désigne le temps du protocole entre deux utilisations successives de la ressource critique. Si on considère négligeable le temps de calcul, ceci se ramène au temps de communication. Dans le cadre de cette thèse, cette métrique a été remplacée par **le taux d'utilisation de la ressource** qui désigne le pourcentage de temps passé à l'utilisation de la ressource sur une période donnée.
- **la complexité en messages** : le nombre de messages nécessaires pour chaque demande d'entrée en section critique. Cette métrique n'est pas directement liée aux performances d'accès à la ressources mais reflète la capacité de l'algorithme à passer à l'échelle.

2.4 Taxonomie des algorithmes

Plusieurs publications comme Michel Raynal [Ray91b], Mukesh Singhal [Sin93] et Martin G. Velasquez [Vel93] décrivent au début des années 90 une classification des algorithmes d'exclusion mutuelle. Ces classifications, classent les algorithmes en deux grandes catégories :

- Les algorithmes à permissions : le site demandeur doit recevoir l'accord d'un ensemble d'autres sites pour accéder à la section critique.
- Les algorithmes à jeton : un jeton unique circule sur l'ensemble des sites et donne le droit à son possesseur d'entrer en section critique. L'unicité du jeton assure la sûreté.

2.4.1 Algorithmes à permissions

Dans ce type d'algorithmes, un processus désirant entrer en section critique doit envoyer une requête à un groupe de processus et attendre leur accord. En absence d'horloge globale, les requêtes sont généralement estampillées avec les horloges logiques de Lamport [Lam78]. Ces estampilles temporelles permettent d'ordonner totalement les requêtes concurrentes dans une file d'attente et ainsi assurer un accès équitable à la section critique. Les algorithmes à permissions peuvent être encore subdivisés en trois sous-catégories : permissions individuelles, permissions d'arbitre et permissions généralisées.

Permissions individuelles

Dans ce type d'algorithmes, un processus donne sa permission à un site ayant envoyé une requête, s'il n'est pas en section critique ou s'il attend la section critique mais que sa requête est moins prioritaire, i.e., moins récente d'après l'horloge logique. Un site peut donc donner plusieurs permissions à différents sites de manière simultanée. Une requête est satisfaite lorsque le site demandeur reçoit $N-1$ permissions.

La demande de permission peut se faire de deux manières :

- <u>Statique</u> : les destinataires des requêtes sont l'ensemble des nœuds. Ce principe de diffusion générale a été introduit par G. Ricart et A.K. Agrawala dans [RA81] qui améliore l'algorithme de Lamport [Lam78] au niveau de la complexité en nombre de messages ($2N-1$ au lieu de $3N-1$), en utilisant toujours les horloges de Lamport et en enlevant l'hypothèse des canaux FIFO qui est indispensable pour l'algorithme de Lamport.
- <u>Dynamique</u> : Pour diminuer le nombre de messages par requête, les processus maintiennent de l'information sur la liste des permissions. Un processus s_i peut considérer qu'il a la permission d'un site s_j, si s_j ne lui a pas envoyé de nouvelle requête. Ainsi un site n'enverra pas de requête aux sites non-demandeurs. O.S.F. Carvalho et G. Roucairol nous donnent un algorithme à permissions individuelles dynamiques dans [CR83].

Permissions d'arbitre

Dans cette approche, un site (l'arbitre) donne sa permission à un seul site à la fois. ainsi lorsqu'il donne sa permission à un site s_i, toute autre requête reçue sera mise en attente. Le site pourra alors donner sa permission à la prochaine requête lorsque s_i sortira de sa section critique. Il est ainsi impossible d'avoir deux permissions simultanées provenant d'un même site. L'algorithme de référence est celui de M. Maekawa [Mae85]. Il utilise un mécanisme de quorum. L'ensemble des sites est divisé en sous-ensembles de manière à ce que l'intersection de chaque sous-ensemble $E \subseteq \Pi$ avec un autre sous-ensemble $E' \subseteq \Pi$ ne soit pas vide. Ceci permet théoriquement de réduire la complexité en nombre de messages à $\mathcal{O}(\sqrt{N})$ alors qu'elle est en moyenne de $\mathcal{O}(N)$ dans les algorithmes à permissions individuelles. Cependant, le fait qu'un arbitre ne donne sa permission qu'à un seul demandeur, peut conduire à des situations d'interblocages. De plus, la construction des ensembles est un problème NP-complet.

Permissions généralisées

Ces algorithmes combinent les deux stratégies décrites ci-dessus. Ils ont été introduits par Sanders [San87] et ont été repris par Singhal [Sin92]. Ainsi, chaque site s_i maintient un sous-ensemble de sites $E \subseteq \Pi$. Lorsque s_i reçoit une demande de la part d'un site $s_j \in E$, s_i donnera une permission de type arbitre et dans le cas contraire ($s_j \notin E$) une permission de type individuel. Ces algorithmes permettent de résoudre le problème d'inter-blocage de l'algorithme de Maekawa.

2.4.2 Algorithmes à jeton

Dans ces algorithmes, un seul processus obtient une permission globale qui est matérialisé par la possession d'un jeton. La présence d'un unique jeton assure de manière triviale la sûreté. Le jeton se transmet de processus en processus. Il existe deux sous catégories d'algorithmes à jeton :

- Non-structurés : pas de topologie logique imposée sur les sites.
- Structurés : les sites sont organisés selon une topologie bien particulière (un arbre ou un anneau par exemple).

Algorithmes à jeton structurés

Une topologie logique relie les sites via des liens. Cette topologie peut évoluer au cours de l'exécution de l'algorithme, dans ce cas cette topologie est dite dynamique et dans le cas contraire elle est dite statique, i.e., les liens logiques restent les mêmes durant toute l'exécution et seule leur direction peut changer.

- **Topologie statique** : Nous pouvons distinguer dans la littérature trois types de topologies statiques :
  - ∗ Anneau : Dans les algorithmes [Lan78] et [Mar85] le jeton circule le long d'un anneau unidirectionnel. L'anneau permet d'assurer la propriété de vivacité. En effet, étant donné que le jeton circule le long de l'anneau, il est certain qu'un processus aura le jeton à un moment donné. L'algorithme de J. Martin [Mar85]

adapte sa complexité en messages en fonction de la charge (i.e., la quantité de requêtes pendantes) : en faible charge la complexité est de $\mathcal{O}(N)$ alors qu'en forte charge la complexité devient constante.

  * Arbre : Le nœud racine de l'arbre est le site qui détient le jeton. Les liens sont orientés vers la racine de manière à ce que lorsqu'un site demande le jeton, cette demande soit propagée jusqu'à la racine. Ainsi, un lien entre deux nœuds indiquera toujours la direction de la racine, i.e., du site possédant le jeton. La complexité moyenne de ces algorithmes est de $\mathcal{O}(logN)$. Dans cette catégorie nous pouvons citer les algorithmes de Nielsen-Mizuno [NM91] et de Raymond [Ray89b]. Nos algorithmes des chapitres 4 et 5 se basant sur l'algorithme de Raymond [Ray89b], la section 2.5.1 détaillera ce dernier.
  * Graphe : Tous les nœuds sont placés dans une topologie arbitraire. Contrairement à la topologie de l'arbre, les cycles de la topologie sont possibles. Les requêtes d'un site se propagent sur le réseau par un mécanisme d'inondation. Le jeton est ensuite retransmis de site en site jusqu'à son destinataire. [CSL90] et [HPR88] sont deux exemples de ce type d'algorithme.

- **Topologie dynamique** : Dans ce type d'algorithme, la topologie logique change au cours de l'exécution. La dynamicité de la structure reflète l'historique des accès à la section critique permettant ainsi d'accroître les performances pour les sites qui ont un accès fréquent à la section critique. Les algorithmes de référence de cette catégorie sont les algorithmes de Naimi-Tréhel [NT87a, NT87b] et Naimi-Tréhel-Arnold [NTA96]. Leur topologie est un arbre dynamique ou plus exactement une forêt dynamique. Chaque nœud maintient deux listes chaînées distribuées : $next$ pour sauvegarder l'ordre des requêtes pendantes et $father$ (ou $last$) qui indique le chemin vers le dernier demandeur. Ainsi le dernier demandeur est la racine d'un arbre de la forêt. Ces algorithmes ont une complexité moyenne en nombre de messages de $\mathcal{O}(logN)$. Notons que l'algorithme de Naimi-Tréhel avec files locales [NT87b] permet d'avoir une complexité constante en cas de forte charge. Ces algorithmes servant de base à l'algorithme du chapitre 7 sont détaillés en section 2.5.2.

Algorithmes à jeton non-structurés

Les algorithmes non-structurés n'imposent pas de topologie particulière : le graphe logique correspond à un graphe complet. Les requêtes se font par diffusion, i.e., un site demandeur enverra une demande de jeton à un ensemble de nœuds. L'ordre des requêtes peut être sauvegardé dans une file d'attente incluse dans le jeton. De façon identique aux algorithmes à permissions individuelles, il existe deux sous-classes d'algorithmes :.

- **Algorithmes statiques** : Ces algorithmes ne sauvegardent pas l'historique des différentes exécutions en section critique. Chaque requête est ainsi diffusée à l'ensemble des nœuds. Nous pouvons donner l'exemple de l'algorithme de Susuki-Kasami [SK85]. Le site possédant le jeton, une fois sorti de sa section critique, le renverra au site le plus ancien dont la requête n'a pas encore été satisfaite.
- **Algorithmes dynamiques** : pour réduire le nombre de messages et éviter de solliciter les sites qui utilisent peu souvent la section critique, les sites maintiennent

un historique des derniers demandeurs. Une requête n'est alors diffusée qu'aux sites présents dans l'historique. Chang, Singhal et Liu [CSL91] ont proposé en 1991 un algorithme de ce type en améliorant l'algorithme de Susuki-Kasami [SK85].

2.4.3 Algorithme centralisé

Des solutions centralisées existent où les différents processus demandant l'accès à la section critique s'adressent à un site coordinateur. Le coordinateur a une connaissance globale de l'ensemble des requêtes pendantes et ordonne les accès à la section critique. Les processus entrent en section critique uniquement sur l'accord du coordinateur. Bien que cette solution ait une complexité en messages constante, son principal défaut réside dans le fait qu'il y ait un goulot d'étranglement au niveau de ce coordinateur ce qui rend un passage à l'échelle difficile lorsque la charge augmente. En effet la capacité à gérer les requêtes dépend beaucoup de la puissance de traitement de la machine centrale.

Il est à noter que l'algorithme centralisé est un cas particulier d'un algorithme à jeton à arbre statique de hauteur 1 (topologie en étoile) où le nœud central fait office de coordinateur.

2.4.4 Algorithmes hybrides et hiérarchiques

Ces algorithmes répartissent le graphe de communications sur plusieurs niveaux hiérarchiques. Ainsi les nœuds d'un niveau k sont partitionnés en plusieurs groupes. Chaque groupe est souvent représenté par un nœud mandataire (le proxy) qui fait l'intermédiaire avec le niveau $k+1$. Un nœud de niveau k ne peut s'adresser qu'aux nœuds de son groupe et aux nœuds d'un groupe de niveau $k+1$ s'il est mandataire. A l'instar d'un routeur dans un réseau, les mandataires jouent le rôle de coordinateurs vis-à-vis des autres membres du groupe. Autrement dit, les mandataires d'un niveau k font aussi partie du niveau $k+1$. Souvent, on se limite à seulement deux niveaux. Une des caractéristiques de ces algorithmes est que chaque niveau est cloisonné et peut être associé à un type d'algorithme différent. Ceci qui permet de faire cohabiter par exemple un algorithme à jeton avec un algorithme à permissions. Cependant, comme dans le modèle centralisé, les mandataires représentent un goulot d'étranglement et sont sensibles aux pannes.

Voici trois exemples représentatifs de ce type d'algorithme :

- **Erciyes** [Erc04] : Cet algorithme s'exécute sur deux niveaux hiérarchiques. Le premier niveau utilise un algorithme centralisé avec un coordinateur qui fait office d'intermédiaire avec le niveau supérieur. Au niveau supérieur les coordinateurs du premier niveau sont organisés en anneau. Lorsqu'un coordinateur reçoit une requête d'un de ses sites, il envoie cette requête à son successeur dans l'anneau. La requête peut être satisfaite si cette dernière fait un tour complet de l'anneau (adaptation de l'algorithme de Ricart-Agrawala [RA81] sur un anneau). L'information sur l'ensemble des requêtes pendantes est entièrement répartie sur les coordinateurs.
- **Bertier-Arantes-Sens** [BAS04, BAS06] : La topologie sous-jacente du système est une grille (un ensemble de clusters). Le principe est de privilégier les demandes locales au sein du cluster possédant le jeton au détriment des demandes des autres clusters. Pour éviter la famine, un mécanisme de préemption a été mis en place :

au-delà d'un certain seuil, dans le cluster qui possède le jeton, les requêtes internes ne sont plus prioritaires sur les requêtes externes. Les deux niveaux se basent sur l'algorithme de Naimi-Tréhel [NT87a].

- **Sopena et al.** [SLAAS07] : L'algorithme est sur deux niveaux hiérarchiques : un niveau coordinateur (niveau 2) et un niveau pour les nœuds applicatifs (niveau 1). Il existe ainsi deux types de verrou : inter pour le niveau 2, et intra pour un groupe de nœuds au niveau 1. L'algorithme est générique, c'est-à-dire qu'il peut combiner n'importe quel type d'algorithme. Si l'on considère des algorithmes à jeton, le nombre total de jetons est de $k + 1$ (k intra et 1 inter).

Les algorithmes hiérarchiques trouvent leur intérêt dans les environnements hétérogènes en termes de latence réseau. Ils sont principalement utilisés pour des applications qui s'exécutent sur une grille ou un ensemble de clusters éloignés géographiquement.

2.5 Algorithmes de bases des contributions de la thèse

Cette section détaille les trois algorithmes d'exclusion mutuelle classique sur lesquels les contributions de cette thèse sont basées. Ces algorithmes sont l'algorithme de Raymond [Ray89b] et les deux versions de l'algorithme de Naimi-Tréhel [NT87a, NT87b]. Ces algorithmes sont des algorithmes à jeton circulant dans un arbre statique pour Raymond et une forêt d'arbres dynamiques pour Naimi-Tréhel. Le choix de ces algorithmes s'explique par leurs bonnes performances en termes de complexité en messages qui est en moyenne logarithmique.

2.5.1 L'algorithme de Raymond

Présentation

L'algorithme de Raymond [Ray89b] repose sur la circulation d'un jeton entre processus dans un arbre statique : seule la direction des liens change de sens pendant l'exécution de l'algorithme. Les nœuds forment ainsi un arbre orienté dont la racine est toujours le possesseur du jeton et donc le seul à pouvoir entrer en section critique. Cet arbre est utilisé pour acheminer les requêtes en direction du jeton : à la réception d'un message de requête sur s_i ou bien à la création d'une nouvelle requête de s_i, ce dernier stocke cette requête dans une FIFO locale et retransmet le message à son père. Cependant, pour des raisons d'efficacité, les requêtes ne sont pas retransmises par un nœud ayant déjà fait une demande (la file locale n'est pas vide). Un nœud devient *requesting* à partir du moment où sa file locale est non vide et toute requête qu'un site retransmet le fait en son nom et non au nom de l'initiateur de la requête. Ainsi chaque nœud fait office de mandataire pour ses fils directs. Lorsque la racine sort de la section critique, elle envoie le jeton au premier élément de sa file locale et prend ce dernier comme père. Quand un processus reçoit le jeton, il dépile la première requête de sa file locale. Si cette requête est de lui, il peut exécuter la section critique ; sinon il retransmet le jeton au premier élément de sa file locale, le choisit comme nouveau père et si des requêtes demeurent pendantes, lui envoie

```
1  Local variables :
2  begin
3      father : site ∈ Π or nil;
4      Q : FIFO queue of sites ;
5      state ∈ {tranquil, requesting, inCS}
6  end
7  Request_CS()
8  begin
9      if father ≠ nil then
10         add self in Q;
11         if state = tranquil then
12             state ← requesting;
13             send Request to father;
14         wait(father = nil);
15     state ← inCS;
16     /* CRITICAL SECTION                    */
17 end
18 Release_CS()
19 begin
20     state ← tranquil;
21     if Q ≠ ∅ then
22         father ← dequeue(Q);
23         send Token to father;
24         if Q ≠ ∅ then
25             state ← requesting;
26             send Request to father;
27 end
28 Initialization
29 begin
30     Q ← ∅;
31     state ← tranquil;
32     father ← according to the initial topology;
33 end
34 Receive_Request() from s_j
35 begin
36     if father = nil and state = tranquil then
37         father ← s_j;
38         send Token to father;
39     else if father ≠ s_j then
40         add s_j in Q;
41         if state = tranquil then
42             state ← requesting;
43             send Request to father;
44 end
45 Receive_Token() from s_j
46 begin
47     father ← dequeue(Q);
48     if father = self then
49         father ← nil;
50         notify(father = nil);
51     else
52         send Token to father;
53         if Q ≠ ∅ then
54             state ← requesting;
55             send Request to father;
56         else
57             state ← tranquil;
58 end
```

FIGURE 2.2 – Algorithme de Raymond

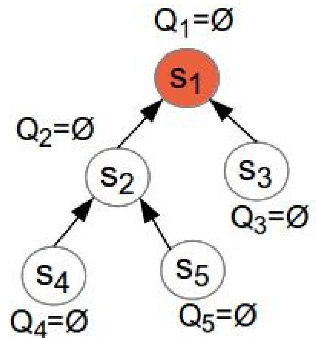
une requête pour récupérer le jeton. La figure 2.2 montre le pseudo-code de l'algorithme de Raymond.

Exemple

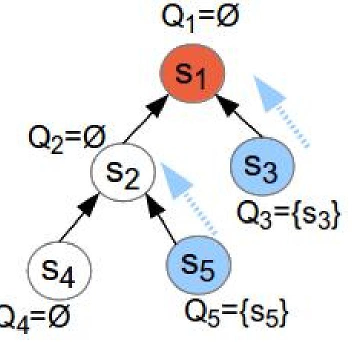
La figure 2.3 présente un exemple commenté d'exécution de l'algorithme de Raymond dans un système distribué de 5 processus.

Avantages

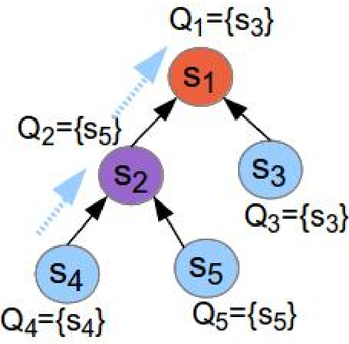
Sa complexité est en moyenne logarithmique par rapport au nombre de nœuds N lorsque la charge est faible et devient constante lorsque la charge augmente. Cette économie de messages est possible grâce au fait qu'un site ne retransmet pas de message de requête à son père si sa file locale contient des requêtes. L'autre avantage de l'algorithme de Raymond est que il est possible de considérer un graphe de communication non complet. Ainsi, la topologie logique peut correspondre complètement ou partiellement à la topologie physique du réseau sous-jacent.



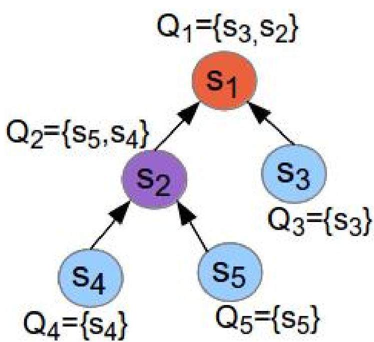
État initial : s_1 possède le jeton et est en section critique.



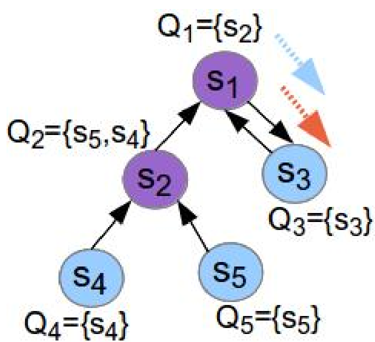
Étape 1 : s_5 et s_3 entrent en état *requesting*. Des messages de requête sont envoyés aux pères respectifs.



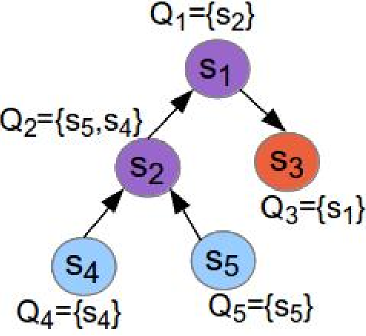
Étape 2 : s_2 et s_1 reçoivent les requêtes : ils les ajoutent dans leur file locale. s_2 devient *requesting* pour la requête de s_5 et envoie une requête à s_1. De plus, s_4 entre en état *requesting* et envoie une requête à s_2.



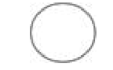
Étape 3 : s_2 reçoit la requete de s_4 : aucun message n'est retransmis car ceci a déjà été fait pour s_5.



Étape 4 : s_1 sort de la section critique, dépile le premier élément de sa file locale et envoie le jeton à s_3. Comme Q_1 n'est pas vide, s_1 envoie également, un message de requête pour la requête de s_2.



Étape 5 : s_3 reçoit le jeton et entre en section critique puisque s_3 est le premier élément de sa file locale. s_3 reçoit ensuite le message de requête et ajoute s_1 dans Q_3.



état *requesting* état *requesting* par délégation état *tranquil* porteur du jeton

message de requête message de jeton lien structurel *father*

FIGURE 2.3 – Exemple d'exécution de l'algorithme de Raymond

2.5.2 Les algorithmes de Naimi-Tréhel

Présentation

Les deux versions de l'algorithme de Naimi-Tréhel (file distribuée [NT87a, NTA96] et files locales [NT87b]) se basent sur la circulation d'un jeton dans une structure de forêt d'arbres dynamiques. Sur le même principe que l'algorithme de Raymond, les nœuds envoient toujours leurs requêtes à leur père. Chaque site s_i possède deux variables :

- *last* : indique l'identifiant du dernier demandeur de section critique du point de vue de s_i, autrement dit l'identifiant du site de l'initiateur du dernier message de requête reçu par s_i. Cette variable permet de définir une structure d'arbre dynamique où la racine d'un arbre ($last = nil$) est le dernier site qui obtiendra le jeton parmi l'ensemble des sites ayant une requête pendante. Initialement, la racine est le porteur du jeton. Lorsqu'un site souhaite entrer en section critique il envoie un message de requête au site pointé par son *last* qui correspond au porteur de jeton le plus probable car il est de son point de vue le demandeur le plus récent.
- *next* : cette variable permet de stocker l'ordre dans lequel les requêtes seront satisfaite. Elle diffère entre les deux versions. Dans la version à file distribuée [NT87a, NTA96] c'est un pointeur de site indiquant le prochain porteur du jeton. Dans la version à files locales, cette variable est une FIFO locale.

Version avec file distribuée [NT87a, NTA96] : En recevant un message de requête, la requête est transmise au site pointé par la variable *last* si cette dernière est différente de *nil*. Si au contraire *last* est égal à *nil*, le site receveur est donc une racine : si il possède le jeton sans être en section critique, le jeton est directement transmis à l'initiateur de la requête sinon il met à jour son pointeur *next* sur ce dernier. À la réception du jeton, un site passe directement en section critique. Le pseudo-code de cet algorithme est donné en figure 2.4.

Version avec files locales [NT87b] : L'objectif de cette version est de réduire le nombre de messages de requête. À l'instar de l'algorithme de Raymond il est inutile de transmettre une requête au *last* si le site courant est lui-même en état *requesting* signifiant qu'il aura le jeton dans un temps fini. La file distribuée des *next* est donc remplacée par une FIFO locale. Un site restera une racine tant qu'il est en état *inCS* ou en état *requesting*. Par conséquent, lorsqu'un site reçoit un message de requête et qu'il est dans un de ces deux états, cette requête est ajoutée dans la FIFO locale. À la libération du jeton, la file locale est transmise dans le jeton jusqu'au prochain porteur. À sa réception, le nouveau porteur fusionne sa file locale avec celle du jeton. Afin de respecter la propriété de vivacité, les requêtes présentes dans la file du jeton précèdent celles de la file du nouveau porteur. L'ensemble des files locales *next* forment donc une file distribuée virtuelle pour laquelle il est possible de trouver une file distribuée équivalente dans la première version de l'algorithme. Le pseudo-code de cet algorithme est donné en figure 2.5.

```
1  Local variables :
2  begin
3     state ∈ {tranquil, requesting, inCS}
      next : site ∈ Π or nil;
4     last : site ∈ Π or nil;
5  end
6  Initialization
7  begin
8     state ← tranquil;
9     next ← nil;
10    if self = elected_node then
11       last ← nil;
12    else
13       last ← elected_node;
14 end
15 Request_CS()
16 begin
17    state ← requesting;
18    if last ≠ nil then
19       send Request(self) to last;
20       last ← nil;
21       wait(state = inCS);
22    state ← inCS;
23    /* CRITICAL SECTION */
24 end
25 Release_CS()
26 begin
27    state ← tranquil;
28    if next ≠ nil then
29       send Token to next;
30       next ← nil;
31 end
32 Receive_Request(requester : site) from s_j
33 begin
34    if last = nil then
35       if state ≠ tranquil then
36          next ← requester;
37       else
38          send Token to requester;
39    else
40       send Request(requester) to last;
41    last ← requester;
42 end
43 Receive_Token() from s_j
44 begin
45    state ← inCS;
46    notify(state = inCS);
47 end
```

Figure 2.4 – Algorithme de Naimi-Tréhel avec file distribuée [NT87a]

```
1  Local variables :
2  begin
3     state ∈ {tranquil, requesting, inCS}
      next : FIFO queue of sites;
4     last : site ∈ Π or nil;
5  end
6  Initialization
7  begin
8     state ← tranquil;
9     next ← ∅;
10    if self = elected_node then
11       last ← nil;
12    else
13       last ← elected_node;
14 end
15 Request_CS()
16 begin
17    state ← requesting;
18    if last ≠ nil then
19       send Request(self) to last;
20       last ← nil;
21       wait(state = inCS);
22    state ← inCS;
23    /* CRITICAL SECTION */
24 end
25 Release_CS()
26 begin
27    state ← tranquil;
28    if next ≠ ∅ then
29       last ← getLast(next);
30       site next_holder ← dequeue(next);
31       send Token(next) to next_holder;
32       next ← ∅;
33 end
34 Receive_Request(requester : site ) from s_j
35 begin
36    if last = nil then
37       if state ≠ tranquil then
38          add requester in next;
39       else
40          send Token(∅) to requester;
41          last ← requester;
42    else
43       send Request(requester) to last;
44       last ← requester;
45 end
46 Receive_Token(remote_queue : Queue ) from s_j
47 begin
48    state ← inCS;
49    next ← remote_queue + next;
50    notify(state = inCS);
51 end
```

Figure 2.5 – Algorithme de Naimi-Tréhel avec files locales [NT87b]

Exemple

La figure 2.6 présente une comparaison d'exécution des deux versions de l'algorithme de Naimi-Tréhel dans un système distribué de 5 processus. Il permet d'illustrer les différences entre les deux versions. Nous pouvons nous rendre compte à l'étape de deux de l'exemple que la version à file locale permet d'économiser deux messages de requête. a l'étape 3, nous pouvons constater à file d'attente globale équivalente, que la version à files locales réduit de manière significative le nombre de liens logiques.

Avantages

Contrairement à l'algorithme de Raymond, un envoi de jeton donnera directement un accès à la section critique par le prochain porteur ce qui réduit les transferts réseau, permettant ainsi d'augmenter le taux d'utilisation de la ressource critique. De plus, un site ayant besoin peu souvent de la section critique n'est pas sollicité pour transmettre des messages de requête. Dans les deux versions la complexité en messages est en moyenne logarithmique par rapport à N, mais la version avec file locale a une complexité constante lorsque la charge augmente.

2.6 Extensions de l'exclusion mutuelle

Le problème de l'exclusion mutuelle classique permet de gérer un accès FIFO sur une ressource partagée. Cependant, diverses extensions de ce problème fondamental peuvent être trouvées dans la littérature.

Exclusion mutuelle à priorité : les requêtes sont associées à un niveau de priorité et l'objectif principal est de respecter le plus possible cet ordre (éviter les inversions). Ce type d'extension permet de considérer différents types d'utilisation (ex : utilisateur et administrateur) ou différents types de client (ex : Nuages). Cette extension fait partie d'une des contributions de cette thèse (chapitre 4). Le chapitre 3 présente un état de l'art de l'exclusion mutuelle à priorité.

Exclusion mutuelle à contraintes de temps : les requêtes doivent être satisfaites avant une date limite requise. Ce type d'extension applicable aux systèmes temps-réel à passage de messages a été défini au cours de cette thèse dans la contribution [CCgrid12] et est détaillée dans le chapitre 5.

Exclusion mutuelle généralisée : Dans cette extension, on propose de généraliser le problème initial :

1. soit à plusieurs exemplaires de la ressource et ainsi autoriser plusieurs processus à être en section critique en utilisant un ou plusieurs exemplaires de la ressource. Le problème du k-mutex, est un exemple de ce type de problème.
2. soit à plusieurs ressources en un seul exemplaire et autoriser plusieurs processus à être en section critique s'ils utilisent des ensembles de ressources disjoints. Cette

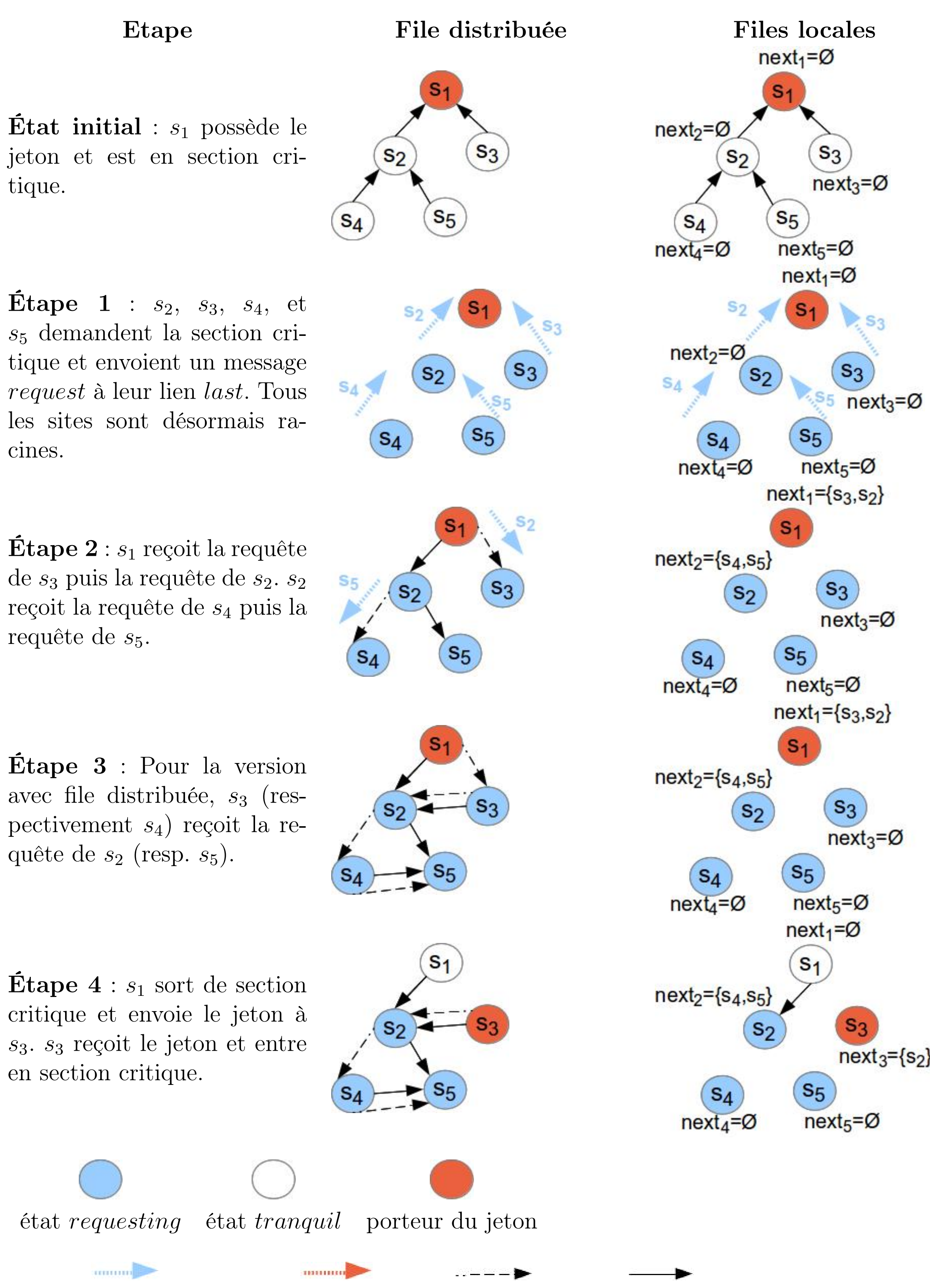


FIGURE 2.6 – Exemple d'exécution des deux versions de l'algorithme de Naimi-Tréhel

extension aussi connu sous le nom du "cocktail des philosophes" a été introduite par Chandy et Misra en 1984 [CM84].

3. soit l'union des deux extensions précédentes en considérant plusieurs ressources en plusieurs exemplaires.

Ces extensions font l'objet d'une contribution de cette thèse et le lecteur pourra trouver une description plus détaillée dans le chapitre 6.

Exclusion mutuelle de groupe : introduit par Joung en 1998 [Jou98] où des groupes de processus du système accèdent simultanément à une ressource partagée. Ce problème est aussi connu sous le nom de la "conversation des philosophes" où les philosophes passent leur temps à penser tout seul ou bien à discuter ensemble dans une salle de réunion sur un sujet particulier. Lorsqu'ils cessent de penser (souhaitent entrer en section critique), ils choisissent un sujet de conversation de leur choix pour en discuter dans la salle de réunion. Comme il n'y a qu'une seule salle (la ressource partagée), la conversation peut commencer si et seulement si la salle est vide. Des philosophes intéressés par la discussion peuvent la rejoindre en cours de route. Le temps de conversation est supposé fini pour assurer que la salle sera à terme disponible pour une autre conversation. La propriété de sûreté assure qu'il y ait au plus un seul sujet de conversation dans la salle. La propriété de vivacité assure que tout philosophe accédera à la salle en temps fini avec le sujet de conversation qui l'intéresse. Enfin, on peut ajouter une propriété de concurrence assurant que tout philosophe intéressé par une conversation en cours pourra accéder à la salle de réunion. Dans cette extension nous pouvons référencer les articles [WJ00], [Jou03], [Vid03], [JPT03], [MN06] et [AWCR13].

Combinaison : Toutes ces extensions sont orthogonales et peuvent être combinées. Nous pouvons citer l'algorithme de Swaroop-Singh qui combine l'exclusion mutuelle de groupe avec l'exclusion mutuelle à priorité [SS07].

2.7 Conclusion

L'exclusion mutuelle classique se définit sur deux propriétés : la sûreté et la vivacité. Il existe une grande diversité d'algorithmes d'exclusion mutuelle pour les systèmes distribués. Ces algorithmes reposent soit sur l'obtention de permissions soit sur la circulation d'un jeton.

Les algorithmes à base de permissions sont plus coûteux en nombre messages du fait qu'un site demandeur fait une diffusion sur un ensemble de sites. Leur complexité est en général de $\mathcal{O}(N)$ mais celle-ci peut être réduite si les processus s'adressent non pas à la totalité des processus mais à un sous-ensemble. Il est possible pour cela d'utiliser les quorums de Maekawa [Mae85] qui restent cependant assez complexes à construire. Le mécanisme d'ordonnancement des requêtes dans les algorithmes à permissions est principalement basé sur les horloges de Lamport [Lam78] et garantit ainsi la politique d'ordonnancement du « premier arrivé premier servi » assurant une équité.

Le nombre de messages des algorithmes à jeton est réduit lorsqu'ils ne se basent pas sur un mécanisme de diffusion comme [SK85] mais sur une topologie logique. Le fait de rendre un algorithme dynamique améliore ses performances car un site demandeur s'adressera aux sites qui ont le plus de chances d'avoir le jeton. Les sites qui ont rarement besoin de la section critique sont donc mis à l'écart des communications et ne servent pas d'intermédiaires. La propriété de sûreté ne se résume qu'à l'unicité du jeton et la vivacité est respectée grâce à la propriété de la structure utilisée pour la topologie des sites.

De nombreux articles cités dans ce chapitre prouvent que le problème originel de l'exclusion mutuelle a été grandement étudié. Il s'agit d'un problème simple qui n'est cependant pas adapté à certaines utilisations. Nous avons ainsi présenté plusieurs extensions (priorité, nombre de ressources, priorité, ...). La suite de ce manuscrit se propose d'étudier et d'apporter des solutions nouvelles à ces différents types de problème.

Pour conclure ce chapitre la figure 2.7 synthétise et classifie des algorithmes d'exclusion mutuelle classique.

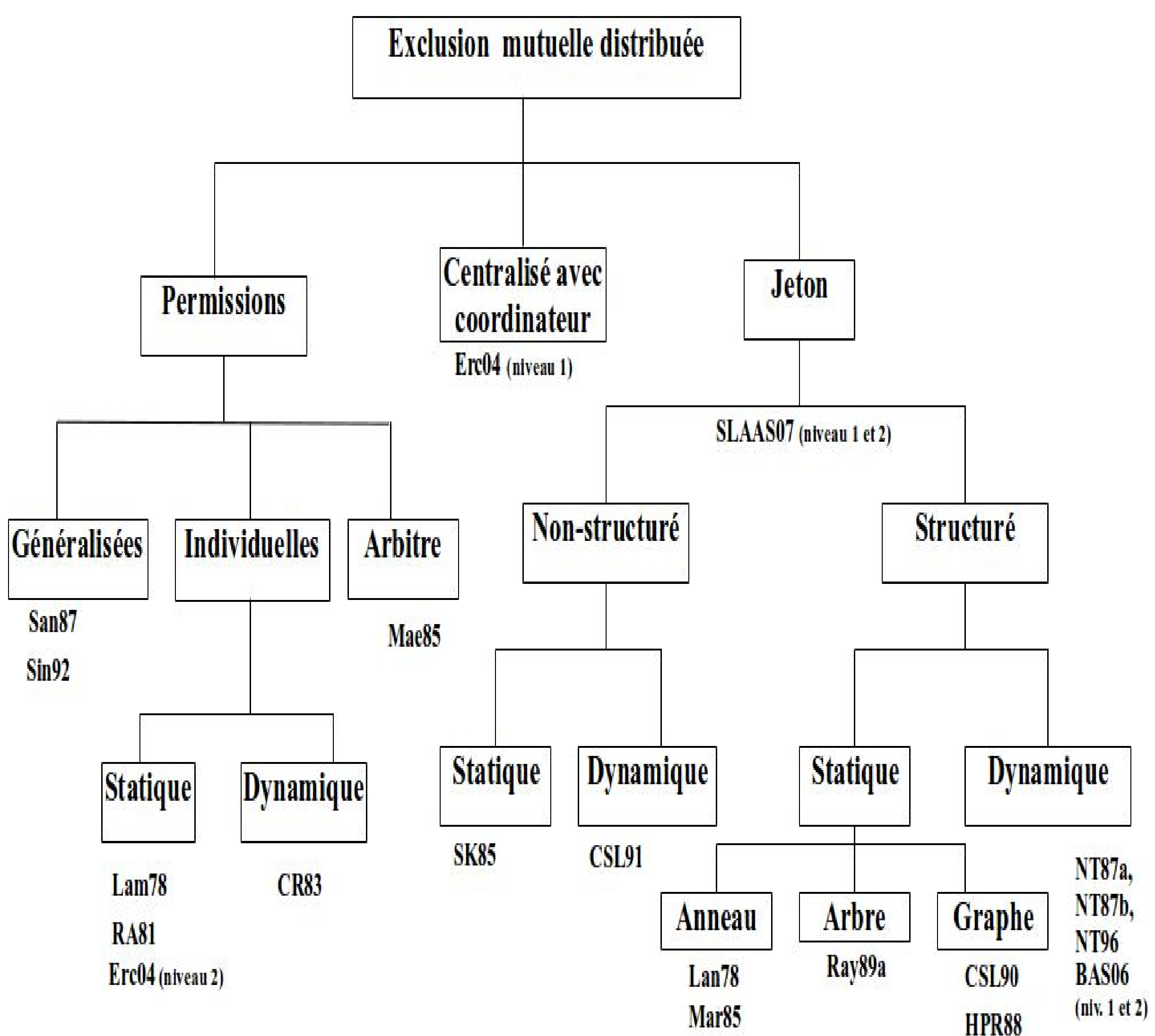


Figure 2.7 – Schéma récapitulatif de l'état de l'art de l'exclusion mutuelle

Chapitre 3

Exclusion mutuelle à priorités

Sommaire

3.1 Introduction **25**
3.2 Extension du modèle du système considéré **26**
3.3 Taxonomie des algorithmes à priorité **26**
3.3.1 Priorités statiques 26
3.3.2 Priorités dynamiques 27
3.4 Description des algorithmes principaux **27**
3.4.1 Algorithme de Mueller 27
3.4.2 Algorithme de Chang 28
3.4.3 Algorithme de Kanrar-Chaki 29
3.5 Conclusion **29**

3.1 Introduction

Nous avons vu dans le chapitre précédent que l'exclusion mutuelle classique satisfaisait les requêtes dans un ordre "premier arrivé, premier servi". Or dans certains systèmes distribués, cette approche n'est pas adaptée lorsque l'on doit gérer des processus avec des priorités différentes. Le modèle de l'exclusion mutuelle a donc été étendu pour gérer ce type d'ordonnancement et des algorithmes distribués ont été proposés (généralement une extension ou une combinaison des algorithmes de base présentés en section 2.4). L'exclusion mutuelle à priorités ajoute une contrainte supplémentaire dans l'ordonnancement des requêtes : les requêtes doivent être satisfaites dans l'ordre de leur priorité dans la mesure du possible. Cette priorité est généralement représentée par une valeur entière : plus cette valeur est élevée, plus la priorité de la requête est haute. La notion de priorité induit donc des ré-ordonnancements dynamiques de la file des requêtes pendantes en fonction des niveaux de priorité des nouvelles requêtes.

Le non respect de l'ordre des priorités introduit des inversions de priorités. Une inversion de priorité est le fait qu'une requête soit satisfaite avant une requête pendante plus prioritaire. Une définition plus approfondie de l'inversion de priorité est donnée en section 4.2.

Le strict respect de l'ordre des priorités peut cependant induire des famines, c'est à dire un temps infini pour qu'un processus obtienne la section critique violant ainsi la propriété de vivacité. La famine peut apparaître lorsqu'un processus de haute priorité émet des requêtes en permanence empêchant ainsi les autres processus de plus basses priorités d'accéder à la section critique. Ainsi, pour éviter ceci, nous verrons qu'il existe des algorithmes à priorités dynamiques qui augmentent les priorités des requêtes en attente pendant l'exécution de l'algorithme. Toute requête pourra alors atteindre à terme la priorité maximale et ainsi assurer qu'elle sera satisfaite en temps fini.

La section 3.2 étend le modèle du système considéré décrit en section 2.2. Dans la section 3.3 nous présenterons les deux familles d'algorithmes d'exclusion mutuelle à priorité (statique et dynamique). La section 3.4 décrira trois algorithmes (Mueller, Chang et Kanrar-Chaki) qui nous serviront de base de comparaison.

3.2 Extension du modèle du système considéré

Nous définissons l'ensemble des priorités $\mathcal{P} \subset \mathbb{N}$ de P éléments. Si deux priorités $p_i, p_j \in \mathcal{P}$ et $p_i < p_j$ alors la priorité p_j est plus importante que p_i. Nous notons $p_{min} \in \mathcal{P}$ la priorité minimale du système où $\nexists p \in \mathcal{P}, p < p_{min}$. De même, nous notons $p_{max} \in \mathcal{P}$ la priorité maximale du système où $\nexists p \in \mathcal{P}, p_{max} < p$.

La demande de section critique se fait désormais en ajoutant un paramètre $p \in \mathcal{P}$, $p_{min} \leq p \leq p_{max}$ à la primitive *Request_CS*, qui devient alors *Request_CS*(p).

3.3 Taxonomie des algorithmes à priorité

La taxonomie des algorithmes d'exclusion mutuelle à priorité peut se résumer en deux familles : priorité statique et priorité dynamique.

3.3.1 Priorités statiques

Dans cette famille d'algorithme, la priorité d'une requête reste la même jusqu'à sa satisfaction. Elle respecte scrupuleusement l'ordre des priorités impliquant aucune inversion de priorité. Cependant une famine pour les plus basses priorités reste possible si des requêtes de priorité haute demandent en permanence la section critique. Nous citons ci-dessous quelques algorithmes à priorité statique.

- L'**algorithme de Goscinksi** [Gos90] est une extension de l'algorithme de Suzuki-Kasami [SK85]. Cet algorithme se base sur un mécanisme de diffusion (complexité en nombre de messages de $\mathcal{O}(N)$). Les requêtes pendantes sont enregistrées dans une file globale et sont ordonnée en fonction de leur priorité. La file est incluse dans le jeton.
- L'**algorithme de Housni-Tréhel** [HT01] adopte une approche hiérarchique où les nœuds sont groupés par priorité. Dans chaque groupe, un nœud routeur représente le groupe auprès des autres groupes. Les processus d'un même groupe sont organisés en arbre statique comme dans l'algorithme de Raymond [Ray89b] et les routeurs appliquent entre eux l'algorithme de Ricart-Agrawala [RA81]. L'inconvénient est

que chaque processus ne peut faire des requêtes qu'en gardant toujours la même priorité (celle de son groupe).

- **Johnson-Newman-Wolfe** ont décrit trois algorithmes dans [JNW96]. Deux d'entre eux utilisent une technique de compression de chemin de Li-Hudak [LH89] pour un accès rapide et un faible coût en messages. Le troisième algorithme étend l'algorithme de Raymond [Ray89b]. Chaque processus maintient une file locale de requêtes reçues triée par ordre de priorité. Seules les nouvelles requêtes avec une priorité plus importante sont retransmises au père dans la structure logique.
- L'**algorithme de Mueller** [Mue98] étend l'algorithme de Naimi-tréhel avec files locales [NT87b]. Une description plus détaillée de cet algorithme est donnée en section 3.4.1.

3.3.2 Priorités dynamiques

Dans la famille de priorité dynamique, la priorité d'une requête est incrémentée au cours du temps pour assurer la vivacité. Généralement les priorités des requêtes pendantes augmentent à chaque enregistrement d'une nouvelle requête de plus haute priorité. Ainsi, l'émission de requêtes de plus haute priorité a pour effet d'augmenter les petites priorités ce qui à terme amène l'accès à la section critique.

L'inconvénient majeur d'une telle approche est que des inversions de priorités peuvent désormais se produire. En effet, l'incrémentation des priorités amènera au fait que des requêtes à petites priorités entreront en section critique avant d'autres requêtes de priorité supérieure. De plus ces algorithmes ont un surcoût en messages dû aux mises à jour des priorités dans le système. Les deux algorithmes de référence de cette famille sont :

- l'**algorithme de Chang** [Cha94]
- l'**algorithme de Kanrar-Chaki** [KC10]

Ces deux algorithmes se basent sur la circulation d'un jeton dans un arbre statique comme l'algorithme de Raymond [Ray89b]. Ils sont décrits de manière détaillée respectivement en section 3.4.2 et 3.4.3.

3.4 Description des algorithmes principaux

3.4.1 Algorithme de Mueller

L'algorithme de Mueller [Mue98] se base sur l'algorithme de Naimi-Tréhel qui utilise un arbre dynamique comme structure logique pour la transmissions des requêtes. Comme la file des requêtes pendantes peut être réordonnancée dynamiquement à cause d'éventuelles nouvelles requêtes de priorité plus importante, la version à file distribuée [NT87a] n'est pas adaptée. En effet, elle rend difficile l'insertion de requêtes au milieu de la file. Le coût de maintien de la cohérence de cette file devient alors prohibitif. L'algorithme de Mueller se base par conséquent sur la version à files locales [NT87b]. Les files locales sont triées en fonction des priorités des requêtes qu'elles contiennent et la concaténation de ces files permet de former une file d'attente virtuelle. La retransmission de requêtes se fait à l'instar de l'algorithme de Raymond [Ray89b], où un site retransmet une requête à son

père que si sa file locale ne contient aucune requête de plus forte priorité. Ce mécanisme est récursif jusqu'à la racine qui est le détenteur du jeton. À chaque libération du jeton, la file locale du site correspondant est transmise dans le jeton et le pointeur de structure indiquant la direction du jeton est mis à jour en y affectant l'identifiant du destinataire du jeton. À la réception du jeton, la file transmise est fusionnée avec la file locale du récepteur. La file qui en résulte est réordonnée en fonction des priorités. L'algorithme possède en plus, des mécanismes temporels et de datation des requêtes pour différencier deux requêtes de même priorité et ainsi privilégier la plus ancienne.

Pour limiter l'inversion de priorité, Mueller a étendu son algorithme dans [Mue99] afin de l'améliorer avec les protocoles PCP (Priority Ceiling Protocol) et PIP (Priority Inheritance Protocol [SRL90]) utilisés dans les systèmes temps-réel.

L'implémentation de cet algorithme est cependant relativement complexe. De plus, en cas de système chargé l'arbre dynamique tend à devenir une chaîne puisque la racine n'est pas le dernier demandeur mais le détenteur du jeton. L'algorithme présente alors une complexité en messages en $\mathcal{O}(\frac{N}{2})$.

3.4.2 Algorithme de Chang

Chang [Cha94] reprend l'algorithme de Raymond [Ray89b] en se basant sur une topologie d'arbre statique. Sur le même principe que l'algorithme de Raymond :

- chaque site possède une file locale triée par priorité et par ordre FIFO en cas de priorité égale ;
- un site retransmet un message de requête que si celle-ci a une priorité supérieure à la priorité maximale de la file locale. Il est en effet inutile de transmettre un message de requête de plus faible priorité sachant qu'il existe une requête de priorité supérieure qui sera satisfaite avant.

Chang applique un mécanisme de priorités dynamiques aux requêtes. Ce mécanisme est appelé *aging strategy* :

- un processus incrémente de un chaque priorité des requêtes dans sa file locale avant d'envoyer le jeton ;
- à la réception d'une requête de priorité p, le processus met à la priorité p toute requête dans la file locale dont la priorité est inférieure à p ;
- à la réception du jeton qui inclut le nombre total d'exécutions de la section critique, le processus incrémente la priorité de toutes ses anciennes requêtes du nombre de sections critiques qui ont été exécutées depuis son dernier passage.

Un tel mécanisme réduit l'écart en terme de temps de réponse moyen entre les priorités (contrairement à l'algorithme de Kanrar-Chaki décrit en section 3.4.3) mais induit un plus grand nombre d'inversions.

Cet algorithme possède une légère optimisation dans l'envoi des messages. Dans l'algorithme de Raymond, rappelons que lorsque le jeton est envoyé par un processus qui possède une file locale non-vide, ce processus enverra aussi un message de requête pour éviter la famine de ses requêtes présentes dans sa file locale. L'optimisation de Chang consiste à inclure cette requête directement dans le jeton.

L'algorithme de Chang hérite des propriétés de l'algorithme de Raymond. Sa complexité moyenne en messages reste toujours logarithmique par rapport à N.

3.4.3 Algorithme de Kanrar-Chaki

L'algorithme de Kanrar-Chaki [KC10] introduit également un mécanisme de priorité dynamique en se basant sur une topologie statique d'arbre comme l'algorithme de Raymond. Le mécanisme de propagation des requêtes et la politique de stockage dans les files locales sont similaires à l'algorithme de Chang : les files locales sont triées en fonction des priorités (FIFO en cas de priorités égales) et un site retransmet un message de requête que si celle-ci a une priorité supérieure à la priorité maximale de la file locale. Cependant le mécanisme de priorité dynamique de l'algorithme de Kanrar-Chaki diffère de celui de Chang. En effet, la priorité d'une requête de la file locale est incrémentée d'un niveau à la réception de chaque message requête ayant une priorité strictement supérieure. Contrairement à Chang, la réception de requête est le seul moment où une priorité peut augmenter. De plus l'algorithme de Kanrar-Chaki n'utilise pas l'optimisation de Chang qui est d'inclure une requête dans le jeton.

Nous donnons un exemple d'exécution de cet algorithme dans la figure 3.1 afin de faciliter sa compréhension car notre nouvel algorithme à priorités présenté dans le chapitre 4 est inspiré de celui de Kanrar-Chaki.

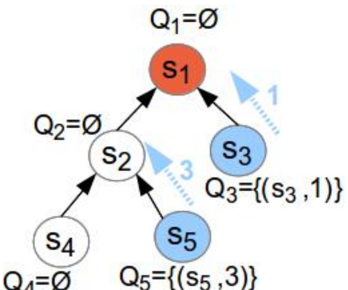
3.5 Conclusion

Nous avons vu dans ce chapitre, les deux familles d'algorithmes d'exclusion mutuelle à priorité. La famille à priorité statique respecte scrupuleusement l'ordre des priorités mais ne respecte pas la propriété de vivacité tandis que la famille à priorité dynamique permet de respecter la vivacité mais génère des inversions. Globalement les algorithmes que nous avons présenté en section 3.4 se basent sur la circulation d'un jeton dans une topologie logique. Ils ont le même mécanisme de propagation des requêtes :

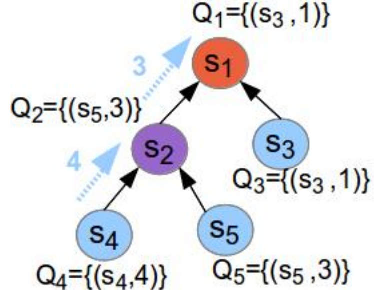
- la racine de l'arbre est le détenteur du jeton,
- chaque processus possède une file locale triée en fonction des priorités et en cas de priorité égale c'est la requête la plus ancienne qui sera favorisée,
- une requête est retransmise au site père que si elle est de priorité supérieure à la priorité maximale de la file.

Les algorithmes de Chang et Kanrar-Chaki héritent des avantages de la topologie statique de l'algorithme de Raymond, en particulier la simplicité de mise en œuvre et la complexité en messages logarithmique par rapport à N. En revanche, un site situé au cœur de l'arbre utilisant peu la section critique sera sollicité tout de même à transmettre le jeton.

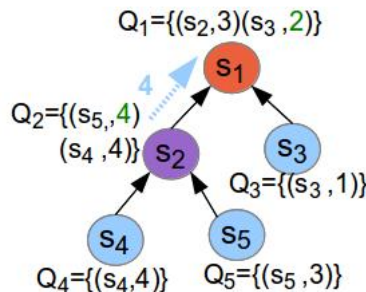
L'algorithme de Mueller a l'avantage que les sites peu demandeurs de la section critique resteront peu sollicités par la transmission de messages : les messages de jeton sont envoyés directement au prochain site qui exécutera la section critique. Cependant, le fait que le site racine soit possesseur du jeton dégrade la complexité en message car les liens logiques tendront à former à chaque transfert du jeton une chaîne et non un arbre.



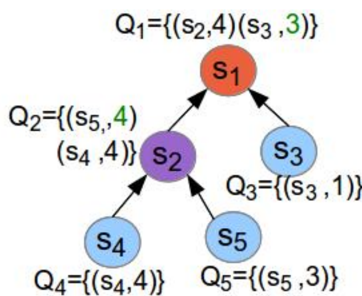
État initial : s_1 possède le jeton et est en section critique. s_3 et s_5 demandent la section critique avec les priorités respectives 1 et 3.



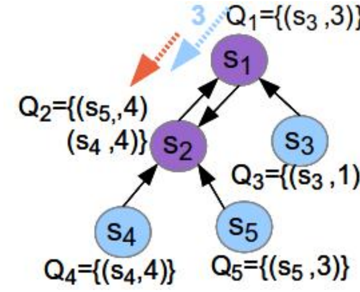
Étape 1 : s_1 et s_2 reçoivent les requêtes de s_3 et s_5 : ils les ajoutent dans leur file locale. s_2 devient *requesting* pour la requête de s_5 et envoie une requête à s_1. De plus s_4 entre en état *requesting* et envoie une requête à s_2 avec la priorité 4.



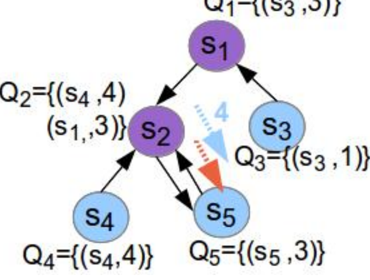
Étape 2 : s_2 reçoit la requête de s_4, incrémente la priorité locale de s_5, insère la requête et transmet une requête de priorité 4 à s_1 car sa priorité maximale a changé.
s_1 reçoit la requête de s_2, incrémente localement la priorité de s_3 et insère la requête.



Étape 3 : s_1 reçoit la requête de s_2, il incrémente la priorité de s_3 et met à jour la priorité de s_2. Notons qu'à ce stade la priorité de s_3 est égale à 3 pour s_1 alors qu'elle reste toujours à 1 pour s_3.



Étape 4 : s_1 sort de la section critique, dépile le premier élément de sa file locale et envoie le jeton à s_3. Comme Q_1 n'est pas vide, s_1 envoie également, un message de requête pour la requête de s_3.



Étape 5 : s_2 reçoit le jeton et le retransmet à s_5 en lui envoyant aussi une requête de priorité 4 pour la requête de s_4. s_2 ajoute la requête de s_1 de priorité 3 à la réception mais ne retransmet rien car sa priorité maximale est de 4.

état *requesting* état *requesting* par délégation état *tranquil* porteur du jeton

message de requête message de jeton lien structurel *father*

FIGURE 3.1 – Exemple d'exécution de l'algorithme de Kanrar-Chaki

Deuxième partie

Exclusion mutuelle à priorité et exclusion mutuelle à contraintes temporelles

Chapitre 4

Temps d'attente et inversions de priorité dans l'exclusion mutuelle

Sommaire

4.1 Introduction **34**
4.2 Définition des inversions **34**
4.3 Réduction des inversions **36**
4.3.1 Corps de l'algorithme 36
4.3.2 Amélioration des communications 38
4.3.3 Retard d'incrémentation de priorité 39
4.3.4 Prise en compte de la topologie 39
4.4 Évaluation du mécanisme de réduction d'inversions **40**
4.4.1 Protocole d'évaluation 40
4.4.2 Résultats en charge constante 41
4.4.3 Résultats en charge dynamique 49
4.4.4 Étude en charge constante avec priorité constante 51
4.4.5 Synthèse de l'évaluation 55
4.5 Réduction du temps d'attente **55**
4.5.1 Un équilibre sur deux objectifs contradictoires 56
4.5.2 Principes de l'algorithme *Awareness* 56
4.5.3 Description de l'algorithme *Awareness* 57
4.6 Évaluation des performances de l'algorithme *Awareness* . **61**
4.6.1 Protocole d'évaluation 62
4.6.2 Résultats pour une fonction de palier donnée 62
4.6.3 Impact de la fonction de palier sur les inversions et le temps de réponse 65
4.7 Conclusion **66**

4.1 Introduction

Nous avons vu dans le chapitre 3 que dans l'exclusion mutuelle distribuée à priorité, les algorithmes pouvaient être classés en deux catégories : les algorithmes à priorités statiques qui respectent scrupuleusement l'ordre des priorités mais violent la propriété de vivacité et les algorithmes à priorités dynamiques qui permettent d'assurer la vivacité mais induisent des inversions de priorités. Nous proposons alors un nouvel algorithme qui :

1. **respecte la propriété de vivacité** : ceci ne peut se faire que par un algorithme à priorité dynamique. Par conséquent notre solution finale appartiendra à cette famille et se basera sur l'algorithme de Kanrar-Chaki (voir section 3.4.3) qui repose sur une topologie en arbre statique.
2. **limite le nombre d'inversions** : les inversions sont indispensables pour assurer la vivacité car sans inversion, il y a un risque de famine. Notre objectif est donc de les limiter en introduisant un mécanisme de ralentissement des incrémentations de priorités.
3. **garde une bonne complexité en messages** : Le mécanisme de ralentissement des incrémentations pour réduire les inversions induit un surcoût en messages. Nous introduisons donc un mécanisme qui prend en compte la localité des requêtes pour limiter ce surcoût.
4. **garde un temps d'attente raisonnable pour les requêtes à faibles priorités**. La limitation en inversions de priorités peut induire dans certaines topologies des temps d'attente très importants pour les requêtes à faible priorité. Nous avons donc étendu notre mécanisme de réduction d'inversion pour donner lieu à un nouvel algorithme appelé "Awareness" s'appuyant sur une vision globale des requêtes. Cet algorithme permet d'être indépendant de la topologie sous-jacente et ainsi d'éviter les cas pathologiques du premier algorithme.

Les trois premiers objectifs sont associés aux publications [CCgrid12] et [Compas13]. La contribution du quatrième objectif a été publiée dans [ICPP13]. Ce chapitre est composé de la section 4.2 qui définit formellement la notion d'inversion de priorité. La contribution est ensuite décrite en deux parties : la première partie décrit et évalue nos mécanismes permettant de satisfaire les trois premiers objectifs dans une topologie aléatoire (sections 4.3 et 4.4) et la deuxième partie décrit et évalue l'algorithme "Awareness" (sections 4.5 et 4.6).

4.2 Définition des inversions

Intuitivement, une inversion de priorité se produit lorsque une requête est satisfaite avant une autre requête pendante de priorité supérieure ou bien plus simplement lorsque l'ordre des priorités a été violé. Lorsqu'une inversion se produit on peut distinguer deux types de requêtes :

- Une **requête favorisée** : la requête qui est satisfaite avant la requête de priorité plus importante.

- Une **requête pénalisée** : la requête du processus qui attend le jeton pendant que le processus de requête de priorité inférieure entre en section critique.

La figure 4.1 montre cinq requêtes avec leur priorité originale respective où les lignes horizontales représentent le temps d'attente de chaque requête. On rappelle qu'une requête req_i de priorité p_i est plus prioritaire qu'une requête req_j de priorité p_j si $p_i > p_j$. On observe que la requête req_1 est donc une requête favorisée puisqu'elle est satisfaite pendant que la requête req_2 attend le jeton. La requête req_2 est alors une requête pénalisée. Nous notons qu'une requête peut être à la fois favorisée et pénalisée (par exemple les requêtes req_2 et req_3).

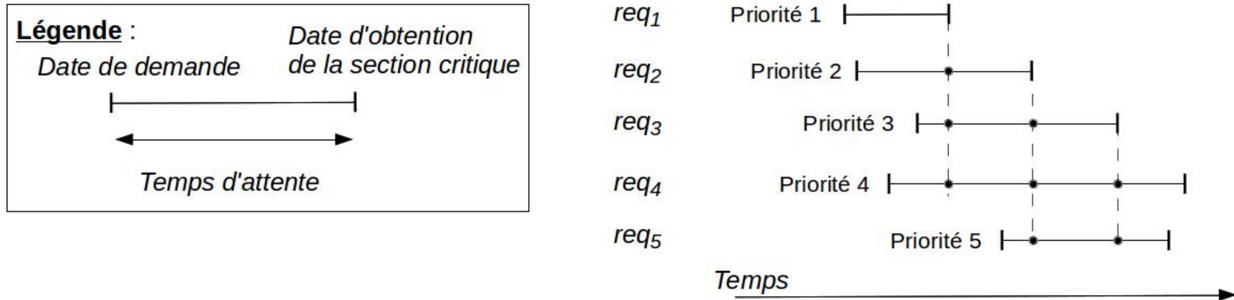


FIGURE 4.1 – Exemple de classe d'inversions de priorité

Soit le triplet $(p, t_r, t_a) \in \mathcal{P} \times T \times T$ qui représente une requête où p est la priorité, t_r est la date où elle a été émise, et t_a la date où le jeton a été acquis. Chaque triplet est unique car la sûreté assure qu'il n'est pas possible que la date d'acquisition du jeton soit la même pour deux requêtes différentes.

Les métriques relatives à l'inversion de priorité peuvent être exprimées par :

- Le **nombre de requêtes favorisées** :

$$\#\{(p, t_r, t_a) \in \mathcal{P} \times T \times T \mid \exists (p', t'_r, t'_a) \in \mathcal{P} \times T \times T, p < p' \wedge t_a \in \,]t'_r, t'_a[\, \}$$

- Le **nombre de requêtes pénalisées** :

$$\#\{(p, t_r, t_a) \in \mathcal{P} \times T \times T \mid \exists (p', t'_r, t'_a) \in \mathcal{P} \times T \times T, p' < p \wedge t'_a \in \,]t_r, t_a[\, \}$$

- Le **nombre total d'inversions de priorité** :

$$\#\{((p, t_r, t_a), (p', t'_r, t'_a)) \in (\mathcal{P} \times T \times T)^2 \mid p' < p \wedge t'_a \in \,]t_r, t_a[\, \}$$

Si l'on considère de nouveau la figure 4.1, on dénombre ainsi :

- 3 requêtes favorisées (lignes en pointillés) ;
- 4 requêtes pénalisées (lignes horizontales ayant au moins un point) ;
- 8 inversions (nombre total de points).

Contrairement à ce que l'on pourrait penser intuitivement, le nombre d'inversions n'est pas la somme du nombre de requêtes pénalisées et de requêtes favorisées comme nous pouvons le remarquer dans l'exemple.

4.3 Réduction des inversions

Notre solution pour améliorer les communications et réduire les inversions se base sur l'algorithme de Kanrar-Chaki (voir section 3.4.3). Nous avons choisi cet algorithme car il a un faible nombre de messages par section critique ($\mathcal{O}(Log(N))$). De plus, son mécanisme d'incrémentation de priorité assure l'absence de famine.

Dans un premier temps, nous appliquons l'optimisation du trafic de messages proposée dans l'algorithme de Chang [Cha94] à savoir l'ajout d'une requête dans le jeton lorsqu'il reste des requêtes pendantes. Nous ajoutons ensuite deux mécanismes :

- le mécanisme "retard" qui utilise des paliers pour ralentir l'incrémentation des priorités ;
- le mécanisme "distance" limite le nombre de transmission du jeton en considérant le nombre de nœuds intermédiaires entre l'actuel possesseur du jeton et les nœuds demandeurs.

L'optimisation du trafic de messages et ces deux mécanismes seront décrits après une description du corps de l'algorithme permettant d'améliorer les communications et de réduire les inversions.

4.3.1 Corps de l'algorithme

Pour chaque site s_i, l'algorithme définit les variables locales suivantes (ligne 1 à 5) :

- $state$: indique l'état du processus
- $father$: l'identifiant du site voisin sur le chemin menant au processus qui détient le jeton (processus racine).
- Q : la file locale de requêtes pendantes reçues par le site.

Chaque élément de la file est un tuple $(s, p, l, d) \in \Pi \times (\mathcal{P} \cup \{p_{max} + 1\}) \times \mathbb{N} \times \mathbb{N}$ où

- s est l'identifiant du voisin qui a transmis la requête
- p est la priorité courante de la requête dans la file locale. Il est possible que cette priorité locale soit supérieure à p_{max}. Ce point est détaillé dans la section 4.3.4
- l est le niveau de retard courant qui correspond au nombre courant de requêtes concurrentes de priorité supérieure qui ont déjà été prises en compte pour augmenter la priorité p à la priorité $p + 1$.
- d est la distance en nombre de nœuds intermédiaires dans l'arbre séparant l'initiateur de la requête et le nœud courant.

Chaque file locale est triée par ordre décroissant de priorité p, puis par ordre croissant de distance d en cas de priorité égale, par ordre de niveau de retard l décroissant en cas de distance égale puis enfin par ordre FIFO en cas de niveaux de retard égaux.

Nous définissons les fonctions suivantes qui manipulent la file locale Q :

- **add**(s, p, l, d) : ajoute une requête (s, p, l, d) en fonction de la politique d'ordre définie ci-dessus.
- **dequeue**(Q) : considère que Q n'est pas vide, retourne le premier élément en l'effaçant de la file Q.
- **head**(Q) : retourne le premier élément de Q mais ne l'efface pas. Si Q est vide l'élément retourné est (nil, nil, nil, nil).
- **reorder**(Q) : permet de réordonner la file en fonction de sa politique de tri.

```
1  Local variables :
2  begin
3     father : site ∈ Π or nil;
4     Q : queue of tuple
      (s, p, l, d) ∈ Π × (𝒫 ∪ {p_max + 1}) × ℕ × ℕ
      ;
5     state ∈ {tranquil, requesting, inCS};
6  end

7  Initialization
8  begin
9     Q ← ∅;
10    state ← tranquil;
11    father ← according to the initial topology;
12 end

13 Receive_Request(p_j ∈ 𝒫, d_j ∈ ℕ) from s_j
   begin
14    if father = nil and state = tranquil then
15       Send Token(∅,∅) to s_j ;
16       father ← s_j ;
17    else if s_j ≠ father then
18       (s_old, p_old, l_old, d_old) ← head(Q);
19       foreach (s, p, l, d) ∈ Q do
20          (s_head, p_head, l_head, d_head) ←
            head(Q);
21          if s = s_j then
22             if p_j ≥ p then
23                p ← p_j ;
24                d ← d_j ;
25                l ← 0 ;
26          else if p_j > p or (p_j = p and
            p = p_head) then
27             l ← l + 1;
28             if l = ℱ(p + 1) then
29                p ← p + 1;
30                l ← 0;
31       if ∄(s, p, l, d) ∈ Q, s = s_j then
32          add (s_j, p_j, 0, d_j) in Q;
33       reorder (Q) ;
34       if father ≠ nil then
35          if (s_old, p_old, l_old, d_old) ≠
            head(Q) then
36             Send Request(p_j, d_j + 1) to
               father;
37 end

38 Request_CS(p ∈ 𝒫)
39 begin
40    state ← requesting;
41    if father ≠ nil then
42       add (self, p, 0, 0) in Q ;
43       if (self, p, 0, 0) = head(Q) then
44          Send Request(p,1) to father;
45       wait(father = nil);
46    state ← inCS;
47    /* CRITICAL SECTION                                  */
48 end

49 Release_CS()
50 begin
51    state ← tranquil;
52    if Q ≠ ∅ then
53       (s_next, p_next, l_next, d_next) ← dequeue(Q);
54       (s_head, p_head, l_head, d_head) ← head(Q);
55       Send Token(min(p_head, p_max),d_head + 1) to
         s_next;
56       father ← s_next;
57 end

58 Receive_Token(p_j ∈ 𝒫, d_j ∈ ℕ) from s_j
59 begin
60    father ← nil ;
61    (s_next, p_next, l_next, d_next) ← dequeue(Q);
62    if p_j ≠ nil then
63       foreach (s, p, l, d) ∈ Q do
64          (s_head, p_head, l_head, d_head) ← head(Q);
65          if p_j > p or (p_j = p and p = p_head)
            then
66             l ← l + 1;
67             if l = ℱ(p + 1) then
68                p ← p + 1;
69                l ← 0;
70       add (s_j, p_j, 0, d_j) in Q;
71    reorder (Q) ;
72    if s_next = self then
73       notify(father = nil);
74    else
75       (s_head, p_head, l_head, d_head) ← head(Q);
76       Send Token(min(p_head, p_max),d_head + 1) to
         s_next;
77       father ← s_next;
78 end
```

FIGURE 4.2 – Algorithme retard-distance

Lorsqu'un site désire accéder à la ressource avec une priorité p, il exécute la fonction *Request_CS* (ligne 38). Celle-ci inclut sa requête dans sa file locale (ligne 42). Si cette requête est la première de la file (c'est à dire la plus prioritaire), le processus envoie un message de requête à son père (ligne 44).

Chaque requête reçue contient la priorité de la requête p et sa distance d en nombre de sauts depuis l'initiateur. Lorsqu'un processus s_i reçoit une requête (ligne 13) d'un voisin s_j, si s_i est la racine mais n'utilise pas le jeton (état *tranquil*) alors le jeton est envoyé en direction du demandeur (ligne 15) via s_j. Si s_i n'est pas la racine, la requête est ajoutée dans Q. Si il existe déjà une requête dans Q provenant de s_j, la priorité et la distance sont mises à jour et l'indice de retard réinitialisé à zéro (lignes 22 à 25). Les indices de retard et éventuellement les priorités des autres requêtes pendantes dans Q sont incrémentés (lignes 26 à 30). Ensuite si le processus a ajouté la requête en tête de file alors le message de requête est retransmis au père (ligne 36). Le test de la ligne 17 est utile quand le jeton est en transit entre le site courant et s_j. En effet, lorsque le jeton est en transit, il existe un cycle sur les liens $father$ entre l'envoyeur du message jeton et son destinataire. Il peut donc arriver qu'un message de requête provenant de s_j croise le message de jeton que le processus courant vient d'envoyer. Le test de la ligne 17 permet ainsi de ne jamais enregistrer une requête venant du site père et évite ainsi des cycles dans la transmission de messages.

Lorsqu'un processus reçoit le jeton, si le jeton contient une requête (voir section 4.3.2), il met à jour les priorités des requêtes dans la file locale et y ajoute ensuite cette requête (lignes 62 à 70). Si sa propre requête est la tête de file (sa requête est la plus prioritaire), le processus peut entrer en section critique et passer à l'état *inCS* (ligne 73). Sinon, le jeton est transmis au nœud de la requête de la tête de file (ligne 76)

Enfin, lorsqu'un site sort de section critique en appelant la procédure *Release_CS* (ligne 50), si sa file locale n'est pas vide, le jeton est envoyé au processus de la requête de la tête de file. Si après la suppression de cette requête, la file demeure non vide, le processus inclut dans le jeton la première requête de la file. Le jeton est ensuite envoyé et le lien $father$ est mis à jour (lignes 55 et 56).

4.3.2 Amélioration des communications

Dans l'algorithme de Kanrar-Chaki, lorsqu'un site ayant plusieurs requêtes en attente envoie le jeton pour satisfaire la première requête de la file, il envoie également un message de requête pour récupérer le jeton. Ceci permet de satisfaire les autres requêtes en attente dans la file locale. Ces envois successifs de deux messages causalement liés posent un problème d'efficacité. Une solution proposée dans l'algorithme de Chang est d'adjoindre la requête lors de l'envoi du jeton (lignes 55 et 76). On économise ainsi un message à chaque envoi du jeton tout en maintenant la cohérence du protocole.

Un autre défaut de l'algorithme de Kanrar-Chaki, réside dans l'envoi systématique d'une requête lorsqu'un site désire entrer en section critique. Nous proposons alors de limiter cet envoi au seul cas où aucune requête pendante de priorité supérieure n'a été émise (ligne 43). On évite ainsi de nombreux envois inutiles en cas de forte charge.

4.3.3 Retard d'incrémentation de priorité

Outre l'envoi de messages inutiles, l'algorithme de Kanrar-Chaki présente un grand nombre d'inversions. En effet, le mécanisme d'incrémentation conduit rapidement à ce que des requêtes initialement de faible priorité obtiennent la priorité maximale dans une file locale. Nous proposons donc un nouveau mécanisme permettant de retarder l'incrémentation de priorité. La valeur d'une priorité d'une requête pendante n'est pas incrémentée à chaque insertion d'une nouvelle requête de plus haute priorité mais seulement après un certain nombre d'insertions. Nous définissons alors une fonction de palier notée $\mathcal{F} : \mathcal{P} \rightarrow \mathbb{N}$. $\mathcal{F}(\mathrm{p})$ qui permet de définir la politique d'incrémentation c'est à dire le nombre d'insertions de requêtes de priorité supérieure à $p-1$ pour qu'une requête de priorité $p-1$ passe localement à la priorité p (lignes 27 à 30 et lignes 66 à 69). $\mathcal{F}$ est une fonction monotone croissante et positive et peut être assimilée à un paramètre de l'algorithme. Puisque notre objectif premier est de réduire le nombre d'inversions au maximum, nous considérerons dans les expérimentions une fonction de palier exponentiellement croissante ($\mathcal{F}(p) = 2^{p+c}$, où la constante c permet d'éviter que les petites priorités n'augmentent trop vite). Ainsi plus la priorité suivante à atteindre est grande plus il sera long de l'obtenir.

4.3.4 Prise en compte de la topologie

Dans l'algorithme original de Kanrar-Chaki, les requêtes de même priorité ne sont pas ordonnées. Cependant la topologie induit des coûts de transmission différents suivant la localisation des sites demandeurs. En ajoutant de l'information à une requête, il devient envisageable d'optimiser le transfert du jeton et donc le nombre de messages ainsi que le taux d'utilisation de la section critique. On ajoute à chaque requête un compteur incrémenté à chaque retransmission. On utilise alors ce compteur pour prendre en compte la localité des requêtes. Cette localité est quantifiée par le nombre de liens intermédiaires entre s_i et s_j que le jeton doit traverser. Ainsi, pour deux requêtes pendantes de même priorité, le jeton sera envoyé à la plus proche. Cependant un tel mécanisme peut introduire des famines puisqu'il peut arriver que le jeton se transmette indéfiniment dans une portion de l'arbre où il existe des processus qui demandent en permanence la section critique avec la même priorité. Ceci peut générer une famine pour d'éventuels processus de même priorité en attente de la section critique se situant loin de cette portion d'arbre : les processus proches dans l'arbre passeront en permanence devant les requêtes éloignées. Pour palier à ce problème, lorsqu'un nœud reçoit une requête de priorité p, il incrémente l'indice de retard de toutes les requêtes de priorité p' si $p > p'$ ou bien si p est la plus haute priorité locale et $p = p'$ (lignes 26 et 65). Ainsi il est possible que localement une requête soit de priorité $p_{max} + 1$, ce qui permet d'assurer que dans un temps fini toute requête sera la première d'une file locale.

4.4 Évaluation du mécanisme de réduction d'inversions

4.4.1 Protocole d'évaluation

Les expériences ont été réalisées sur une grappe de 32 machines avec un processus par machine. Ceci permet d'éviter les effets de bord de contention au niveau des cartes réseau puisqu'il y a un seul processus par carte. Chaque nœud a deux processeurs Xeon 2.5GHz, 16 Go de mémoire RAM, s'exécute sur un noyau Linux 2.6 (cluster Grid5000 Nancy). Les nœuds sont reliés par un switch Ethernet de 20 Gbit/s. Les algorithmes ont été implémentés en C++ en utilisant l'intergiciel OpenMPI.

Une expérience est caractérisée par :

- N : le nombre de processus (32 dans notre cas).
- α : le temps d'une section critique (5 millisecondes dans notre cas).
- β : intervalle de temps d'attente entre la sortie d'une section critique par un processus et l'émission de la prochaine requête par ce même processus, autrement dit le temps où un site reste à l'état *tranquil* (paramètre calculé grâce à une valeur de charge ρ décrite ci-après)
- γ : la latence réseau,i.e., le temps d'acheminement d'un message entre deux processus voisins (0,15 millisecondes dans notre cas)
- ρ : la charge du système exprimée par le rapport $\beta/(\alpha+\gamma)$. La charge détermine la fréquence à laquelle la section critique est demandée, autrement dit le débit de requêtes en entrée du système. Ce paramètre est inversement proportionnel à la charge du système : plus la valeur de ce paramètre est faible, plus la charge est haute et vice versa. Ce rapport s'exprime proportionnellement à N. Le paramètre γ doit être pris en compte dans le calcul de la charge ρ si sa valeur n'est pas négligeable par rapport à α. En effet le temps de transfert du jeton peut être vu comme une extension du temps de section critique α.
- θ : le temps de l'expérience (120 secondes dans notre cas).
- $\mathcal{F}(p)$: fonction de palier (égale à 2^{p+c}, où $c = 6$)

Les métriques considérées dans cette évaluation de performances sont celles décrites en section 2.3.4, à savoir :

- **le nombre de messages moyen par requête** : le ratio entre le nombre de total de messages de ce type émis par le protocole nombre total de requêtes.
- le **temps de réponse moyen** : le temps moyen entre l'émission d'une requête et l'obtention de la section critique. Cela correspond au temps moyen en état *requesting*. Dans cette étude, cette métrique est donnée par niveau de priorité.
- le **taux d'utilisation** : le pourcentage de temps passé en section critique durant l'expérience.

En plus de ces métriques propres à l'exclusion mutuelle classique, nous ajoutons une métrique supplémentaire propre à l'exclusion mutuelle à priorité qui est **le nombre total d'inversions de priorités** décrit en section 4.2. Nous normalisons ici ce nombre total d'inversions par le nombre total de requêtes émises car les différents algorithmes considérés n'ont pas le même débit d'exécution de section critique.

Dans chaque expérience nous avons considéré une topologie logique en arbre binaire et 8 priorités différentes. Nous avons comparé les algorithmes suivants :

- L'algorithme de Kanrar-Chaki [KC10] (voir section 3.4.3).
- L'algorithme de Chang [Cha94] (voir section 3.4.2).
- *CommOpti* correspond à une version modifiée de l'algorithme de Kanrar-Chaki avec l'optimisation de communication décrit en section 4.3.2.
- *CommOpti_retard* correspond à l'algorithme *CommOpti* auquel on a ajouté le mécanisme de retard d'incrémentation de priorité (voir section 4.3.3).
- *CommOpti_retard_distance* correspond à l'algorithme *CommOpti_retard* auquel on a ajouté le mécanisme de prise en compte de topologie : le mécanisme "distance" (voir section 4.3.4)

4.4.2 Résultats en charge constante

Dans cette partie, nous présentons des résultats d'évaluation de performances lorsque la charge de requête reste constante dans une expérience, i.e., la fréquence de demandes de section critique (paramètre ρ) ne varie pas dans une expérience. L'intervalle entre deux demandes (paramètre β) est calculé de manière aléatoire suivant une loi de Poisson où la moyenne est calculée grâce au paramètre de charge ρ.

Dans chaque expérience, les processus demandent la section critique périodiquement et choisissent de manière uniformément aléatoire une priorité à chaque nouvelle requête. Pour considérer un régime stationnaire durant les mesures, nous avons introduit un temps de préchauffage au moment du démarrage du système. Par conséquent la charge de requête pendante est constante durant les mesures de l'expérience. Dans cette partie, notre étude comporte deux étapes : nous considérons dans un premier temps une charge fixe donnée et dans un second temps nous étudions l'impact de différentes charges (chacune toujours constante dans une expérience) sur le comportement de l'algorithme.

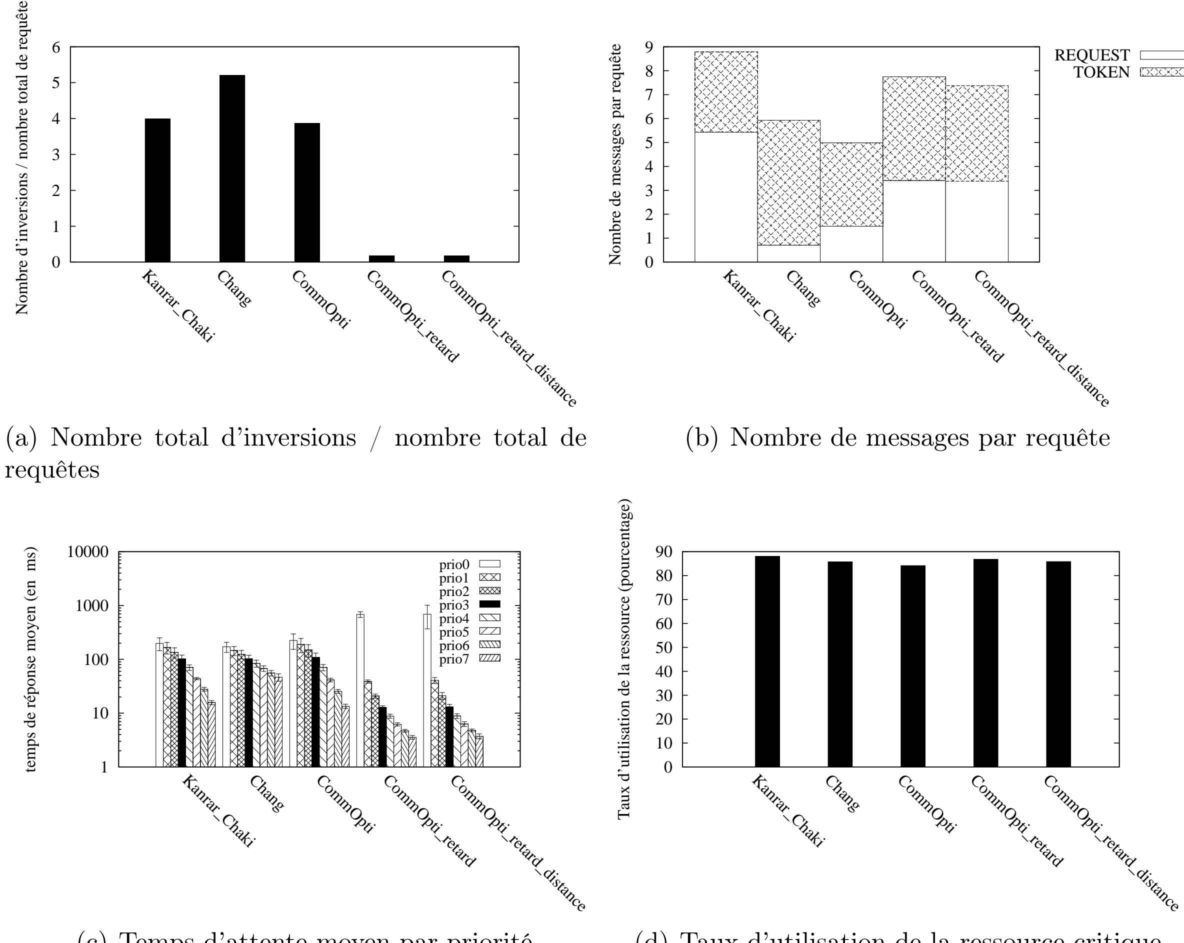
Évaluation de performance pour une charge donnée

Étude globale

La charge est fixée à $\rho = 0,5N$ qui correspond à une charge intermédiaire (environ 50 % des processus attendent la section critique). La figure 4.3 montre le comportement des algorithmes sur les différentes métriques considérées.

Nous pouvons observer dans la figure 4.3(a) que l'algorithme de Kanrar-Chaki présente environ 25 % d'inversions en moins que l'algorithme de Chang tout en ayant 1 taux d'inversion très élevé avec en moyenne 4 inversions par requête. Cependant ceci se fait au détriment de la complexité en nombre de messages (figure 4.3(b)) : 40 % de messages supplémentaires. Les performances de *CommOpti* montrent que le simple ajout des mécanismes d'optimisation des messages à l'algorithme de Kanrar-Chaki suffisent à obtenir une complexité comparable à celle de Chang tout en conservant le même taux d'inversion.

Toutefois, le nombre d'inversions de priorités obtenues par *CommOpti* reste encore très important (4 inversions par requête). L'ajout du mécanisme de retard d'incrémentation permet de réduire considérablement ce nombre d'un facteur 25. Ce bon résultat sur la métrique la plus importante pour un algorithme à priorité se fait au détriment du nombre de messages. En effet, avec le mécanisme de retard, les sites atteignent la priorité maximale



(a) Nombre total d'inversions / nombre total de requêtes

(b) Nombre de messages par requête

(c) Temps d'attente moyen par priorité

(d) Taux d'utilisation de la ressource critique

Figure 4.3 – Performances du mécanisme de retard avec une charge intermédiaire

plus lentement et sont donc susceptibles de retransmettre d'avantage de requêtes. Pour contrebalancer le surcoût en messages généré par le mécanisme de retard, l'ajout du mécanisme *"distance"* qui prend en compte la topologie des requêtes permet d'économiser 5 % des messages tout en conservant un taux d'inversions de priorités très faible. Nous pourrons constater plus tard que ce gain est plus important lorsque la charge augmente.

En ce qui concerne le temps de réponse moyen, nous observons dans la figure 4.3(c) que l'algorithme original de Kanrar-Chaki a un comportement régulier (en forme d'escalier),i.e., quand la priorité augmente, le temps de réponse moyen diminue. À l'inverse, avec le mécanisme de retard, les écarts de latence entre les différentes priorités sont moins réguliers : la priorité 0 voit sa latence largement augmentée (traitement "best-effort"), tandis que les plus hautes priorités bénéficient d'une forte amélioration des temps d'accès. Enfin, lorsque l'on compare les latences de *CommOpti_retard* avec *CommOpti_retard_distance*, on remarque qu'il n'y a pas de différence sur les latences moyennes des différentes priorités. Le gain en nombre de messages ne se fait qu'au prix d'une augmentation de l'écart type pour les faibles priorités.

Pour terminer, la figure 4.3(d) nous permet de vérifier que les performances de nos différents mécanismes en termes de respect des priorités ne se font pas au dépend des performances globales : le taux d'utilisation du jeton reste inchangé autour de 80%.

En conclusion cette étude confirme que le retard de l'incrémentation est essentiel pour le respect de l'ordre des priorités tandis que la localité des requêtes est utile pour la réduction du nombre de messages générés par l'algorithme.

Approfondissement de l'étude sur les inversions

La figure 4.4 présente une série de résultats visant à affiner l'étude de performance sur les inversions. Ainsi, les sous-figures montrent pour chaque algorithme :

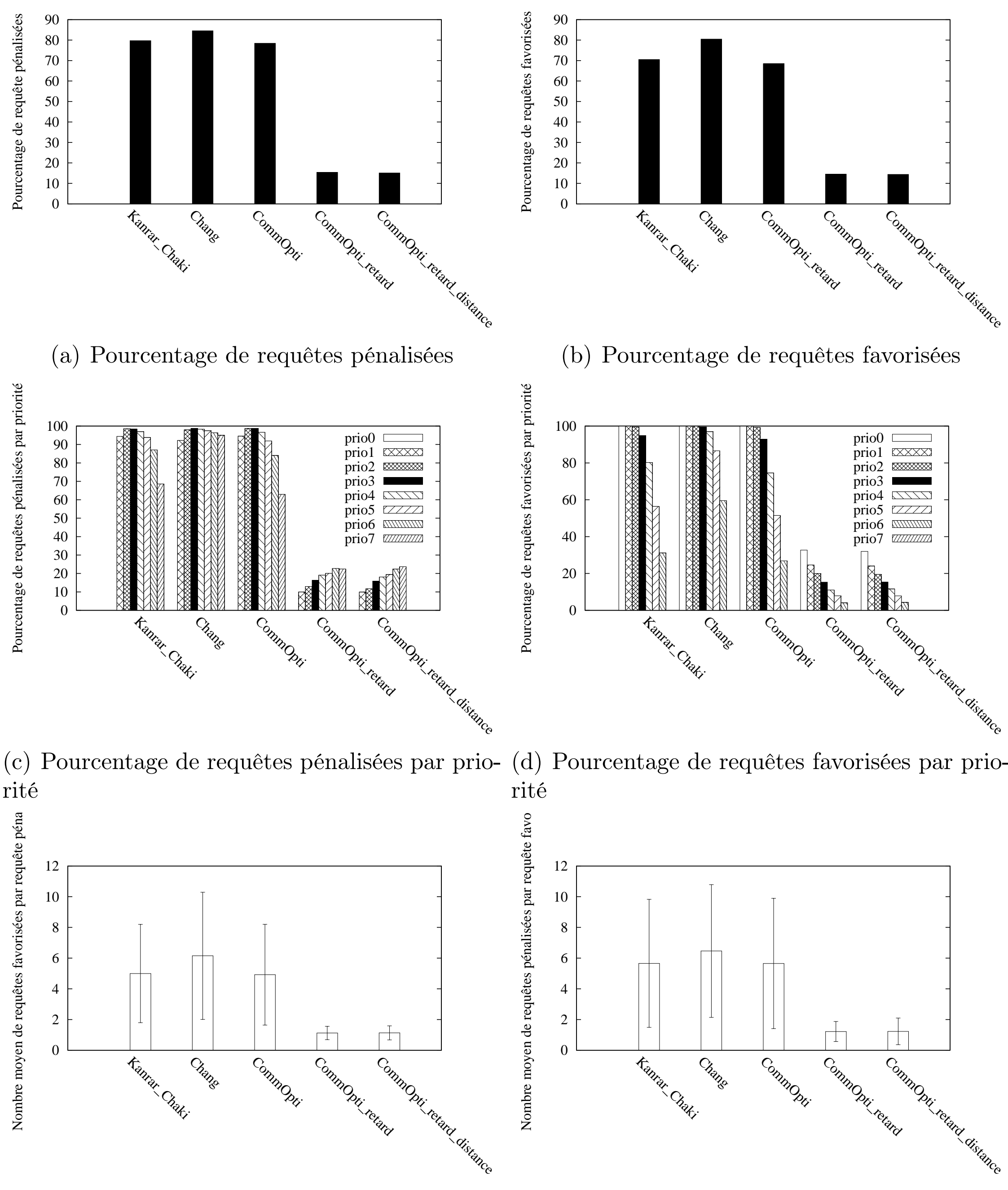
- Le pourcentage de requêtes pénalisées et le nombre de requêtes favorisées (Figures 4.4(a) et 4.4(b)).
- Pour chaque niveau de priorité, le pourcentage de requêtes pénalisées et favorisées respectivement (Figures 4.4(c) et 4.4(d)).
- Le nombre moyen de requêtes favorisées par requête pénalisée et le nombre moyen de requêtes pénalisées par requête favorisée (Figures 4.4(e) et 4.4(f)).

Comme nous pouvons l'observer dans les figures 4.4(a) et 4.4(b), les algorithmes sans mécanisme de retard induisent plus de requêtes pénalisées que de favorisées mais pour les algorithmes avec mécanisme de retard, ces deux métriques ont la même valeur.

Dans les figures 4.4(c) et 4.4(d), on retrouve cette différence pour chacune des priorités. Mais cette figure nous montre aussi des différences sur la distribution des requêtes pénalisées et favorisées au sein d'un même algorithme : dans les algorithmes avec mécanisme de retard, le nombre de requêtes pénalisées augmente linéairement avec la priorité, tandis que dans les autres algorithmes, les requêtes les plus pénalisées sont les requêtes de priorités intermédiaires.

Pour comprendre ces différences, il faut savoir que le risque d'être pénalisé est un compromis entre deux phénomènes :

- le nombre de requêtes susceptibles de dépasser une requête plus prioritaire dépend uniquement de la priorité initiale, et non de l'algorithme. En effet, plus la priorité



(a) Pourcentage de requêtes pénalisées

(b) Pourcentage de requêtes favorisées

(c) Pourcentage de requêtes pénalisées par priorité

(d) Pourcentage de requêtes favorisées par priorité

(e) Nombre moyen de requêtes favorisées par requête pénalisée

(f) Nombre moyen de requêtes pénalisées par requête favorisée

FIGURE 4.4 – Analyse approfondie des inversions pour une charge intermédiaire

d'une requête est haute, plus le nombre de requêtes de priorité inférieure augmente. Ainsi, la probabilité de se faire dépasser par une requête de priorité inférieure augmente.
- La probabilité de dépasser qui dépend du mécanisme d'incrémentation et de la priorité initiale. Plus la priorité initiale d'une requête est basse et plus le mécanisme d'incrémentation est rapide, plus elle risque de pénaliser des requêtes de priorité supérieure.

Dans les algorithmes bénéficiant du retard de l'incrémentation, le deuxième phénomène devient négligeable. Le premier phénomène explique à lui seul la linéarité entre le nombre d'inversions et la priorité. À l'inverse, dans le cas des autres algorithmes, l'augmentation est rapide et donc non négligeable, on remarque ainsi que :
- La vitesse d'augmentation est très rapide pour les faibles priorités (priorités 0 et 1) puisqu'elles ont de grande chance de se faire dépasser par des requêtes de priorité supérieure. Cette augmentation rapide a pour effet de fortement pénaliser les priorités intermédiaires de valeur 3.
- Les requêtes de plus hautes priorités (priorités 6 et 7) peuvent être doublées par plus de requêtes mais ces dernières n'augmentent que plus lentement leur priorité (cas pour les priorités 4 et 5) ou sont très éloignées (cas pour les priorités 0 et 1).

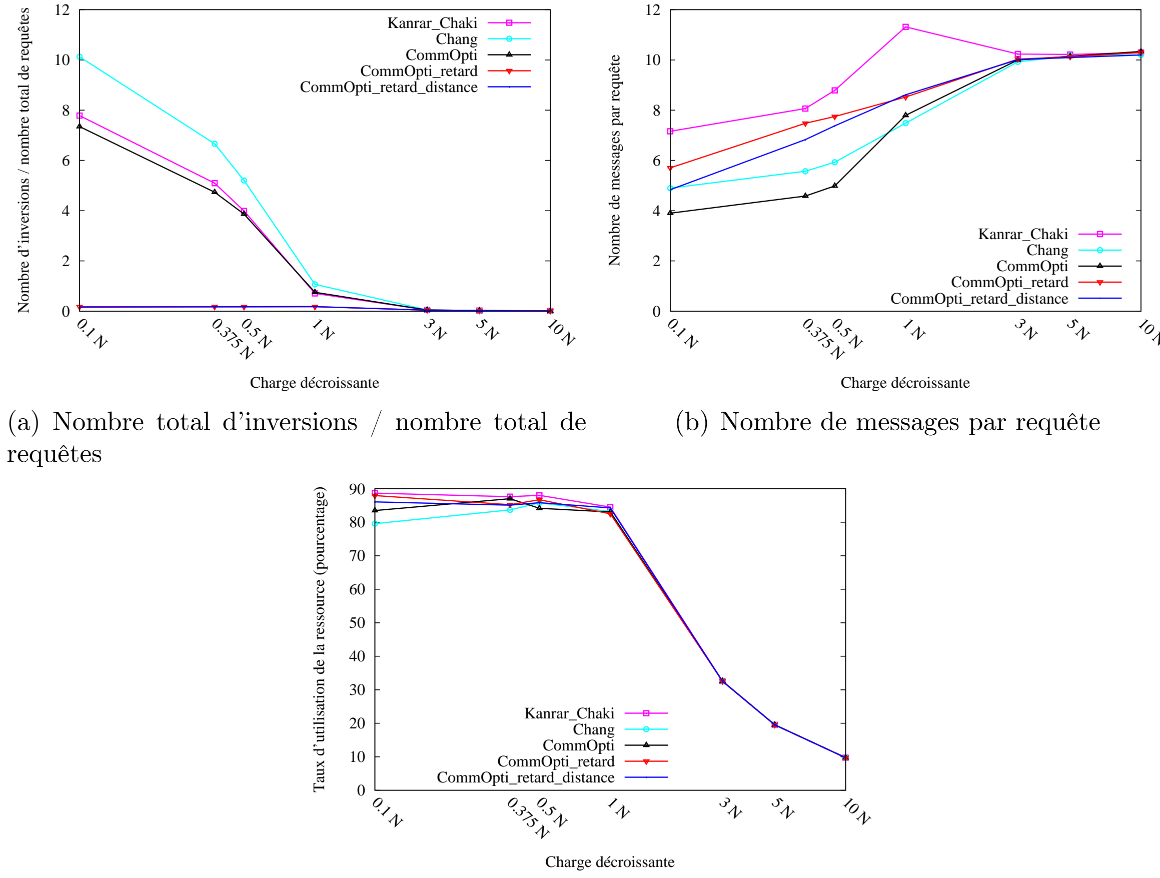
Dans ces algorithmes on observe donc un compromis entre les deux phénomènes conduisant à pénaliser d'avantage les priorités intermédiaires.

La figure 4.4(e) montre que pour les algorithmes avec retard d'incrémentation, les requêtes pénalisées ne sont doublées en moyenne qu'une seule fois (écart-type faible), tandis qu'elles le sont 5 à 6 fois dans les autres approches. En considérant les figures 4.4(a) et 4.4(b), on peut conclure que les requêtes pénalisées sont moins nombreuses et qu'elles sont doublées par moins de requêtes. Ceci explique les très bons résultats en termes d'inversions globales observées dans la figure 4.3(a).

Évaluation de l'impact de la charge

Nous présentons maintenant les résultats d'une étude de l'impact de la charge (paramètre ρ) sur les performances des différents algorithmes. Nous avons ainsi renouvelé les expériences avec différentes valeurs de ρ proportionnellement au nombre de processus N : $0.1N$, $0.375N$, $0.5N$, $1N$, $3N$, $5N$, et $10N$. Ces valeurs correspondent respectivement environ à 85%, 65%, 55%, 15%, 1%, 0.5% et 0.2% de processus en moyenne en attente de section critique à un instant donné. Pour structurer notre étude nous distinguons trois ensemble de charge :
- **Charge haute** $(0,1N \leq \rho < 0,375N)$: une majorité de processus demande la section critique de manière concurrente
- **Charge intermédiaire** $(0,375N \leq \rho < 3N)$: en moyenne la moitié des processus demande la section critique
- **Charge basse** $(3N \leq \rho \leq 10N)$: les requêtes concurrentes sont rares

Dans la figure 4.5, nous étudions le nombre de messages par requête, le taux d'utilisation de la section critique et le nombre de violations tandis que dans la figure 4.6, nous approfondissons de nouveau l'analyse des inversions.



(a) Nombre total d'inversions / nombre total de requêtes

(b) Nombre de messages par requête

(c) Taux d'utilisation de la ressource critique

FIGURE 4.5 – Étude de l'impact de la charge sur le nombre de messages, le taux d'utilisation et le nombre d'inversions

Étude globale

Nous remarquons sur la figure 4.5(a) que les algorithmes avec mécanisme de retard sont insensibles aux fortes charges : le nombre d'inversions reste très faible quelle que soit la valeur de ρ. En revanche, le nombre d'inversions augmente considérablement pour les autres algorithmes ne bénéficiant pas de ce mécanisme lorsque la charge augmente. En effet, la forte charge accroît le nombre de requêtes concurrentes, ce qui les conduit rapidement à atteindre la priorité maximale. Ces algorithmes ne pouvant plus distinguer les priorités, génèrent donc un grand nombre d'inversions.

D'autre part, nous observons sur la figure 4.5(b) que le nombre de messages diminue lorsque la charge augmente quel que soit l'algorithme. Ce phénomène est une conséquence directe de l'algorithme de Raymond à la base des cinq algorithmes : un nœud ne retransmet pas de requête si il est déjà demandeur de la section critique. En cas de forte charge, ce cas est très courant, ce qui réduit naturellement le nombre de retransmissions. Outre ce phénomène global, on peut observer également dans cette figure, le surcoût en messages engendré par le mécanisme de retard, ainsi que le gain apporté par le mécanisme de prise en compte de la topologie (mécanisme *"distance"*). Il est cependant intéressant de remarquer que l'efficacité du mécanisme *"distance"* est d'autant plus importante que la charge augmente, l'algorithme devenant plus économe pour une charge de $0,1N$ ($\simeq 85$ %). Ceci est particulièrement intéressant pour des applications présentant des pics de charges.

Concernant le taux d'utilisation de la section critique, nous observons dans la figure 4.5(c) que tous les algorithmes ont le même comportement, i.e., pour une valeur de ρ donnée, ils satisfont tous le même nombre de requêtes.

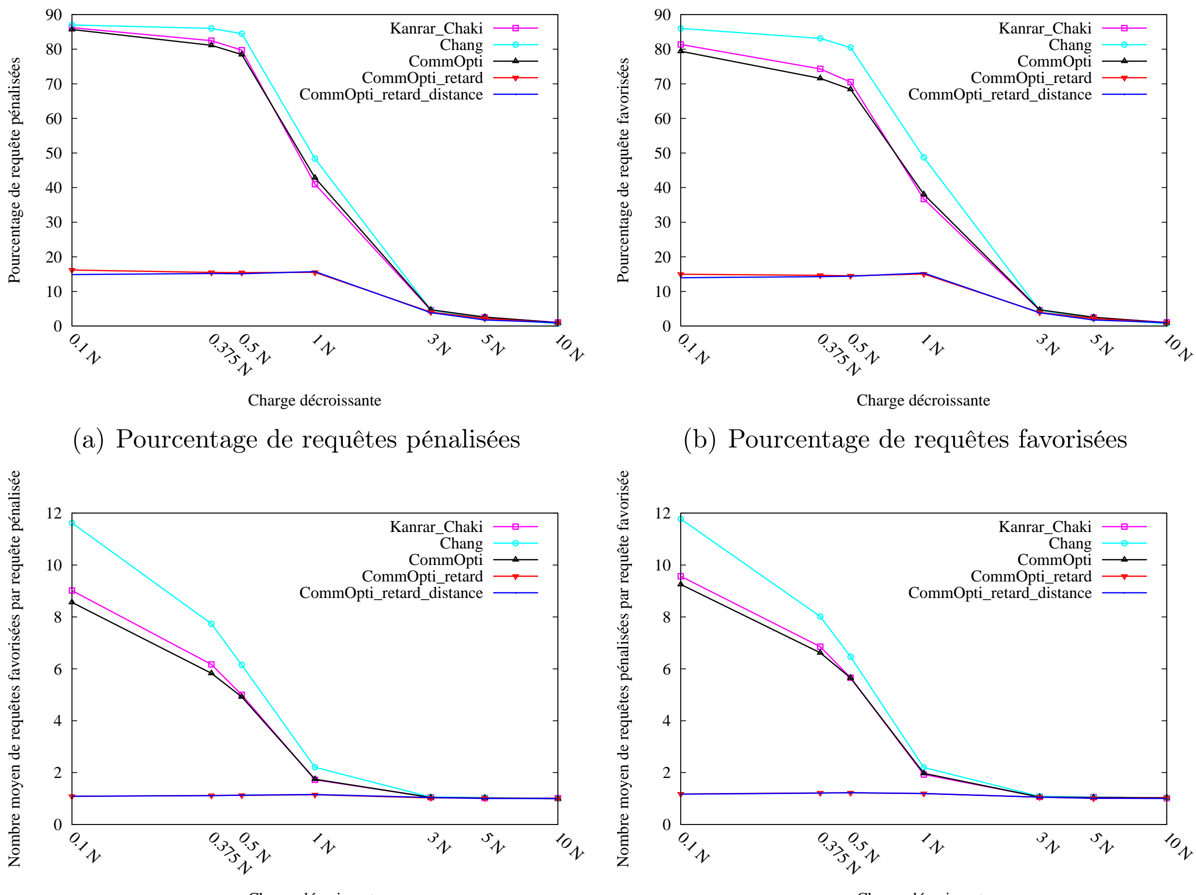
Il est important de souligner, que les trois graphiques de la figure 4.5 confirment qu'en cas de faible charge ($\rho > 3N$), tous les algorithmes se comportent de la même manière car en absence de concurrence la priorité est ignorée.

Approfondissement de l'étude sur les inversions (figure 4.6)

Nous représentons respectivement dans les figures 4.6(a), 4.6(b), 4.6(c) et 4.6(d), le nombre de requêtes pénalisées, le nombre de requêtes favorisées, le nombre moyen de fois où une requête est pénalisée et le nombre moyen de fois où une requête est favorisée.

Dans la figure 4.6(a), nous remarquons, qu'en cas de faible charge ($10N$), il n'y a aucune requête pénalisée quel que soit l'algorithme. Lorsque la charge commence à augmenter (de $10N$ à $3N$), quelques requêtes pénalisées commencent à apparaître. Avec une charge moyenne (de $3N$ à $0.5N$), nous remarquons que seuls les algorithmes sans mécanisme de retard, augmentent de manière significative le nombre de requêtes pénalisées (jusqu'à 80%) tandis que les algorithmes avec mécanisme de retard continuent d'augmenter très légèrement.

Finalement lorsque la charge est forte ($\rho < 0,5N$), le pourcentage de requêtes pénalisées augmente légèrement pour atteindre 85% pour les algorithmes sans mécanisme de retard. Ceci représente la proportion des requêtes ayant une priorité initiale strictement supérieure à zéro. En effet, les requêtes avec une priorité initiale de 0 ne peuvent pas être pénalisées. Autrement dit 100% des requêtes pénalisables sont pénalisées. On retrouve le même type de comportement pour les requêtes favorisées (Figure 4.6(b)).



(a) Pourcentage de requêtes pénalisées

(b) Pourcentage de requêtes favorisées

(c) Nombre moyen de requêtes favorisées par requête pénalisée

(d) Nombre moyen de requêtes pénalisées par requête favorisée

FIGURE 4.6 – Impact de la charge sur les requêtes pénalisées et favorisées

Pour terminer, nous nous intéressons au nombre moyen de requêtes doublant une requête pénalisée (Figure 4.6(c)). En comparant ces résultats avec la figure 4.6(a), nous remarquons que si le nombre de requêtes pénalisées devient important (40%) pour une charge intermédiaire ($1N$), elles ne sont doublées en moyenne que deux fois. Une fois passé le seuil de $0,5N$, presque toutes les requêtes pénalisables sont pénalisées, mais le nombre de requêtes les dépassant continue d'augmenter fortement. Ceci explique pourquoi la quantité d'inversions continue d'augmenter en forte charge (entre $0,5N$ et $0,1N$) dans la figure 4.5(a). On comprend alors les mauvaises performances en forte charge : toutes les requêtes pénalisables sont doublées environ 10 fois.

Les résultats présentés dans cette section ont montré l'impact de la charge sur les performances des algorithmes. On peut donc se demander comment ils s'adapteraient à des changements de charge durant leur exécution.

4.4.3 Résultats en charge dynamique

Le but de l'évaluation décrite dans cette section est d'évaluer le comportement des algorithmes lorsque la charge varie dans une même expérience pendant son exécution. Nous voulons ainsi montrer la capacité des algorithmes à s'adapter à la variation de charge. Ainsi nous allons injecter des pics de charge (forte demande) à intervalles réguliers au cours d'une expérience. Nous modélisons ici la charge par le pourcentage de processus qui sont en attente du jeton. Comme dans la section précédente, les processus choisissent de manière uniformément aléatoire une priorité à chaque nouvelle requête.

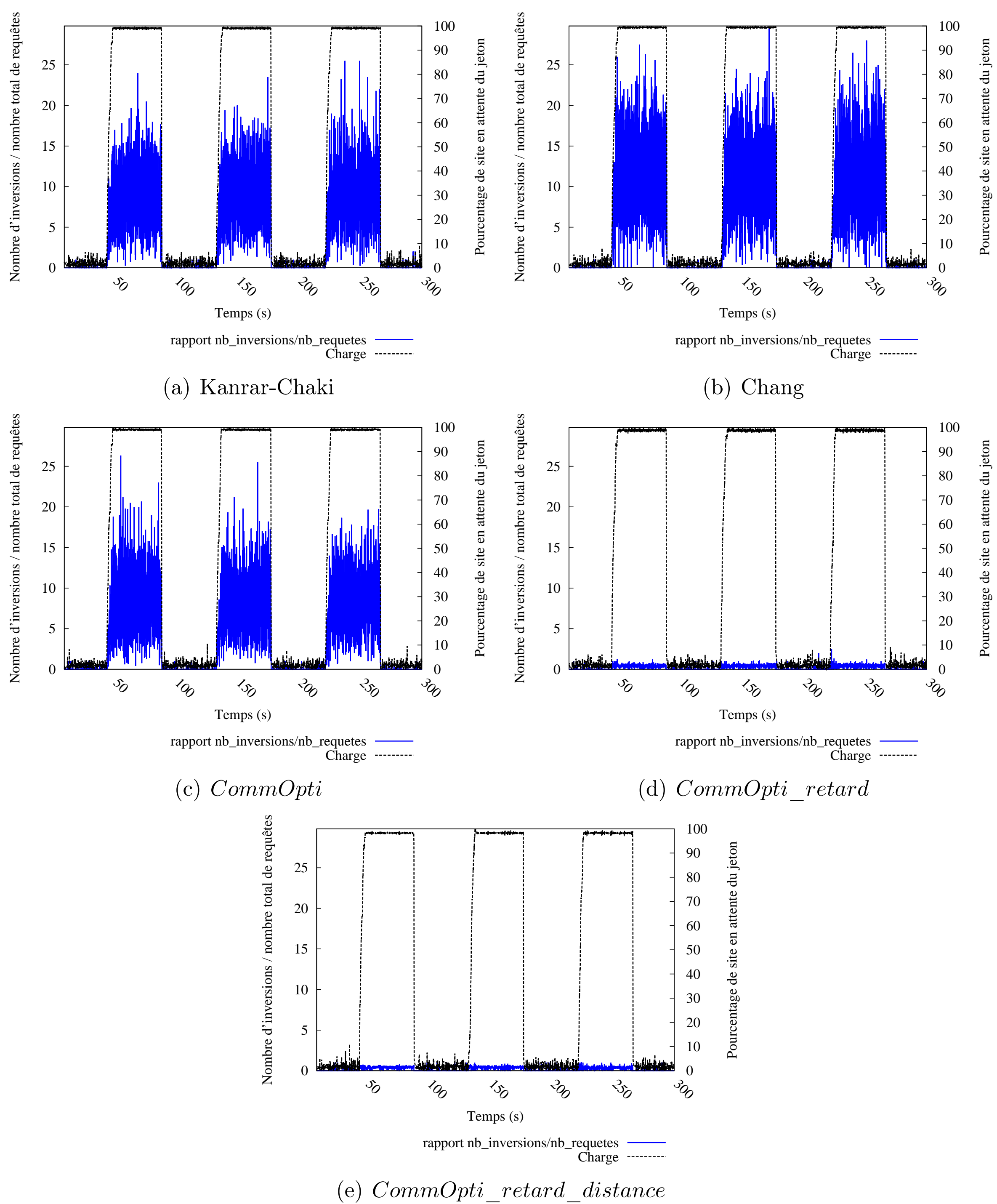
Les figures 4.7(a), 4.7(b),4.7(c), 4.7(d), et 4.7(e) montrent respectivement le nombre d'inversions pour Kanrar-Chaki, Chang, *CommOpti*, *CommOpti_retard* et *CommOpti_retard_distance*.

Les figures 4.7 montrent pour tous les algorithmes, le nombre d'inversions de priorités à un intervalle de temps donné de chaque expérience. Un point d'abscisse est un échantillon égal à un intervalle de 50 ms. Dans cette analyse, le temps d'expérience θ est égal à 300 secondes.

Les figures 4.7(a), 4.7(b), et 4.7(c) confirment que la quantité d'inversions augmente de manière significative pendant la durée du pic de charge pour les algorithmes sans mécanisme de retard : pendant le pic de charge, le nombre total d'inversions par requête varie entre une valeur minimum proche de 1 et une valeur maximale au alentour de 25 . Ce résultat est cohérent avec la figure 4.5(a) où nous pouvions observer une absence d'inversion en cas de faible charge puisque le nombre de requêtes augmente plus rapidement que le nombre d'inversions.

En revanche, les algorithmes à mécanisme de retard sont insensibles à la variation de charge. Pendant le pic de charge, la quantité d'inversions est bornée par une valeur maximale qui est plus petite que la valeur minimale des autres algorithmes.

En conclusion, l'étude en charge dynamique confirme que le mécanisme de retard d'incrémentation s'adapte à la charge en ce qui concerne la quantité d'inversions générées.



(a) Kanrar-Chaki

(b) Chang

(c) *CommOpti*

(d) *CommOpti_retard*

(e) *CommOpti_retard_distance*

FIGURE 4.7 – Rapport (nombre total d'inversions / nombre total de requêtes) avec une charge dynamique

4.4.4 Étude en charge constante avec priorité constante

Dans les sections 4.4.2 et 4.4.3, nous considérions que les priorités étaient choisies de manière aléatoire à chaque nouvelle requête. Cependant, dans beaucoup d'applications, les priorités sont assignées statiquement au processus. Ainsi, contrairement aux précédentes expériences, les processus demanderont désormais la section critique avec la même priorité pendant toute la durée de l'exécution de l'application. D'autre part, puisque notre approche se base sur une topologie logique en arbre statique, la position des nœuds a une influence sur les performances. Dans les précédentes expériences, cette influence n'était pas perceptible car les processus changeaient de priorité à chaque nouvelle requête. Le but de cette section est d'étudier le comportement des algorithmes avec différentes distributions des priorités sur l'arbre.

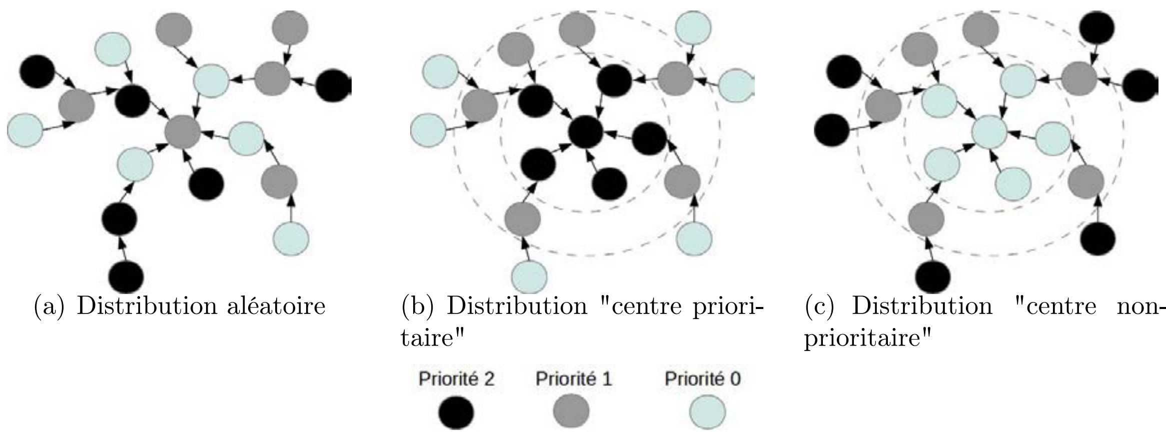
Nous notons **centre du graphe**, l'ensemble des nœuds pour qui l'excentricité est égale au rayon du graphe (excentricité minimale). Par conséquent, les distances maximales entre les nœuds centraux du graphe et les autres nœuds sont les plus petites.

Nous définissons alors trois politiques possibles de distribution de priorité :

- **Aléatoire** : les processus sont placés de manière aléatoire dans la topologie (Figure 4.8(a)).
- **Centre prioritaire** : les processus à plus haute priorité sont placés dans le centre du graphe. Ainsi, plus un site est éloigné du centre, plus sa priorité diminue (Figure 4.8(b)). Autrement dit, si nous considérons que la racine initiale de l'arbre est le nœud central alors un nœud a une priorité strictement plus faible que son père initial (sauf pour les nœuds au niveau 1) mais ont une priorité strictement plus forte que leurs fils initiaux (sauf pour le nœud racine initial). Cette distribution possède deux avantages :
  - ∗ le coût en messages des requêtes est moins important. En effet les trajets du jeton sont réduits car il restera la plupart du temps dans la zone centrale, "attiré" par les processus à forte priorité
  - ∗ le nombre d'inversions de priorité est intrinsèquement réduit : lorsque le jeton s'éloigne de la zone centrale du graphe pour satisfaire des requêtes à faible priorité, il est probable que pendant son transfert, il satisfasse les requêtes à priorités intermédiaires dans un ordre décroissant. Ceci induit donc un meilleur respect des priorités.
- **Centre non-prioritaire** : À l'inverse de la distribution "Centre prioritaire", les processus à plus basse priorité sont placés dans le centre du graphe. Ainsi, plus un site s'éloigne du centre, plus sa priorité augmente (Figure 4.8(c)).

Les expériences de cette section ont été menées avec une charge constante intermédiaire ($\rho = 0.5N$). Le nombre de processus N considéré est toujours égal à 32 et le nombre de priorités P toujours égal à 8 : à chaque niveau de priorité il y a 4 processus. Nous nous intéresserons uniquement au temps de réponse. Les résultats sont donnés pour chaque topologie sous forme d'histogrammes (Figure 4.9) et sous forme de tableaux de valeurs (Figure 4.10).

En premier lieu, quelle que soit la distribution, on observe globalement le même comportement des algorithmes par rapport aux précédentes expériences de la section 4.4.2 : les algorithmes *CommOpti_retard* et *CommOpti_retard_distance* ont de meilleures



(a) Distribution aléatoire (b) Distribution "centre prioritaire" (c) Distribution "centre non-prioritaire"

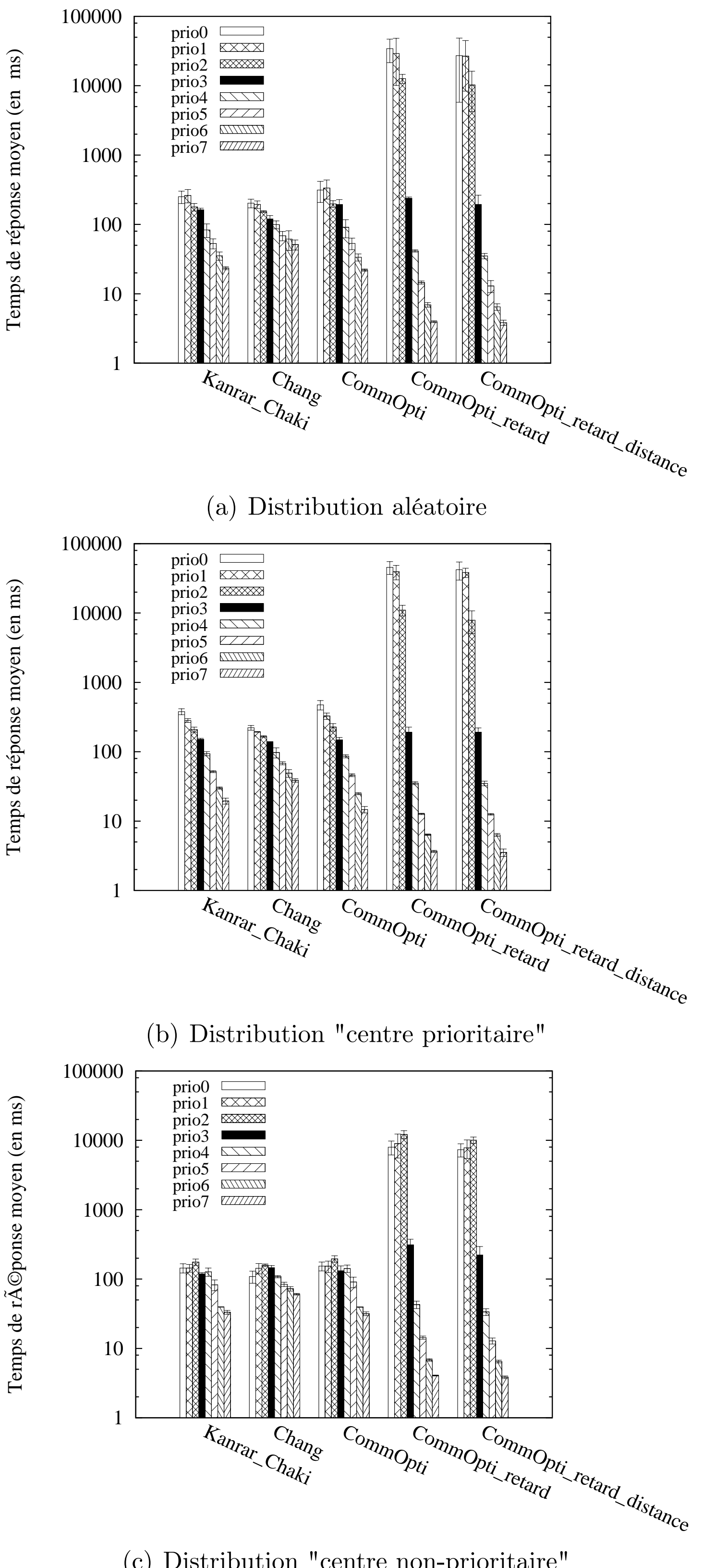
FIGURE 4.8 – Schémas des distributions de priorités considérées dans l'arbre

performances car les grandes priorités ont un temps de réponse plus faible au détriment des petites priorités.

En second lieu cette expérience montre que la topologie a un réel impact sur les performances des algorithmes. En disposant les processus plus prioritaires au centre du graphe (Distribution "centre prioritaire", Figure 4.9(b) et Tableau 4.10(b)), le temps d'attente des moyennes et grandes priorités est réduit pour tous les algorithmes lorsque l'on fait une comparaison avec la distribution aléatoire (Figure 4.9(a) et Tableau 4.10(a)). Ce résultat est tout à fait cohérent puisque dans la disposition "centre prioritaire", les sites centraux, plus prioritaires ont plus de chance d'intercepter le jeton pour entrer en section critique.

Les expériences montrent aussi trois autres résultats intéressants :

- globalement le gain en temps de réponse des hautes priorités est réduit dans la distribution "centre prioritaire" quelque soit l'algorithme
- Le temps de réponse des hautes priorités des algorithmes de Chang et Kanrar-Chaki en distribution "centre prioritaire" (donc favorable aux plus hautes priorités) sont encore très supérieurs à ceux des algorithmes *CommOpti_retard* et *CommOpti_retard_distance* en distribution aléatoire. Les requêtes à haute priorité (respectivement à faible priorité) des algorithmes à mécanisme de retard présentent un plus faible temps de réponse (respectivement plus fort) en distribution aléatoire que les autres algorithmes en distribution "centre prioritaire" (distribution qui est normalement favorable aux plus hautes priorités). Cependant, les algorithmes sans mécanisme de retard, ont des meilleures performances dans une distribution "centre prioritaire" que dans une distribution aléatoire. Ceci confirme que le placement de priorité dans la topologie a un impact sur le temps de réponse (voir tableaux 4.10(a) et 4.10(b)).
- Quel que soit l'algorithme et quelle que soit la distribution, les écart-types des temps de réponse des requêtes à hautes priorités sont faibles. Cependant la distribution aléatoire augmente les écart-types des petites priorités pour tout les algorithmes. Cette différence peut s'expliquer par le fait que la probabilité qu'une priorité d'une requête augmente est fortement lié à la position du processus dans le graphe : à



(a) Distribution aléatoire

(b) Distribution "centre prioritaire"

(c) Distribution "centre non-prioritaire"

FIGURE 4.9 – Temps de réponse moyen en fonction de la distribution des priorités dans l'arbre

Aléatoire	prio0		prio1		prio2		prio3		prio4		prio5		prio6		prio7	
_	moy	Ec-type	moy	Ec-type	moy	Ec-type	moy	Ec-type	moy	Ec-type	moy	Ec-type	moy	Ec-type	moy	Ec-type
Kanrar_Chaki	250,64	50,61	261,27	56,61	178,62	22,08	162,29	09,17	83,25	18,66	53,14	08,80	35,26	04,84	23,38	01,19
Chang	202,14	29,32	194,14	24,42	153,29	04,57	120,03	14,09	99,40	13,02	68,58	10,63	61,49	19,43	51,18	08,47
CommOpti	312,89	105,34	334,91	102,74	197,40	22,05	193,45	34,42	90,61	26,49	53,29	10,13	33,64	03,91	21,98	00,77
CommOpti_retard	34233,85	12746,94	29158,51	19071,79	12675,00	1871,17	238,27	10,42	41,50	01,56	14,53	00,73	06,93	00,52	03,95	00,12
CommOpti_retard_distance	27194,55	21432,00	26597,70	18229,90	10205,80	5973,76	194,48	69,97	35,05	03,13	12,93	02,54	06,47	00,70	03,83	00,34

(a) Distribution aléatoire

Centre prioritaire	prio0		prio1		prio2		prio3		prio4		prio5		prio6		prio7	
_	moy	Ec-type	moy	Ec-type	moy	Ec-type	moy	Ec-type	moy	Ec-type	moy	Ec-type	moy	Ec-type	moy	Ec-type
Kanrar_Chaki	379,09	37,13	281,63	17,56	208,94	17,85	149,36	06,87	93,83	06,41	51,62	01,73	29,97	01,05	19,49	01,93
Chang	221,79	17,36	193,30	02,48	166,10	04,25	138,61	01,70	97,51	16,24	68,19	03,49	49,06	06,18	38,66	02,23
CommOpti	473,59	73,12	326,99	34,75	226,17	28,11	148,14	12,63	86,01	04,44	45,89	02,19	24,89	00,83	14,69	01,58
CommOpti_retard	45528,14	9531,29	39304,08	9208,16	11007,86	1920,55	190,47	36,32	35,24	01,70	12,70	00,26	06,37	00,16	03,65	00,13
CommOpti_retard_distance	42140,91	12315,02	38438,71	5899,04	7891,51	2848,17	191,07	29,20	34,99	02,69	12,54	00,33	06,32	00,30	03,52	00,44

(b) Distribution "centre prioritaire"

Centre non prioritaire	prio0		prio1		prio2		prio3		prio4		prio5		prio6		prio7	
_	moy	Ec-type	moy	Ec-type	moy	Ec-type	moy	Ec-type	moy	Ec-type	moy	Ec-type	moy	Ec-type	moy	Ec-type
Kanrar_Chaki	144,21	21,85	143,93	18,13	176,29	18,16	118,65	05,10	126,79	17,46	82,35	14,55	39,63	00,25	33,18	02,12
Chang	108,58	21,64	143,00	24,33	158,06	07,05	145,92	10,33	108,09	03,88	84,04	06,10	72,31	05,24	60,53	01,61
CommOpti	153,41	22,41	152,98	28,77	195,29	21,10	131,80	22,92	141,68	17,07	90,65	15,76	39,42	00,18	31,66	01,99
CommOpti_retard	7963,77	1815,43	8946,36	3387,89	12166,91	1638,32	312,45	62,72	42,74	05,21	14,33	00,75	06,81	00,27	04,07	00,06
CommOpti_retard_distance	7326,17	1582,07	7820,12	2334,06	10075,06	1079,55	221,74	73,01	33,55	03,79	12,86	01,29	06,48	00,29	03,84	00,17

(c) Distribution "centre non-prioritaire"

FIGURE 4.10 – Temps de réponse moyen en fonction de la distribution des priorités dans l'arbre (tableau des valeurs)

l'inverse des sites en bordure de graphe, les sites centraux ont plus de chance de recevoir des requêtes et donc plus de chance d'augmenter en priorité.

Contrairement aux distributions aléatoires et "centre prioritaire", la distribution "centre non-prioritaire" groupant les processus à petite priorité dans le centre du graphe, dégrade les performances des algorithmes (figure 4.9(c)). Nous pouvons observer une inversion dans le temps de réponse : les priorités 1 et 2 ont un temps de réponse plus important que les priorités 0. Ce comportement illustre de nouveau l'impact du placement des priorités sur les performances de l'algorithme. Il existe un compromis entre la priorité et la place du processus dans le graphe : les petites (respectivement les hautes) priorités comme 0 et 1 (respectivement 6 et 7) sont favorisées (respectivement pénalisées) par leur position centrale (respectivement en bordure) mais sont pénalisées (respectivement favorisées) par leur valeur de priorité. D'autre part, les requêtes de priorité intermédiaire (2 et 3) ne profitent ni de la position centrale ni de leur valeur de priorité. Par conséquent, ces dernières ne sont pas du tout favorisées, ce qui explique qu'elles présentent le pire temps de réponse dans la distribution "centre non-prioritaire" .

En conclusion, lorsque les priorités sont directement associées aux processus, leur place dans l'arbre a un impact sur le temps de réponse des requêtes.

4.4.5 Synthèse de l'évaluation

Nous avons comparé notre mécanisme avec les algorithmes de Chang et de Kanrar-Chaki dans deux configurations.

Dans la première configuration où les processus pouvaient changer de priorité à chaque nouvelle requête, nous avons pu constater que le retard de l'incrémentation permettait un meilleur respect de l'ordre des priorités et que la prise en compte de la localité des requêtes était utile pour réduire le surcoût en nombre de messages générés par ce mécanisme. Ce résultat restait valable aussi bien en faible charge qu'en forte charge.

Dans la seconde configuration où les processus avaient une priorité qui leur était associée statiquement, nous avons pu constater l'impact de la topologie sur le temps de réponse. Si les processus à faible priorité se situent dans la partie centrale du graphe, alors les requêtes les plus pénalisées sont les requêtes à priorité intermédiaire. À l'inverse, si les processus à faible priorité se trouvent en bordure de graphe, nous avons constaté qu'entre deux priorités successives, le temps d'attente de la priorité supérieure était moins important. Ceci implique donc un meilleur respect de priorités.

4.5 Réduction du temps d'attente

Les évaluations de performances de la section 4.4 ont montré que le mécanisme de retard d'incrémentation de priorité permettait de réduire de manière conséquente le nombre d'inversions. Cependant elle a aussi montré que le temps d'attente augmentait significativement. Cette section a pour objectif d'introduire un mécanisme améliorant l'équité de l'algorithme tout en conservant un bon respect des priorités.

4.5.1 Un équilibre sur deux objectifs contradictoires

Vouloir minimiser le nombre d'inversions et le temps d'attente des requêtes à faible priorité sont deux objectifs contradictoires : d'une part, si il n'y a pas d'inversion de priorité alors le temps de réponse des requêtes à faible priorité peut être infini (cas des algorithmes à priorités statiques), d'autre part, une réduction du temps de réponse des requêtes à faible priorité implique un grand nombre d'inversions : les requêtes moins prioritaires doublent celles plus prioritaires (cas de l'algorithme de Chang et de Kanrar-Chaki).

Par conséquent, il faut trouver un équilibre entre ces deux objectifs contradictoires. À notre niveau il n'est pas possible de connaître le point d'équilibre car celui-ci dépend des besoins de l'application qui utilise l'algorithme de verrouillage. Il est en revanche possible de fournir des mécanismes permettant d'ajuster ce compromis. Dans notre cas, pour une charge donnée, le réglage entre le temps de réponse et le nombre d'inversions peut se faire grâce à deux paramètres fixés par l'utilisateur :

- **la fonction de palier $\mathcal{F}$** : permet de définir la vitesse d'incrémentation des priorités. Plus cette fonction est croissante plus l'incrémentation sera retardée impliquant une réduction du nombre d'inversions au détriment du temps de réponse et vice versa.
- **le placement des processus dans la topologie** : la section 4.4.4 a montré que la place des processus dans la topologie logique a un impact sur les performances : si les nœuds placés au centre du graphe sont les plus prioritaires (e.g distribution "centre prioritaire") alors le nombre d'inversions diminue naturellement mais le temps d'attente des petites priorités est plus important.

Cependant, cette dépendance à la topologie, et de facto du nombre de processus, apporte une forte dépendance entre les besoins de l'application et le système.

4.5.2 Principes de l'algorithme *Awareness*

Comme nous venons de le voir, les performances de algorithme "retard-distance" peuvent être affectées par la topologie. En effet si la distribution choisie par l'utilisateur est de type "centre prioritaire" (choix motivé par une meilleure performance en termes de complexité en messages par exemple) et si nous considérons un plus grand nombre de processus, il est possible lorsque la charge devient suffisamment haute que les temps d'attente des requêtes à faible priorité deviennent énormes et inacceptables pour une application. En cas de forte charge, le jeton restera la plupart du temps dans la région centrale du graphe. En effet dans l'algorithme "retard-distance", un processus n'incrémente les priorités de sa file locale qu'à à la réception d'une requête de plus forte priorité provenant de son sous-arbre. Or, dans la topologie "centre prioritaire", le sous arbre d'un processus s_i est la plupart du temps composé de processus de priorités inférieures. Les priorités des requêtes dans la file locale de s_i ne seront donc incrémentées qu'aux très rares réceptions du jeton contenant probablement une requête de plus forte priorité. Par conséquent, la distribution des priorités peut amener l'algorithme à pénaliser énormément les faibles priorités. Pour mieux comprendre le problème il faut raisonner en termes de connaissance. Dans l'algorithme "retard-distance", un processus ne connaît que les requêtes émises dans son sous-arbre courant et ignore toutes les requêtes émises dans le reste du système. Un

tel manque de connaissance limite l'efficacité de prise de décision.

Notre allons donc modifier notre algorithme en lui ajoutant un mécanisme assurant que chaque processus connaisse à terme le nombre total de requêtes émises pour chaque niveau de priorité. Il sera alors possible d'incrémenter les priorités en prenant en compte celles qui proviennent du sous-arbre mais aussi celles qui proviennent du reste du système. Les requêtes émises dans la zone centrale du graphe pourront être comptabilisées par les processus en bordure de graphe (de faible priorité). Ces derniers ne seront donc plus pénalisés par leur position dans le graphe.

Ce mécanisme a fait l'objet de la contribution [ICPP13] avec l'algorithme *Awareness*[1] qui étend l'algorithme de "retard-distance". La figure 4.11 illustre les différences entre les algorithmes de "Kanrar-Chaki", "retard-distance", et "Awareness", dans une topologie "centre prioritaire". Cet exemple montre le nombre de requêtes de priorité initial p nécessaires pour atteindre la configuration 2 à partir de la configuration 1. Dans la configuration 1 le jeton est utilisé en zone de priorité p (le centre du graphe) et les processus s_1 et s_2 attendent le jeton pour entrer en section critique. Dans la configuration 2, s_2 détient le jeton et la priorité de s_1 dans la file locale de s_2 a été incrémentée. Cette exemple montre qu'il y a une différence de facteur $\mathcal{F}(p)$ entre les algorithmes "retard-distance" et "Awareness". La figure 4.12 résume pour chaque algorithme un ordre de grandeur en termes de nombre de requêtes nécessaires émises de priorité p pour qu'une requête de priorité initiale $p' \leq p$ reçoive le jeton.

4.5.3 Description de l'algorithme *Awareness*

Cette section présente l'algorithme "Awareness" qui étend l'algorithme "retard-distance" (voir section 4.3). Le but de cet algorithme est de faire en sorte qu'un nœud puisse considérer l'ensemble des requêtes pendantes du système pour incrémenter la priorité des requêtes enregistrées dans la file locale. Le mécanisme de distance reste inchangé mais le mécanisme de retard de d'incrémentation présente des différences notables.

Pour propager la connaissance de l'ensemble des requêtes dans le système, nous utilisons le jeton. Ainsi, un vecteur $Vtok$ de P entrées (une entrée par priorité) est inclus dans le jeton. Chaque entrée p de ce vecteur est à terme égal au nombre total de requêtes émises de priorité p. Le pseudo code de l'algorithme "Awareness" est donné figure 4.13.

Variables locales et messages

Chaque site s_i maintient les variables suivantes :

- les variables communes à l'algorithme "retard-distance" (voir section 4.3.1) :
  - ∗ $state$: indique l'état du processus
  - ∗ $father$: l'identifiant du site voisin sur le chemin menant au processus qui détient le jeton (processus racine).
  - ∗ Q : la file locale de requêtes pendantes reçues par le site. La définition d'un élément de la file et la politique de tri restent inchangées.
- les variables supplémentaires :

1. le nom de cet algorithme a été inspiré d'un dicton d'un célèbre philosophe contemporain : "Être aware c'est être conscient de tout ce qui se trouve autour de nous, à l'intérieur et entre nous." JCVD

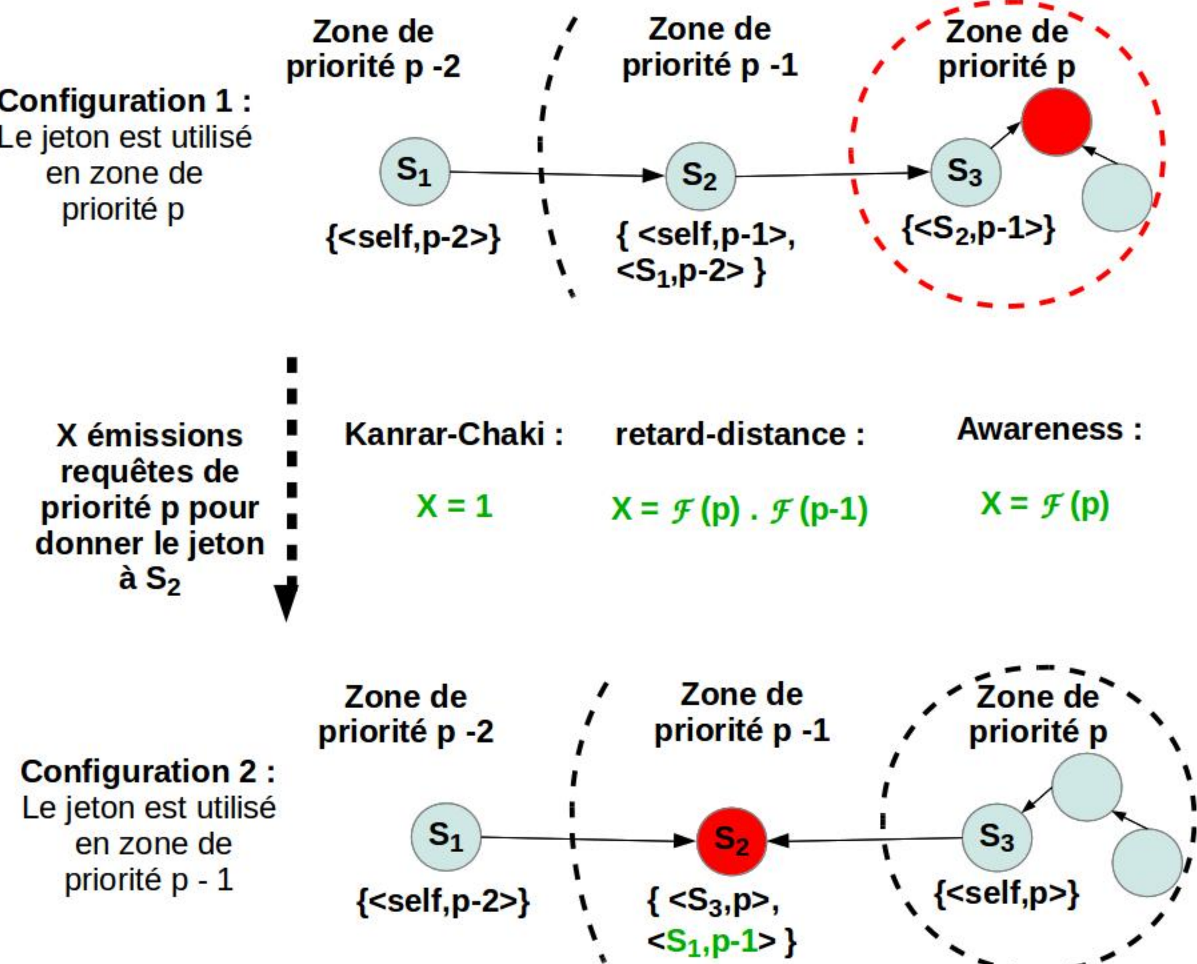


FIGURE 4.11 – Différence de comportement entre les différents algorithmes

Algorithme	ordre de
Kanrar-Chaki	$p - p'$
"Retard-distance"	$\sum_{i=p'+1}^{p} \left(\prod_{j=i}^{p} \mathcal{F}(j) \right)$
Awareness	$(p - p')\mathcal{F}(p)$

FIGURE 4.12 – Ordre de grandeur en nombre d'émissions de requêtes de priorité p, pour acheminer le jeton en zone de priorité $p' < p$

```
1  Local variables :
2  begin
3      father : site ∈ Π or nil;
4      state ∈ {tranquil, requesting, inCS};
5      Q : queue of tuple
       (s, p, l, d) ∈ Π × (𝒫 ∪ {p_max + 1}) × ℕ × ℕ ;
6      last_token : Vector of P integers ;
7      pending : Vector of P integers ;
8  end

9  UpdateLocalQueue(V : Vector of P integers)
10 begin
11     (s_head, p_head, l_head, d_head) ← head(Q);
12     for priority p_k from p_min + 1 to p_max do
13         for n from 1 to V[p_i] do
14             foreach (s, p, l, d) in Q do
15                 if p_k > p or (p_k = p and
                   p_k = p_head) then
16                     l ← l + 1;
17                     if l = ℱ(p + 1) then
18                         p ← p + 1;
19                         l ← 0;
20     reorder (Q) ;
21 end

22 Request_CS(p ∈ 𝒫)
23 begin
24     state ← requesting;
25     if father ≠ nil then
26         add (self, p, 0, 0) in Q ;
27         if (self, p, 0, 0) = head(Q) then
28             Send Request(p,1) to father;
29         else
30             pending[p] ← pending[p] + 1;
31         wait(father = nil);
32     else
33         pending[p] ← pending[p] + 1;
34     state ← inCS;
35     /* CRITICAL SECTION                              */
36 end

37 Release_CS()
38 begin
39     state ← tranquil;
40     UpdateLocalqueue(pending);
41     last_token ← last_token + pending ;
42     pending[i] ← 0 ∀i ∈ [1, P];
43     if Q ≠ ∅ then
44         (s_next, p_next, l_next, d_next) ← dequeue(Q);
45         (s_head, p_head, l_head, d_head) ← head(Q);
46         Send Token(min(p_head, p_max),d_head + 1,
           last_token) to s_next;
47         father ← s_next;
48 end

49 Initialization
50 begin
51     father ← according to the initial topology;
52     Q ← ∅;
53     state ← tranquil;
54     last_token[i] ← 0 ∀i ∈ [1, P];
55     pending[i] ← 0 ∀i ∈ [1, P];
56 end

57 Receive_Request(p_j ∈ 𝒫, d_j ∈ ℕ) from s_j
58 begin
59     if father = nil and state = tranquil then
60         last_token[p_j] ← last_token[p_j] + 1;
61         Send Token(∅,∅,last_token) to s_j ;
62         father ← s_j ;
63     else if s_j ≠ father then
64         (s_old, p_old, l_old, d_old) ← head(Q);
65         if ∃(s, p, l, d) ∈ Q, s = s_j then
66             if p ≤ p_j then
67                 p ← p_j ;
68                 d ← d_j ;
69                 l ← 0;
70                 reorder(Q) ;
71         else
72             addqueue(< s_j, p_j, 0, d_j >,Q) ;
73         if (s_old, p_old, l_old, d_old) ≠ head(Q) then
74             if father ≠ nil then
75                 Send Request(p_j, d_j + 1) to father;
76         else
77             pending[p_j] ← pending[p_j] + 1;
78     else
79         pending[p_j] ← pending[p_j] + 1;
80 end

81 Receive_Token(p_j ∈ 𝒫, d_j ∈ ℕ, Vtok) from s_j
82 begin
83     father ← nil ;
84     (s_next, p_next, l_next, d_next) ← dequeue(Q);
85     Vtok ← Vtok + pending;
86     UpdateLocalqueue(Vtok − last_token);
87     last_token ← Vtok;
88     pending[i] ← 0 ∀i ∈ [1, P];
89     if p_j ≠ nil then
90         add (s_j, p_j, 0, d_j) in Q;
91     if s_next = self then
92         notify(father = nil);
93     else
94         (s_head, p_head, l_head, d_head) ← head(Q);
95         Send Token(min(p_head, p_max),d_head + 1) to
           s_next;
96         father ← s_next;
97 end
```

FIGURE 4.13 – Algorithme Awareness

∗ *pending* (vecteur de P entiers) : $pending[p_j]$ correspond au nombre de requêtes de priorité p_j qui n'ont pas encore été prises en compte pour incrémenter les priorités. Ce vecteur est utilisé pour compter le nombre global de requêtes en attente connues par s_i depuis la dernière fois où il a, soit libéré la section critique, soit reçu le jeton. Parmi l'ensemble des vecteurs *pending* du système, une requête est comptabilisée exactement une seule fois.

∗ *last_token* (vecteur de P entiers) : l'image du dernier vecteur du jeton $Vtok$ connue par s_i.

Les deux types de message de cet algorithme sont toujours les messages de requête et de jeton incluant respectivement les informations suivantes :

- *request* ($< p, d >$) : p est la priorité courante de la requête ; d est la distance séparant le nœud initiateur de la requête et le nœud destinataire du message.
- *token* ($< p, d, Vtok >$) : p est la priorité courante de la requête incluse dans le jeton (valeur *nil* si aucune requête) ; d est la distance séparant le nœud initiateur de la requête et le destinataire du message de jeton ; $Vtok$ est un vecteur de compteurs du nombre global de requêtes émises.

Les fonctions *add*(s, p, l, d), *dequeue(Q)*, *head(Q)* et *reorder(Q)* qui manipulent la file Q restent inchangées.

Description de l'algorithme

En appelant la procédure *Request_CS* (ligne 22), s'il ne possède pas le jeton, s_i ajoute sa requête dans sa file locale (ligne 26). Si la requête est la tête de file (donc la plus prioritaire localement), s_i envoie un message de requête au site père (ligne 28). D'autre part, si s_i ne retransmet pas la requête (lignes 30 et 33), il enregistre la requête en incrémentant l'entrée correspondante du vecteur *pending*. Enfin si s_i possède le jeton, il entre directement en section critique (ligne 34).

En sortant de la section critique, la procédure *Release_CS* (ligne 37) met à jour la priorité des requêtes dans la file locale de s_i en appelant la fonction *UpdateLocalQueue* décrite ci-après. Elle actualise le vecteur du jeton en ajoutant le nombre de requêtes pendantes de chaque priorité (ligne 41) et remet à zéro les compteurs du vecteur *pending* (ligne 42). Si la file locale n'est pas vide, s_i envoie le jeton à s_{next}, le nœud en tête de file (ligne 46) et efface cette requête de la file (ligne 44). Si après la suppression de cette requête, la file demeure non vide, le processus inclut dans le jeton la première requête de la file. La variable $father$ pointe désormais sur s_{next} (ligne 47).

À la réception d'une requête provenant d'un site s_j (ligne 57), si le nœud courant est le nœud racine mais n'est pas en section critique, le compteur correspondant du vecteur du jeton est incrémenté. Le jeton est ensuite renvoyé (ligne 61) vers s_j et la variable $father$ pointe désormais sur s_j. D'autre part, si le nœud courant n'est pas le site racine ou n'est pas en section critique, la nouvelle requête est enregistrée dans la file locale Q s'il n'existe pas de requête en provenance de s_j dans Q (ligne 72). Si au contraire, il existe une requête en provenance de s_j dans Q, la requête concernée est mise à jour en actualisant les valeurs de priorité et de distance reçues (lignes 65 à 66). La requête est ensuite retransmise au père si il n'est pas la racine et si la tête de file a changé (ligne 74) ; sinon, la requête est comptabilisée dans le vecteur *pending* (ligne 77). Contrairement à l'algorithme "retard-

distance", on peut remarquer que l'incrémentation des priorités ne se fait plus au moment où le site reçoit la requête. La ligne 79 permet de comptabiliser tout de même la requête si le message de requête provenant de s_j a croisé le message de jeton que le processus courant a déjà envoyé en direction de s_j. Si cette comptabilisation n'était pas faite alors la requête correspondante n'aurait jamais été prise en compte.

De manière similaire à la procédure $Release_CS$, lorsqu'un site s_i reçoit le jeton (ligne 81), il ajoute le vecteur *pending* au vecteur $Vtok$ du jeton (ligne 85). La file locale est alors mise à jour par l'appel à la fonction *UpdateLocalQueue* (line 86) dont le paramètre est la différence entre la nouvelle valeur de $Vtok$ et l'ancienne valeur de $Vtok$ contenue dans $last_token$. Cette différence représente pour s_i les requêtes qui n'ont pas encore été prises en compte pour incrémenter les priorités des requêtes de la file locale. La variable $last_token$ est alors mise à jour et le vecteur *pending* remis à zéro. Si une requête est incluse dans le jeton ($p_j \neq nil$), s_i inclut cette requête dans la file locale (ligne 90). Si sa propre requête est la tête de file (c'est à dire la requête à la plus forte priorité), le processus peut alors entrer en section critique (ligne 73). Sinon le jeton est retransmis au processus en tête de file locale (ligne 95). La variable $father$ pointe désormais vers le destinataire du jeton (ligne 96).

Mise à jour de la file locale (fonction *UpdateLocalQueue*)

Comme mentionné précédemment, un site met à jour les priorités des requêtes enregistrées dans sa file locale en appelant la fonction *UpdateLocalQueue* (ligne 9) à chaque fois qu'il reçoit le jeton ou qu'il libère la section critique.

Pour chaque priorité p_i du vecteur paramètre V, la fonction incrémente de un le niveau de retard l de chaque requête req de la file locale qui possède une priorité inférieure à p_k (ligne 15). Similairement à l'algorithme "retard-distance", à chaque fois que la valeur de ce niveau atteint $\mathcal{F}(p_k + 1)$, la priorité de req est incrémentée et le niveau de retard remis à zéro. La première condition du test de la ligne 15 empêche la priorité de req d'être supérieure à la variable de boucle p_k variant de $p_{min}+1$ à p_{max}. La seconde condition du test permet de résoudre le problème de famine induit par le mécanisme de prise en compte de la topologie (voir section 4.3.4). Comme dans l'algorithme "retard-distance", une priorité d'une requête peut être localement incrémentée au dessus de la priorité maximale de la file locale (bornée par p_{max}, la priorité maximale du système) et ainsi à terme préempter les processus de même priorité de distance plus petite.

4.6 Évaluation des performances de l'algorithme *Awareness*

Nous comparons dans cette section, l'algorithme Awareness avec les algorithmes de Kanrar-Chaki, Chang et "retard-distance".

4.6.1 Protocole d'évaluation

Les expériences ont été menées sur la même plate-forme que les expériences de la section 4.4. Les paramètres d'expérience sont à grande majorité les mêmes que la section 4.4 sauf pour :

- N : le nombre de processus est désormais de 64. On peut ainsi étudier le comportement des différents algorithmes avec un nombre plus important de processus.
- α : temps d'exécution d'une section critique est maintenant égal à 2,5 ms.
- γ : le temps d'acheminement d'un message entre deux processus voisins maintenant égal à 2,5 ms. Ceci nous permet de constater le comportement des algorithmes lorsque la latence réseau n'est pas négligeable par rapport à α.

La topologie considérée est de type "centre prioritaire" (voir section 4.4.4) dans un arbre binaire complet. Ceci implique donc que désormais le nombre de priorités considérées est de 6 ($Log_2(64)$). Nous considérons deux valeurs de charge : haute ($\rho = 0,1N$) et intermédiaire ($\rho = 0,5N$).

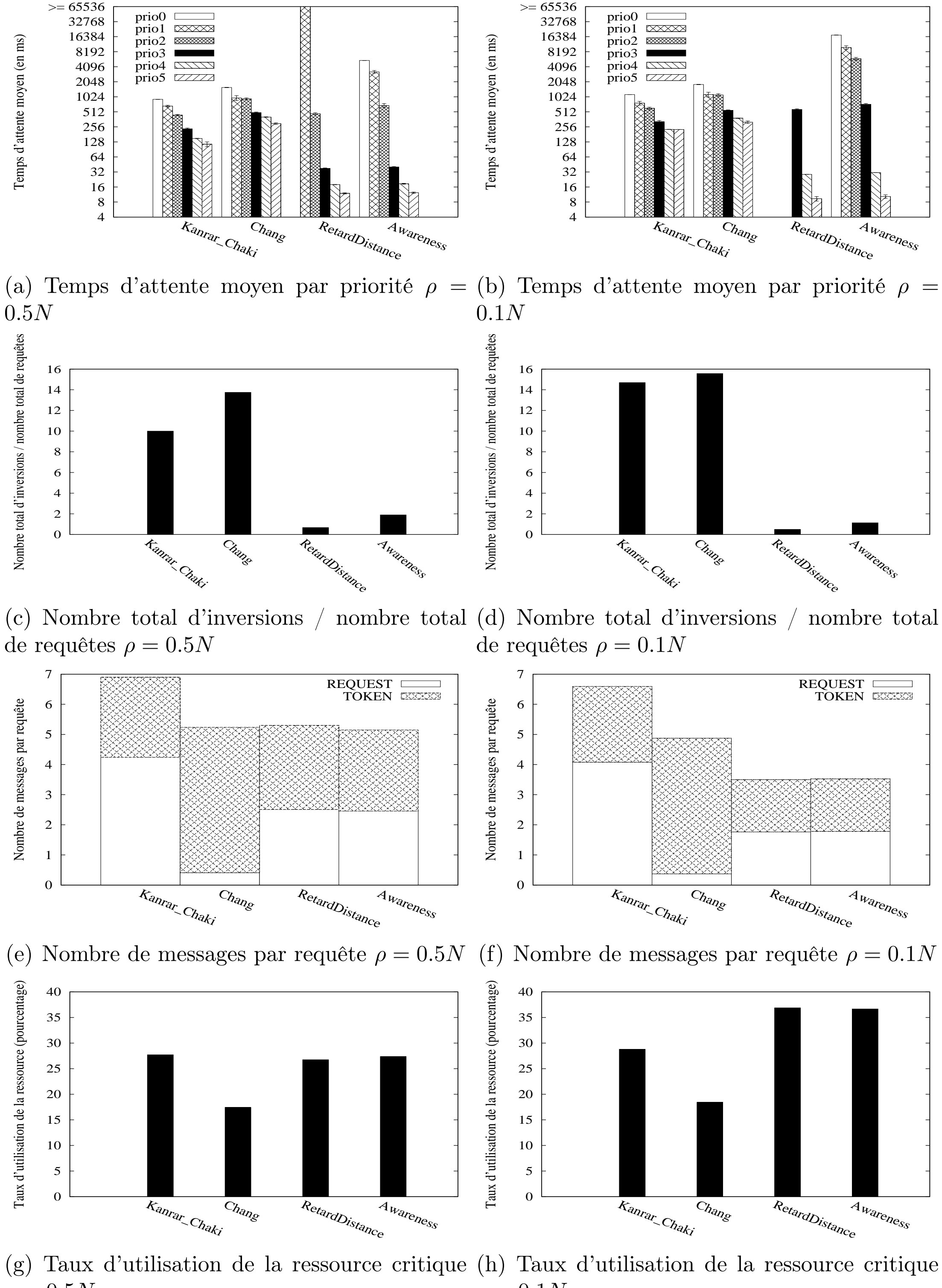
4.6.2 Résultats pour une fonction de palier donnée

Dans cette section, nous considérons la même fonction de palier que la section 4.4.4 à savoir $\mathcal{F}(p) = 2^{p+c}$ avec $c = 6$. Nous étudions également les mêmes métriques. La figure 4.14 montre pour deux valeurs de charge considérées (moyenne charge à gauche et forte charge à droite) les résultats concernant ces métriques.

Temps de réponse moyen

Dans les figures 4.14(b) et 4.14(a), nous pouvons observer le même comportement que dans la section 4.4.4 pour les algorithmes de Chang et de Kanrar-Chaki. Cependant l'algorithme "retard-distance" diffère car nous pouvons observer que les petites priorités (0,1 et 2 pour $\rho = 0,5N$ et priorité 0 $\rho = 0,1N$) ont un temps de réponse supérieur à la durée d'expérimentation. L'algorithme "retard-distance" pénalise trop les petites priorités. Nous nommerons un tel temps d'attente une "pseudo-famine". En effet, la famine réelle ne peut en théorie jamais arriver mais les requêtes à faibles priorités sont satisfaites après un très long temps d'attente. L'algorithme "retard-distance" ne passe donc pas à l'échelle en termes de temps d'attente à charge haute et moyenne dans la configuration "centre prioritaire".

Nous pouvons remarquer que sur l'ensemble des algorithmes, le temps de réponse des priorités faibles et intermédiaires est globalement plus haut en forte charge qu'en charge intermédiaire. En effet en charge intermédiaire, les priorités faibles et intermédiaires ont plus de chances d'être satisfaites puisque la fréquence des requêtes à hautes priorités est moins importante qu'en forte charge. En comparant les algorithmes "retard-distance" et "Awareness", nous observons que les hautes priorités (4 et 5) présentent presque le même temps de réponse dans les deux algorithmes. D'autre part, les priorités intermédiaires (2 et 3) sont plus pénalisées dans l'algorithme "Awareness" que dans l'algorithme "retard-distance" tandis que les petites priorités (0 et 1) sont beaucoup moins pénalisées dans l'algorithme "Awareness".



(a) Temps d'attente moyen par priorité $\rho = 0.5N$

(b) Temps d'attente moyen par priorité $\rho = 0.1N$

(c) Nombre total d'inversions / nombre total de requêtes $\rho = 0.5N$

(d) Nombre total d'inversions / nombre total de requêtes $\rho = 0.1N$

(e) Nombre de messages par requête $\rho = 0.5N$

(f) Nombre de messages par requête $\rho = 0.1N$

(g) Taux d'utilisation de la ressource critique $\rho = 0.5N$

(h) Taux d'utilisation de la ressource critique $\rho = 0.1N$

FIGURE 4.14 – Performances de l'algorithme "Awareness" en charge moyenne ($\rho = 0.5N$) et en charge haute ($\rho = 0.1N$)

Inversions de priorités

Les figures 4.14(d) et 4.14(c) montrent une grande différence entre nos algorithmes et les algorithmes de Chang et Kanrar-Chaki en termes de quantité d'inversions de priorité. Cet écart est dû au mécanisme de retard d'incrémentation. On remarque cependant un léger surcoût pour l'algorithme "Awareness" par rapport à l'algorithme "retard-distance". Ceci est cohérent car l'algorithme "Awareness" favorise plus facilement les requêtes de faibles priorités. Lorsque la charge diminue (Figure 4.14(c)), le nombre d'inversions de priorités diminue pour les algorithmes de Chang et Kanrar-Chaki. Cette réduction s'explique par la réduction du nombre de requêtes diminue également impliquant potentiellement moins d'inversions. On observe l'inverse pour l'algorithme "Awareness" : la quantité d'inversions par requête augmente. Lorsque la charge est moins importante, la probabilité que le jeton s'éloigne du centre est plus importante car le débit d'entrée de requêtes de forte priorité diminue. Comme la latence réseau (paramètre γ) n'est pas négligeable par rapport au temps de section critique α le temps pour qu'une requête à forte priorité atteigne le porteur du jeton est accru. Pendant ce transfert, des requêtes de priorités plus faible, proche du jeton, peuvent alors être satisfaites.

Nombre de messages par requête

Les figures 4.14(f) et 4.14(e) montrent que pour tous les algorithmes, le nombre total de messages par requête est plus important en charge intermédiaire qu'en haute charge.

Analysons en premier lieu les messages de requête. Dans la configuration "centre prioritaire", une réception de requête sur un site s_i concerne une requête qui a très probablement une priorité inférieure à la priorité de s_i (les fils d'un processus sont la plupart du temps des processus en zone moins prioritaire). Par conséquent en cas de forte charge, les requêtes ont moins de chances d'être retransmises dans la zone supérieure de l'arbre puisque les algorithmes ne retransmettent une requête que les requêtes de plus haute priorité. Ainsi le nombre de messages de requête est réduit en forte charge.

Analysons maintenant les messages contenant le jeton. Le transfert du jeton depuis une zone de l'arbre de priorité p vers une zone de priorité inférieure à p a lieu dans deux cas :

(1) soit il n'y a plus de requête pendante dans la zone de priorité p

(2) soit il existe au moins une requête de priorité inférieure à p qui a été incrémentée dans la zone de priorité p et qui est devenue égale à p.

En cas de forte charge, le jeton a beaucoup moins de chances de quitter la zone de haute priorité pour aller dans une zone de priorité inférieure à cause du second cas. Le jeton voyage principalement dans le centre du graphe (les zones les plus prioritaires). En charge intermédiaire, le premier cas peut se produire plus fréquemment (absence de requête pendante) ce qui implique que le jeton peut plus facilement quitter le centre du graphe. La conséquence directe est une augmentation du nombre de messages de jeton. Si nous considérons les deux types de messages, en cas de forte charge, les algorithmes "Retard-distance", "Awareness" et Chang présentent un nombre de messages moins important que l'algorithme de Kanrar-Chaki ceci grâce à la possibilité d'inclure le jeton dans une requête tel que cela a été proposé par Chang (voir section 4.3.2). D'autre part, les algo-

rithmes "Retard-distance" et "Awareness" présentent un nombre de messages inférieur à l'algorithme de Chang grâce à :

- l'exploitation de la localité des requêtes (voir section 4.3.4),
- un meilleur respect des priorités ; dans l'algorithme Chang, les processus atteignent rapidement la priorité maximale induisant plus de transferts de jeton dans la topologie.

En comparant les deux figures 4.14(e) et 4.14(f), en cas de charge moyenne, les algorithmes "Retard-distance" et "Awareness" présentent une augmentation en messages d'environ 50% tandis que les algorithmes de Chang et Kanrar-Chaki ont une augmentation de seulement 5%. Nous avons remarqué que la diminution de la charge induit une augmentation du nombre de messages. Nous pouvons alors souligner que cette baisse de charge a un impact plus important sur les algorithmes "Retard-distance" et "Awareness" que les algorithmes de Chang et Kanrar-Chaki.

Taux d'utilisation de la ressource

Puisque le paramètre de latence réseau est comparable au temps de section critique ($\gamma \simeq \alpha$), la valeur maximum théorique du taux d'utilisation est de 50 % . Ainsi, dans le meilleur des cas (chaque réception de jeton implique le début d'une section critique), le jeton passe au moins autant de temps dans le réseau qu'à être utilisé. De manière générale, les algorithmes de Chang et Kanrar-Chaki sont moins efficaces en cas de forte charge et plus particulièrement pour l'algorithme de Chang. Ce comportement est une conséquence directe de son nombre important d'inversions de priorités : les processus atteignent rapidement la priorité maximale impliquant que le jeton a plus de chance de voyager sur des plus longues distances entre deux sections critiques, ce qui dégrade le taux d'utilisation de la ressource lorsque la latence réseau n'est pas négligeable par rapport au temps d'exécution de la section critique.

Synthèse

Le mécanisme de retard d'incrémentation de priorité et le mécanisme permettant à tout site de connaître l'ensemble des requêtes émises réduisent de manière importante le nombre d'inversions de priorités (jusqu'à 10 fois en cas de moyenne charge et 15 fois en cas de forte charge). Ces mécanismes n'induisent pas de surcoût en messages lorsqu'on les compare à l'algorithme de Chang. De plus, l'algorithme de Chang est moins efficace en termes de taux d'utilisation de la ressource. Enfin notre algorithme "Awareness" réduit de manière significative le temps de réponse des requêtes à faible priorité tout en gardant une faible quantité d'inversions. De plus il ne dégrade ni la complexité en message ni le taux d'utilisation lorsqu'on le compare à l'algorithme "retard-distance"

4.6.3 Impact de la fonction de palier sur les inversions et le temps de réponse

Dans cette section, nous évaluons l'impact de la fonction de palier sur les quatre algorithmes précédemment considérés. Nous définissons donc cinq familles de fonction de

palier ;

- constante : $\mathcal{F}_c(p) = c$
- linéaire : $\mathcal{F}_c(p) = p * c$
- polynomiale : $\mathcal{F}_c(p) = p^c$
- exponentielle : $\mathcal{F}_c(p) = c^p$
- puissance de deux : $\mathcal{F}_c(p) = 2^{p+c}$ (famille considérée jusqu'à présent avec $c = 6$)

Pour une valeur de charge donnée et une constante c donnée, nous avons mesuré le nombre d'inversions et le temps de réponse maximum. En faisant varier la constante c entre chaque expérience, nous étudions le comportement des algorithmes "retard-distance" et "Awareness" pour différentes fonctions.

La figure 4.15 montre cette étude pour une charge haute et intermédiaire. Chaque sous-figure montre l'impact sur les algorithmes d'une famille de fonction pour une charge donnée. Les coordonnées d'un point correspond aux métriques suivantes : l'axe des abscisses représente le rapport du nombre d'inversions sur le nombre de requêtes et l'axe des ordonnées représente le temps de réponse maximum toute priorité confondue. Ainsi pour un rapport inversions/requêtes donné, nous pouvons comparer le temps de réponse maximum entre les différents algorithmes. L'objectif de la fonction de palier est de trouver un point qui est le plus proche de l'origine des deux axes (coin inférieur gauche).

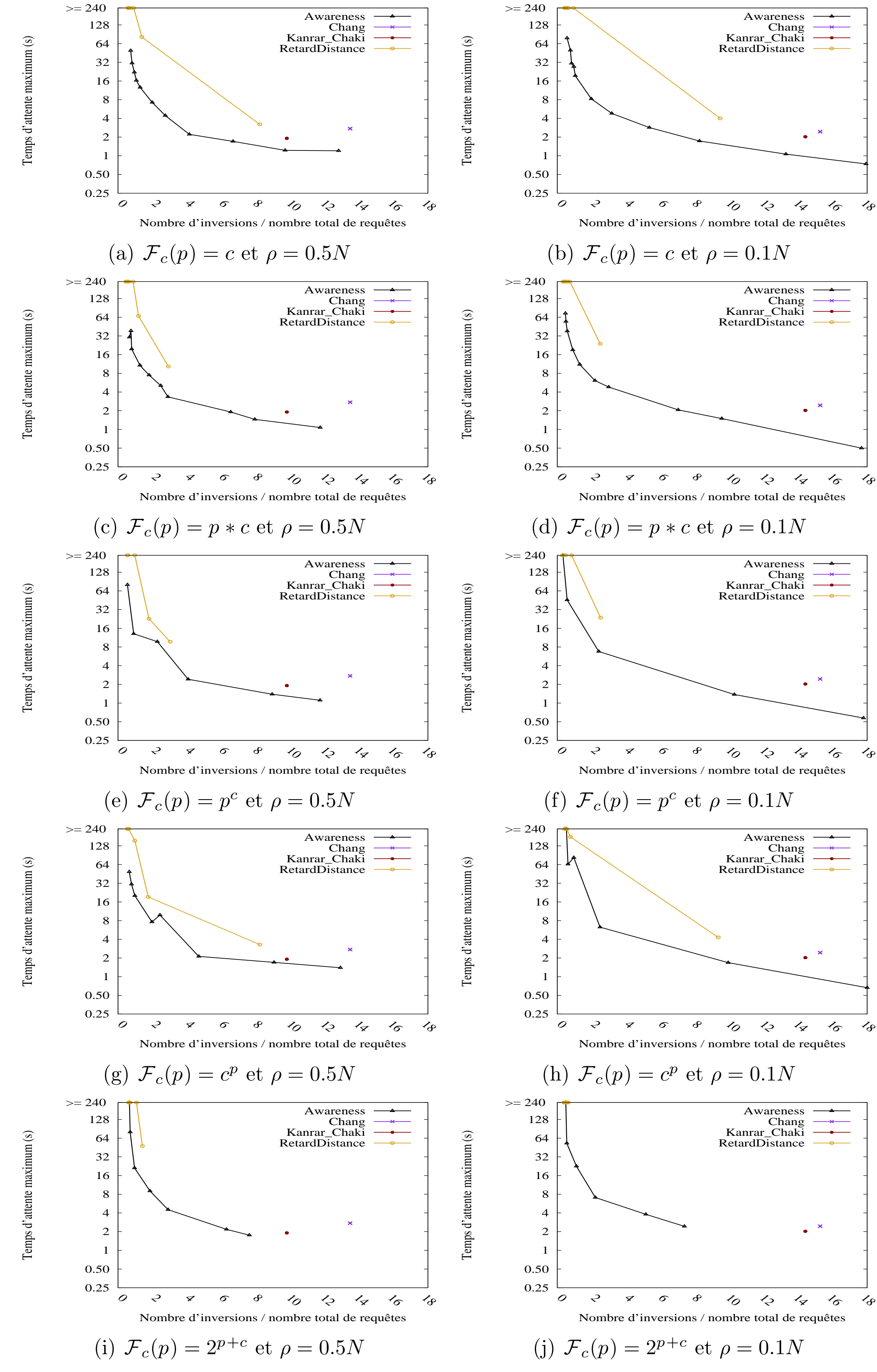
Nous observons pour les algorithmes "retard-distance" et "Awareness" différentes valeurs d'inversion. Cependant pour une valeur donnée de la constante c, l'intervalle de valeurs pour les deux courbes ne sont pas les mêmes, impliquant une comparaison difficile. Nous avons aussi inclus dans les sous-figures, les algorithmes de Chang et de Kanrar-Chaki afin d'être exhaustif dans la comparaison. Ces deux algorithmes sont représentés par un point puisqu'ils n'utilisent pas de fonction de palier. Par conséquent, ces points sont les mêmes pour chaque sous-figure d'une charge donnée.

Dans l'ensemble, on peut observer que pour une charge quelconque et une fonction de palier quelconque, l'algorithme "Awareness" a un temps de réponse moins important que l'algorithme "retard-distance" (au moins 2 fois moins). Il est intéressant de noter que pour une charge donnée, toutes les courbes de l'algorithme "Awareness" sont presque similaires, indépendamment de la famille de fonction. En fait, son mécanisme de connaissance globale permet de réduire l'incidence du choix de la famille de fonction de palier.

Enfin, sur l'algorithme "Awareness", pour une fonction de palier donnée, on peut remarquer qu'en charge intermédiaire la pente de la courbe est légèrement plus importante qu'en charge haute (entre les valeurs 2 et 4 de l'abscisse, on constate une pente plus importante en charge intermédiaire). Ceci illustre l'influence de la charge sur l'effet de la fonction de palier. Plus la charge est haute, plus le nombre de requêtes de haute priorité sera grand et donc plus l'indice de retard augmentera rapidement. Par conséquent la dégradation du temps de réponse est moins rapide en forte charge.

4.7 Conclusion

Dans ce chapitre nous avons présenté des mécanismes permettant de construire un nouvel algorithme d'exclusion mutuelle à priorité. L'algorithme final "Awareness" remplit les objectifs définis en section 4.1, à savoir :



(a) $\mathcal{F}_c(p) = c$ et $\rho = 0.5N$

(b) $\mathcal{F}_c(p) = c$ et $\rho = 0.1N$

(c) $\mathcal{F}_c(p) = p * c$ et $\rho = 0.5N$

(d) $\mathcal{F}_c(p) = p * c$ et $\rho = 0.1N$

(e) $\mathcal{F}_c(p) = p^c$ et $\rho = 0.5N$

(f) $\mathcal{F}_c(p) = p^c$ et $\rho = 0.1N$

(g) $\mathcal{F}_c(p) = c^p$ et $\rho = 0.5N$

(h) $\mathcal{F}_c(p) = c^p$ et $\rho = 0.1N$

(i) $\mathcal{F}_c(p) = 2^{p+c}$ et $\rho = 0.5N$

(j) $\mathcal{F}_c(p) = 2^{p+c}$ et $\rho = 0.1N$

FIGURE 4.15 – Étude de cinq familles de fonction de palier en charge moyenne ($\rho = 0.5N$) et en charge haute ($\rho = 0.1N$)

- **le respect de la vivacité** en se basant sur l'algorithme de Kanrar-Chaki .
- **une limitation des inversions de priorités** : le mécanisme de retard réduit considérablement le taux d'inversions avec une fonction de palier exponentielle comme nous avons pu l'analyser dans la section 4.4 .
- **une complexité en messages comparable à l'existant** : le mécanisme de retard augmentait de facto le nombre de messages par requête mais grâce au mécanisme "distance" qui tient compte de la localité de requête et au mécanisme d'inclusion d'une requête dans le jeton, cette complexité a pu être réduite.
- **une limitation du temps d'attente maximum** : l'algorithme "Awareness" permet à n'importe quel processus, quelle que soit sa position dans la topologie, de connaître l'ensemble des requêtes émises dans le système. Grâce à ce mécanisme, il est possible de réduire considérablement le temps d'attente maximum pour un taux d'inversions équivalent.

Enfin, l'utilisateur de l'algorithme peut ajuster le compromis entre le temps d'attente et le taux d'inversion grâce à la fonction de palier $\mathcal{F}$: plus cette fonction est croissante, plus le taux d'inversions diminuera et plus le temps d'attente des petites priorités augmentera.

Chapitre 5

Exclusion mutuelle avec dates d'échéance

Sommaire

5.1 Introduction **69**
5.2 Motivations **70**
5.2.1 Cloud Computing 70
5.2.2 Service Level Agreement 71
5.3 Description de l'algorithme **71**
5.3.1 Description générale 71
5.3.2 Contrôle d'admission 74
5.3.3 Mécanisme de préemption 77
5.4 Évaluation des performances **81**
5.4.1 Protocole d'évaluation 81
5.4.2 Métriques 82
5.4.3 Impact global 82
5.4.4 Impact de la préemption pour une charge donnée 84
5.4.5 Impact de la charge 86
5.5 Conclusion **86**

5.1 Introduction

Dans ce chapitre nous considérons une extension du problème de l'exclusion mutuelle qui consiste à satisfaire les requêtes des processus en fonction de contraintes temporelles. Plus précisément, nous considérons que les requêtes incluent une date d'échéance et le but de notre algorithme est d'assurer qu'un maximum de requêtes soient satisfaites avant la date d'échéance.

Pour réduire le plus possible le nombre de requêtes non satisfaites avant leur échéance, l'algorithme contrôle dans un premier temps que la contrainte de la requête soit réalisable. Lors de cette phase de contrôle, l'algorithme devra tenir compte de l'ensemble des requêtes

pendantes. Une fois que cette phase de contrôle d'admission est positive, la requête est définitivement acceptée : le système s'engage donc à la satisfaire avant sa date d'échéance. À l'inverse, si l'admission est négative, alors la requête est définitivement rejetée et ne donnera pas lieu à une section critique : le processus est alors obligé de refaire une nouvelle requête en donnant éventuellement des contraintes plus faibles.

Ce type d'exclusion mutuelle avec contrainte s'inscrit dans un modèle plus général d'application avec SLA où un contrat de qualité de service est négocié avec le fournisseur. Les applications s'exécutent souvent dans le modèle du Cloud Computing où il a été reconnu dans [AFG+09] que le verrouillage de ressources était un défi majeur.

Ce chapitre décrit en section 5.2 les particularités du modèle du Cloud Computing permettant de motiver cette contribution. L'algorithme sera ensuite décrit avec ses deux mécanismes principaux en section 5.3 : le mécanisme de contrôle d'admission et un mécanisme de préemption permettant d'améliorer le débit de section critique. S'ensuivra ensuite une évaluation de performances en section 5.4.

Cette contribution a donné lieu à deux publications, [CCgrid12] et [CCgrid13] dont la deuxième a été faite en collaboration avec les membres du projet ANR MyCloud [MyC].

5.2 Motivations

5.2.1 Cloud Computing

Dans les Clouds, les ressources sont considérées comme des services qu'il s'agisse de matériel (ex : serveur, routeur, GPU ...), de plates-formes de développement ou de logiciel (ex : webmail, office). Ceci permet de ne plus acheter (ou louer) ces ressources mais de payer uniquement que ce que l'on consomme (informatique à la demande). Il existe plusieurs niveaux de couche de service. À l'instar de la pile OSI pour les réseaux, une couche d'un niveau k fournit un service pour la couche de niveau k+1. Les trois couches principales dans ce modèle ([LKN+09, RCL09, ZCB10]) sont par ordre de niveau croissant :

- **IaaS** : "Infrastructure as a Service", permet de fournir des ressources matérielles virtualisées. Par exemple, Amazon EC2 [EC2] fournit des machines serveurs virtuelles et Amazon S3 [S3] fournit des ressources pour le stockage.
- **PaaS** : "Platform as a Service", fournit une interface de programmation pour une application qui s'exécutera dans un Cloud. Par exemple, Google offre Google App engine [App] qui est une plate-forme de conception et d'hébergement. d'applications web.
- **SaaS** : "Software as a Service", fournit une application pour les utilisateurs finaux (exemple : bureautique en ligne) ou bien pour des développeurs d'applications de haut niveau. Cette couche repose sur une couche PaaS mais peut aussi utiliser directement la couche IaaS.

Les ressources étant mutualisées entre les différents utilisateurs du système, des mécanismes de verrouillage sont donc nécessaires pour éviter toute incohérence ou dysfonctionnement. Ces mécanismes peuvent être présents à différents niveaux. En IaaS, ils servent à réserver des ressources à accès exclusif. En PaaS, ils servent pour la programmation d'applications réparties (exemple : MPI-IO). Nous pouvons citer comme exemple le service

de verrouillage de Chubby [Bur06] proposé par Google qui assure une fiabilité de service grâce à une réplication de serveurs.

5.2.2 Service Level Agreement

Le Cloud Computing s'accompagne d'un modèle commercial dans lequel le client n'achète plus de machine ou de logiciel mais un service donné avec une qualité négociée. Il doit donc convenir d'un contrat avec le fournisseur qui définit un niveau et un type de qualité de service que ce dernier doit respecter. Ce contrat est plus communément appelé SLA (Service Level Agreement) et fait suite à une phase de négociation entre le client et le fournisseur (gérée usuellement par un contrôle d'admission) or les algorithmes distribués d'exclusion mutuelle ne prennent pas (ou peu) en compte la notion de QoS associée aux requêtes clientes. Notre but est donc de fournir une API pour un service de verrouillage intégrant la notion de SLA. L'algorithme décrit dans ce chapitre pourrait être ainsi appliqué sur des plates-formes de Cloud de type PaaS.

5.3 Description de l'algorithme

Cette section décrit notre algorithme d'exclusion mutuelle permettant de satisfaire les requêtes avant leur date d'échéance. Cet algorithme doit :

- éviter les violations de SLA, c'est-à-dire les requêtes non satisfaites avant leur date d'échéance (exigence du client).
- satisfaire le plus de requêtes possibles afin d'avoir un bon débit de section critique (exigence du fournisseur).

Notre algorithme se base sur l'algorithme de Raymond décrit en section 2.5.1. On décrira dans un premier temps les principes de base de l'algorithme puis le fonctionnement des deux mécanismes principaux :

- **le contrôle d'admission** qui permet de décider si une requête peut être satisfaite ou non,
- **le mécanisme de préemption** qui permet de décider du chemin que le jeton suivra dans la topologie afin d'en rentabiliser son utilisation.

Notre algorithme implique des changements au niveau applicatif. Désormais lorsqu'un processus demande la section critique, il fournit deux valeurs de contraintes :

- la date d'échéance d'accès à la section critique,
- le temps de section critique requis

De plus, la requête doit être soumise à un contrôle d'admission avant d'être définitivement acceptée. Par conséquent, la procédure *Request_CS* devra renvoyer à l'application un résultat booléen indiquant si le processus peut exécuter la section critique ou au contraire si sa requête ne peut être acceptée.

5.3.1 Description générale

Comme dans l'algorithme original de Raymond, notre algorithme repose sur la circulation d'un unique jeton dans une topologie en arbre statique où le nœud racine est le

```
1  Local variables :
2  begin
3      father : site ∈ Π or nil;
4      Q : queue of (s_i, s_n, t, α, d) ∈ Π × Π × T × ℕ × ℕ ;
5      state ∈ {tranquil, requesting, inCS}
6  end
7  Initialization
8  begin
9      Q ← ∅;
10     state ← tranquil;
11     father ← according to the initial topology;
12 end
13 Request_CS(t_dead ∈ T, α ∈ ℕ) with boolean result
14 begin
15     state ← requesting;
16     boolean accept;
17     accept ← controlReqCS(t_dead, α);
18     if accept = false then
19         state ← tranquil;
20         return false;
21         /* Cancel : rejected */
22     if father ≠ nil then
23         wait(father = nil or t_now > t_dead);
24         if t_now > t_dead then
25             state ← tranquil;
26             Q ← Q − {(s_i, s_n, t, α, d) ∈ Q | s_i = self} ;
27             return false;
28             /* Cancel : too late */
29     state ← inCS;
30     return true;
31     /* CRITICAL SECTION */
32 end
33 Release_CS()
34 begin
35     state ← tranquil;
36     Q ← Q − {(s_i, s_n, t, α, d) ∈ Q | s_i = self ∨ t < t_now}
       ;
37     if Q ≠ ∅ then
38         (s_inext, s_nnext, t_next, α_next, d_next) ←
           preempt();
39         send Token(Q) to s_nnext;
40         father ← s_nnext;
41         Q ← Q − {(s_i, s_n, t, α, d) ∈ Q | s_n = father} ;
42 end
43 Receive_Request(s_i ∈ Π, t_deadj ∈ T, α ∈ ℕ, d_j ∈ ℕ)
   from s_j
44 begin
45     if father = nil and state = tranquil then
46         send Token(∅) to s_j;
47         father ← s_j;
48     else if father ≠ s_j then
49         Q ← Q − {(s'_i, s'_n, t', α', d') ∈ Q | s'_i = s_i} ;
50         controlRecvReq(s_i, s_j, t_deadj, α, d_j);
51 end
52 Receive_Token(Q_j) from s_j
53 begin
54     father ← nil;
55     controlRecvTok(Q_j, s_j);
56     Q ← Q − {(s_i, s_n, t, α, d) ∈ Q | t < t_now} ;
57     if Q ≠ ∅ then
58         (s_inext, s_nnext, t_next, α_next, d_next) ←
           preempt();
59         if s_next = self then
60             notify(father = nil);
61         else
62             send Token(Q) to s_nnext;
63             father ← s_nnext;
64             Q ← Q − {(s_i, s_n, t, α, d) ∈ Q | s_n =
               father} ;
65 end
```

FIGURE 5.1 – Algorithme à dates d'échéances

détenteur du jeton. De plus, chaque site maintient une file locale de requête. Le pseudo code de notre algorithme est donné dans la figure 5.1.

Variables

Les variables locales à chaque processus restent inchangées par rapport à l'algorithme de Raymond hormis pour la file locale qui est un ensemble de tuples de type $(s_i, s_n, t, \alpha, d) \in \Pi \times \Pi \times T \times \mathbb{N} \times \mathbb{N}$ où

- s_i est l'identifiant du site initiateur de la requête,
- s_n est l'identifiant du voisin qui a transmis la requête, (similaire à Raymond)
- t est la date d'échéance de la requête associée,
- α est le temps de section critique requis en unité de temps,
- d est la distance en nombre de liens intermédiaires séparant l'initiateur de la requête et le nœud courant. Ce paramètre est utilisé pour le mécanisme de préemption.

Chaque file locale est ordonnée selon une politique *Earliest Deadline First* (EDF) c'est-à-dire par ordre de dates d'échéance croissantes. En cas de dates égales, l'ordre appliqué est celui du temps de section critique requis croissant. On préfère ainsi favoriser la requête qui demande moins de temps de section critique. Cependant l'ordre de satisfaction des requêtes ne dépend pas uniquement de cet ordre mais dépend aussi de la politique de préemption. Les différentes politiques de préemptions seront détaillées section 5.3.3.

Messages

Nous considérons deux types de message :

- $request(s_i, t_{dead}, \alpha, d)$: message de requête contenant l'identifiant du nœud initiateur, les deux valeurs de contrainte et la distance en nombre de liens entre le nœud receveur du message et le nœud initiateur.
- $token(Q_j)$: message de jeton qui contient la file des requêtes à traiter. Nous verrons ultérieurement que la file du porteur du jeton est un élément clé dans notre algorithme.

Fonctions

Pour avoir un maximum de modularité dans les différentes politiques de décision, nous considérons que les mécanismes de contrôle d'admission et de préemption qui seront détaillés respectivement en section 5.3.2 et 5.3.3 fournissent l'API suivante :

- Pour le contrôle d'admission :
  - **controlReqCS** (ligne 17) : appelée au moment où une nouvelle requête est émise. Cette procédure retourne un résultat booléen qui vaut vrai si la nouvelle requête a été acceptée, faux sinon. Elle retournera également faux si la date d'échéance de la requête a expiré.
  - **controlRecvReq** (ligne 50) : appelée lorsque le processus reçoit un message de requête, cette fonction permet de contrôler si la requête reçue peut être validée localement ou non.
  - **controlRecvTok** (ligne 55) : appelée lorsque le processus reçoit le jeton, cette fonction permet de fusionner la file du jeton avec la file locale du site courant.

  Le contrôle d'admission a la responsabilité de transmettre les messages de requêtes ainsi que les messages nécessaires pour son fonctionnement. En cas de réponse positive sur une requête, le contrôle d'admission assure au processus qu'elle est enregistrée dans le système et qu'elle sera satisfaite avant la date d'échéance.
- Pour le mécanisme de préemption :
  - **preempt** : appelée au moment où un site décide d'un prochain destinataire pour envoyer le jeton soit lors d'une sortie de section critique, soit lors de la réception du jeton. Cette fonction renvoie le tuple de la file locale correspondant à la requête qui peut réquisitionner le jeton.

Description

En exécutant la procédure *Request_CS* (ligne 14) le site appelle en premier lieu la fonction **controlReqCS**(t_{dead}, α) (ligne 17) en passant en paramètre les deux valeurs de

contrainte. Si le résultat de cette fonction est négatif alors le processus se remet à l'état *tranquil* et abandonne sa requête (lignes 18 à 20). Sinon, si le résultat est positif, le processus se met en attente du jeton (ligne 23). Cependant, si le jeton n'arrive pas avant la date d'échéance, il y a violation de SLA : la requête est alors abandonnée et ne donnera pas lieu à une section critique (lignes 24 à 27). En effet, ce cas est possible en cas d'erreur d'estimation du contrôle d'admission. Nous annulons donc les requêtes qui ne sont pas satisfaites à leur date d'échéance pour éviter l'effet cascade sur les autres requêtes et ne pas dégrader ainsi le taux de violations.

À la réception d'un message de requête (ligne 44) réclamé par s_i, si le processus possède le jeton mais ne l'utilise pas, le jeton est alors envoyé en direction du demandeur (lignes 45 à 47). Sinon, en ligne 49, le processus efface de sa file locale une requête obsolète de s_i qui pourrait éventuellement subsister et appelle ensuite la fonction **controlRecvReq** du contrôle d'admission en passant en paramètre les informations transmises dans le message de requête.

Lorsqu'un processus reçoit le jeton (ligne 53), il appelle la fonction **controlRecvTok** du contrôle d'admission en lui passant en paramètre la file du jeton (ligne 55). Le traitement qui suit est alors similaire à la méthode *Release_CS*. Afin d'éviter de stocker indéfiniment des requêtes qui auraient manqué leur date d'échéance, le processus efface de sa file locale toute requête obsolète (lignes 36 et 56). Si la file demeure non vide alors le processus cherche un prochain destinataire du jeton avec le mécanisme de préemption en appelant la méthode **preempt** (lignes 38 et 58). Le jeton est ensuite envoyé (ligne 62) ou bien éventuellement utilisé par le processus (ligne 60).Afin d'assurer qu'un processus ne connaisse pas les requêtes provenant de son père (invariant de l'algorithme de Raymond), le processus efface ces dernières de sa file locale (lignes 41 et 64) à chaque transmission du jeton. Notons que lorsque le site entre en section critique, sa requête n'est pas effacée de la file locale (contrairement à l'algorithme de Raymond) car ses contraintes peuvent encore servir pour la décision du contrôle d'admission.

5.3.2 Contrôle d'admission

Comme nous l'avons évoqué précédemment, le contrôle d'admission permet d'éviter d'accepter des requêtes que l'on ne peut pas satisfaire dans l'état actuel du système. Sa décision permet d'accepter ou de rejeter les requêtes en amont. Si la réponse est positive, alors le système s'engage à satisfaire la requête en respectant ses contraintes avec une forte probabilité. Si la réponse est négative, la requête est définitivement rejetée et le processus applicatif doit reformuler une nouvelle requête avec des contraintes moins importantes et/ou attendre que la charge du système diminue.

Notre mécanisme de contrôle d'admission est entièrement distribué. Chaque processus a donc, en fonction de sa place dans l'arbre, un rôle dans ce mécanisme. Pour cela, nous nous servons de la propriété hiérarchique de la topologie. Ainsi, plus un processus se situe proche de la racine, plus sa connaissance potentielle sur les requêtes pendantes du système est grande. Par conséquent, le site racine est le seul à pouvoir connaître l'ensemble des requêtes pendantes du système et donc le seul à pouvoir prendre la décision finale d'accepter ou pas une requête. Cependant, chaque processus connaît un sous-ensemble des requêtes pendantes de son père. Il est alors possible de filtrer les messages de requêtes

et éviter à ce qu'une requête invalidée par un processus soit retransmise au père qui donnerait la même décision. Ainsi, à la réception ou à l'initiation d'une nouvelle requête, le processus va dans un premier temps décider en fonction des requêtes déjà présentes dans sa file locale si la requête est faisable ou pas. Si tel est le cas, la requête est alors retransmise au père. Si la décision locale est négative, alors la requête est rejetée définitivement. Ce mécanisme est récursif jusqu'au processus racine.

Un site qui envoie le jeton y inclus dans le message sa file locale. Ainsi le prochain détenteur du jeton pourra fusionner sa file locale avec celle du jeton lors de l'appel à **controlRecvTok**. Cette fusion permet d'assurer que le porteur du jeton connaîtra toujours l'ensemble des requêtes pendantes du système.

Décision locale

La file locale Q est ordonnée selon une politique EDF. À la création ou à la réception d'une requête req (**controlReqCS** ou **controlRecvReq**), le processus détermine la place potentielle i que req pourrait occuper dans Q. Il peut alors déterminer si req peut être satisfaite ou pas. Nous avons considéré qu'une requête req est faisable si :

(1) la requête à la place $i-1$ dans Q (notée req_{i-1}) doit respecter les contraintes de req après son insertion

(2) req doit respecter les contraintes de la requête à la position $i+1$ dans Q (notée req_{i+1}) qui a déjà été validée.

Pour respecter ces deux conditions, il est nécessaire de considérer le pire scénario où les requêtes validées seront satisfaites à leur date d'échéance. Ainsi pour assurer (1), la date d'échéance de la requête à la place $i-1$ plus le temps de sa section critique et le temps réseau pour acheminer le jeton entre le site initiateur de req_{i-1} au site initiateur de req ne doit pas dépasser la date d'échéance de req. Symétriquement, pour assurer (2), la date d'échéance de req plus son temps de section critique et le temps réseau pour acheminer le jeton entre le site initiateur de req au site initiateur de req_{i+1} ne doit pas dépasser la date d'échéance de req_{i+1}. Il est important de préciser que dans notre cas, ces conditions sont strictement appliquées (aucune marge de dépassement n'est prise en compte).

Le calcul de faisabilité implique l'ajout de deux hypothèses sur le système :

1. les horloges des différents sites sont synchronisées (présence d'un serveur NTP)
2. les sites connaissent le temps d'acheminement maximal d'un message vers un voisin noté γ_{max}.

La suppression de ces hypothèses sera une perspective de recherche de ce chapitre et sera discutée en chapitre 8.

Nous considérons que le calcul de la décision locale se fait par l'appel à la fonction **localDecision** qui retourne vrai si elle est positive, faux sinon.

Notification du résultat au site initiateur

Lorsqu'une requête est rejetée par un site, un message de rejet *Reject* est alors renvoyé vers le site initiateur. Un tel message est transmis jusqu'à ce dernier qui pourra finalement annuler sa requête. En revanche en cas d'acceptation de la requête, nous avons défini deux stratégies de notification.

```
66  Local variables :
67  begin
68      Q_tmp : queue of
        (s_i, s_n, t, α, d) ∈ Π × Π × T × ℕ × ℕ ;
69      permission ∈ {no, yes, empty};
70  end
71  Initialization
72  begin
73      Q_tmp ← ∅;
74      permission ← empty;
75  end
76  controlReqCS(t_dead ∈ T, α ∈ ℕ ) with boolean
    result
77  begin
78      if localDecision(t_dead, α) then
79          if father = nil then
80              add (self, self, t_dead, α, 0) in Q;
81              return true ;
82          else
83              add (self, self, t_dead, α, 0) in Q_tmp;
84              permission ← empty;
85              Send Request(self, t_dead, α, 1)) to father ;
86              wait(permission ≠ empty or
                t_now > t_dead);
87              if t_now > t_dead or permission = no then
88                  return false ;
89              else
90                  return true ;
91      else
92          return false;
93  end
94  controlRecvReq(s_i ∈ Π, s_n ∈ Π, t_dead ∈ T, α ∈ ℕ, d ∈
    ℕ)
95  begin
96      Q_tmp ← Q_tmp − {(s'_i, s'_n, t', α', d') ∈ Q | s'_i = s_i} ;
97      if localDecision(t_dead, α) then
98          if father = nil then
99              Validate(s_i, s_n, t_dead, α, d);
100         else
101             add (s_i, s_n, t_dead, α, d) in Q_tmp;
102             Send Request(s_i, t_dead, α, d + 1)) to
                father ;
103     else
104         Send Reject(s_i,t_dead) to s_n ;
105 end
106 controlRecvTok(Q_token, s_j ∈ Π)
107 begin
108     foreach (s_i, s_n, t_dead, α, d) ∈ Q_token do
109         if s_n ≠ self then
110             add (s_i, s_j, t_dead, α, d + 1) in Q;
111     foreach (s_i, s_n, t_dead, α, d) ∈ Q_tmp do
112         if ∄(s'_i, s'_n, t'_dead, α', d') ∈ Q | s'_i = s_i then
113             if localDecision(t_dead, α) then
114                 Validate(s_i, s_n, t_dead, α, d);
115             else
116                 Invalidate(s_i, s_n, t_dead);
117     Q_tmp ← ∅;
118 end
119 Receive_Reject(s_i ∈ Π, t_dead ∈ T) from s_j
120 begin
121     if ∃(s'_i, s'_n, t'_dead, α', d') ∈ Q_tmp, s_i = s'_i ∧ t_dead =
        t'_dead then
122         remove (s'_i, s'_n, t'_dead, α', d') from Q_tmp;
123         Invalidate(s'_i, s'_n, t'_dead);
124     else
125         /* ne rien faire, ceci concerne une
               requête obsolète                    */
126 end
127 Receive_Validation(s_i ∈ Π, t_dead ∈ T) from s_j
128 begin
129     if ∃(s'_i, s'_n, t'_dead, α', d') ∈ Q_tmp, s_i = s'_i ∧ t_dead =
        t'_dead then
130         remove (s'_i, s'_n, t'_dead, α', d') from Q_tmp;
131         Validate(s'_i, s'_n, t'_dead, α', d');
132     else
133         /* ne rien faire, ceci concerne une
               requête obsolète                    */
134 end
135 Validate(s_i ∈ Π, s_n ∈ Π, t_dead ∈ T, α ∈ ℕ, d ∈ ℕ)
136 begin
137     if t_dead ≥ t_now then
138         add (s_i, s_n, t_dead, α, d) in Q;
139         if s_i = self then
140             permission ← yes;
141             notify(permission ≠ empty)
142         else
143             Send Validation(s_i,t_dead) to s_n ;
144 end
145 Invalidate(s_i ∈ Π, s_n ∈ Π, t_dead ∈ T)
146 begin
147     if t_dead ≥ t_now then
148         if s_i = self then
149             permission ← no;
150             notify(permission ≠ empty)
151         else
152             Send Reject(s_i,t_dead) to s_n ;
153 end
```

FIGURE 5.2 – Contrôle d'admission avec acquittement

Stratégie avec acquittement : *Le processus initiateur doit attendre un message d'acquittement de son père qui confirme que sa requête a été acceptée par le système.* Ainsi lorsque la requête est localement acceptée, elle n'est pas immédiatement ajoutée dans la file locale Q mais dans une file temporaire Q_{tmp}. Le pseudo-code de cette version est donné figure 5.2. Une requête est ajoutée dans la file locale définitive seulement après la réception du message d'acquittement envoyé par le père. Puisque seul le site racine connaît toutes les requêtes du système, il est le seul à pouvoir initier le transfert du message d'acquittement. Ce message est alors retransmis le long des processus séparant la racine du processus initiateur. Une requête est effacée de la file locale Q du processus initiateur une fois que la section critique a été libérée ou bien si la date d'échéance de la requête a expiré.

Stratégie sans acquittement : *Le processus initiateur et les processus intermédiaires ajoutent directement la requête dans la file locale Q.* Par conséquent, ils n'attendent aucun message d'acquittement. Une requête est alors définitivement acceptée lorsque le porteur du jeton l'a reçue et validée. Les requêtes contenues dans la file du jeton sont les seules à être définitivement validées. À la réception du jeton, le processus doit donc contrôler localement toutes les requêtes présentes dans sa file locale et absentes de la file du jeton. Dans ce type de notification, une requête est effacée de la file locale Q du processus initiateur une fois que la section critique a été libérée ou bien à la réception d'un message de rejet ou bien si la date d'échéance de la requête a expiré. Le pseudo-code de cette version est donné figure 5.3.

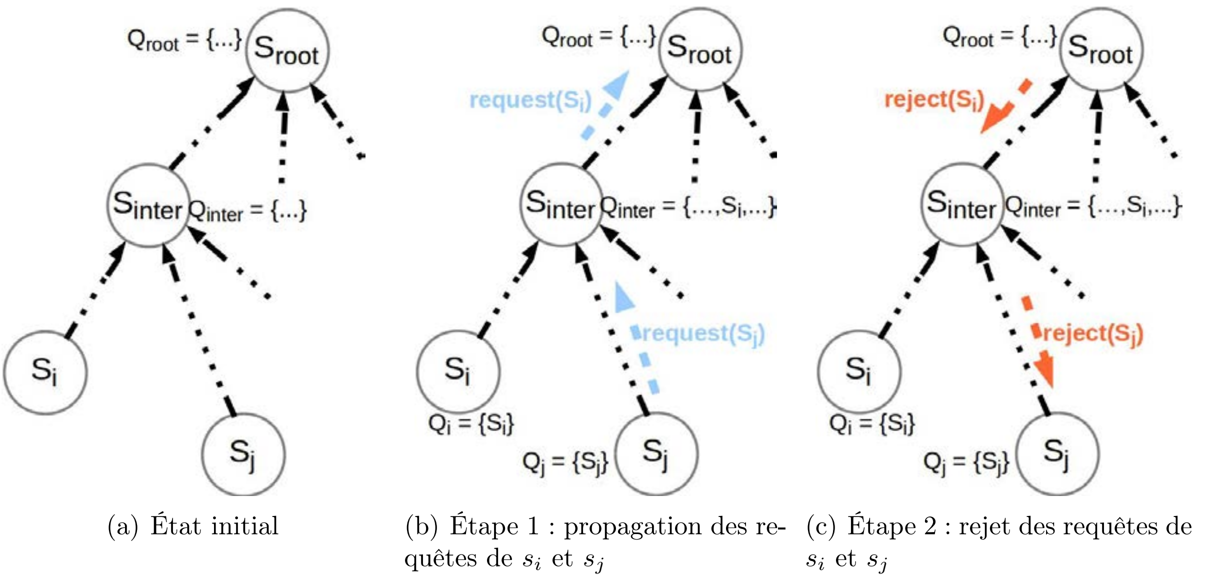
Comparaison entre les deux stratégies : la deuxième stratégie est plus économe en messages car elle n'utilise pas de message de validation. Cependant elle a une vision plus pessimiste et peut rejeter des requêtes qui pourraient être validées. Prenons par exemple la configuration de la figure 5.4(a) avec quatre sites distincts s_i, s_j, s_{inter} et s_{root} où s_{root} est la racine du système et s_{inter} est un ancêtre commun de s_i et s_j. Considérons que s_i a émis une requête req_i qui est validée par tous les processus entre s_i et s_{root} sauf par s_{root}. Considérons que s_j a émis une requête req_j qui elle peut être validée par s_{root}. Supposons que req_i soit ajoutée dans la file locale du site s_{inter} se trouvant entre s_i et s_{root} mais soit encore en transit entre s_{inter} et s_{root} (figure 5.4(b)). Si pendant ce temps s_{inter} reçoit req_j et l'invalide à cause des contraintes de req_i alors req_j (figure 5.4(c)) sera définitivement perdue dans la version sans acquittement alors qu'elle aurait pu être validée dans la version avec acquittement. Ainsi dans la version sans acquittement on refusera deux requêtes tandis que dans la version avec acquittement on ne refusera que req_i. Si ce cas de figure apparaît régulièrement, alors la version sans acquittement sera moins efficace en termes de taux d'utilisation de la ressource critique.

5.3.3 Mécanisme de préemption

Le mécanisme de préemption permet d'améliorer le taux d'utilisation de la ressource critique en prenant en compte la localité des requêtes. Contrairement à un ordonnancement classique d'exclusion mutuelle (FIFO ou à priorité), l'ordre dans lequel les requêtes

```
154 Local variables :
155 begin
156 |   permission ∈ {no, yes, empty};
157 end
158 Initialization
159 begin
160 |   permission ← empty;
161 end
162 controlRecvTok(Q_token, s_j ∈ Π)
163 begin
164     queue Q_old ← Q;
165     Q ← ∅;
166     foreach (s_i, s_n, t_dead, α, d) ∈ Q_token do
167         if s_n ≠ self then
168             add (s_i, s_j, t_dead, α, d + 1) in Q;
169         else
170             if ∃(s'_i, s'_n, t'_dead, α', d') ∈ Q_old, s'_i = s_i
                then
171                 add (s'_i, s'_n, t'_dead, α', d') in Q;
172     foreach (s_i, s_n, t_dead, α, d) ∈ Q_old do
173         if ∄(s'_i, s'_n, t'_dead, α', d) ∈ Q, s'_i = s_i then
174             if localDecision(t_dead, α) then
175                 add (s_i, s_n, t_dead, α, d) in Q;
176                 if s_i = self then
177                     permission ← yes;
178                     notify(permission ≠ empty);
179             else
180                 Invalidate(s_i, s_n, t_dead);
181 end
182 controlReqCS(t_dead ∈ T, α ∈ ℕ ) with boolean
    result
183 begin
184     /* identique à la version avec acquittement
           en remplaçant Qtmp par Q                */
185 end
186 controlRecvReq(s_i ∈ Π, s_n ∈ Π, t_dead ∈ T, α ∈
    ℕ, d ∈ ℕ)
187 begin
188     if localDecision(t_dead, α) then
189         add (s_i, s_n, t_dead, α, d) in Q;
190         if father ≠ nil then
191             Send Request(s_i, t_dead, α, d + 1)) to
                father ;
192     else
193         Send Reject(s_i, t_dead) to s_n ;
194 end
195 Receive_Reject(s_i ∈ Π, t_dead ∈ T) from s_j
196 begin
197     /* identique à la version avec acquittement
           en remplaçant Qtmp par Q                */
198 end
199 Invalidate(s_i ∈ Π, s_n ∈ Π, t_dead ∈ T)
200 begin
201     /* identique à la version avec acquittement
           */
202 end
```

FIGURE 5.3 – Contrôle d'admission sans acquittement



(a) État initial (b) Étape 1 : propagation des requêtes de s_i et s_j (c) Étape 2 : rejet des requêtes de s_i et s_j

FIGURE 5.4 – Exemple pour comparer les deux stratégies de validation

sont satisfaites importe peu à condition que l'ensemble des requêtes acceptées soient satisfaites avant leur date d'échéance. Il peut alors être possible entre deux sections critiques successives d'un ordonnancement EDF de satisfaire d'autres requêtes de nœuds se situant entre les deux processus concernés. Nous notons req_{head} la requête du site s_{head} qui se trouve en tête de file du jeton, i.e., la prochaine requête à satisfaire selon l'ordonnancement EDF. Pendant le trajet du jeton, chaque processus intermédiaire peut donc l'utiliser à la seule contrainte que le jeton doit arriver sur s_{head} avant sa date d'échéance.

Pour que des sites puissent utiliser le jeton pendant son trajet ils doivent respecter la **condition de préemption** : *la somme des durées de leur section critique ajoutée au temps de transmission du jeton ne doit pas dépasser la date d'échéance de* s_{head}. La condition de préemption est vérifiable en appelant la fonction **canPreempt** dont le pseudo-code est donné en figure 5.5.

```
203 canPreempt(req) with boolean result
204 begin
205     /* La fonction retourne vraie si la requête req peut préempter req_head, faux sinon  */
206     (s_i, s_n, t_dead, α, d) ← req;
207     (s_head, s_nhead, t_head, α_head, d_head) ← head(Q);
208     if s_n = s_nhead then
209         /* le site courant n'est pas un site appartenant au chemin (s_i, s_head), le jeton peut donc se
               diriger vers s_i sans risque de violer req_head                                     */
210         return true;
211     else
212         return (2 * d + d_head) * γ_max + α < t_head
213 end
```

FIGURE 5.5 – Fonction de condition de préemption

Nous avons défini trois politiques de préemption :

- **Pas de préemption** : Le jeton suit scrupuleusement l'ordonnancement EDF. Il n'est pas utilisé durant son trajet vers s_{head}. Lorsqu'un site reçoit le jeton, il le transfère directement au voisin correspondant à la requête en tête de Q_{token}.
- **Préemption simple** : Le jeton peut être utilisé uniquement par les sites se situant sur le chemin menant à s_{head}. À la réception du jeton, un site ayant une requête pendante peut entrer en section critique même s'il n'est pas le site s_{head} à condition qu'il satisfasse la contrainte de préemption.
- **Préemption étendue** : Le jeton peut être détourné du chemin menant à s_{head}. À la réception d'un jeton, si une préemption simple n'est pas possible le processus pourra contrôler s'il existe un voisin de distance 1 (pas nécessairement sur le chemin menant à s_{head}) qui respecte la condition de préemption. Si tel n'est pas le cas, il contrôlera pour des sites de distance 2 et ainsi de suite jusqu'à une distance maximale paramétrable. Cette distance notée ψ est appelée **taille de préemption maximum** et est bornée par le diamètre de l'arbre.

Ces trois politiques sont illustrées en figure 5.6. À chaque politique, nous donnons un exemple et le pseudo-code de la fonction **preempt**. Dans les exemples, les flèches rouges indiquent une transmission de jeton. Un nœud rouge indique que le processus correspondant est entré en section critique lorsqu'il a reçu le jeton.

Nous pouvons remarquer que la préemption étendue est une généralité des deux autres

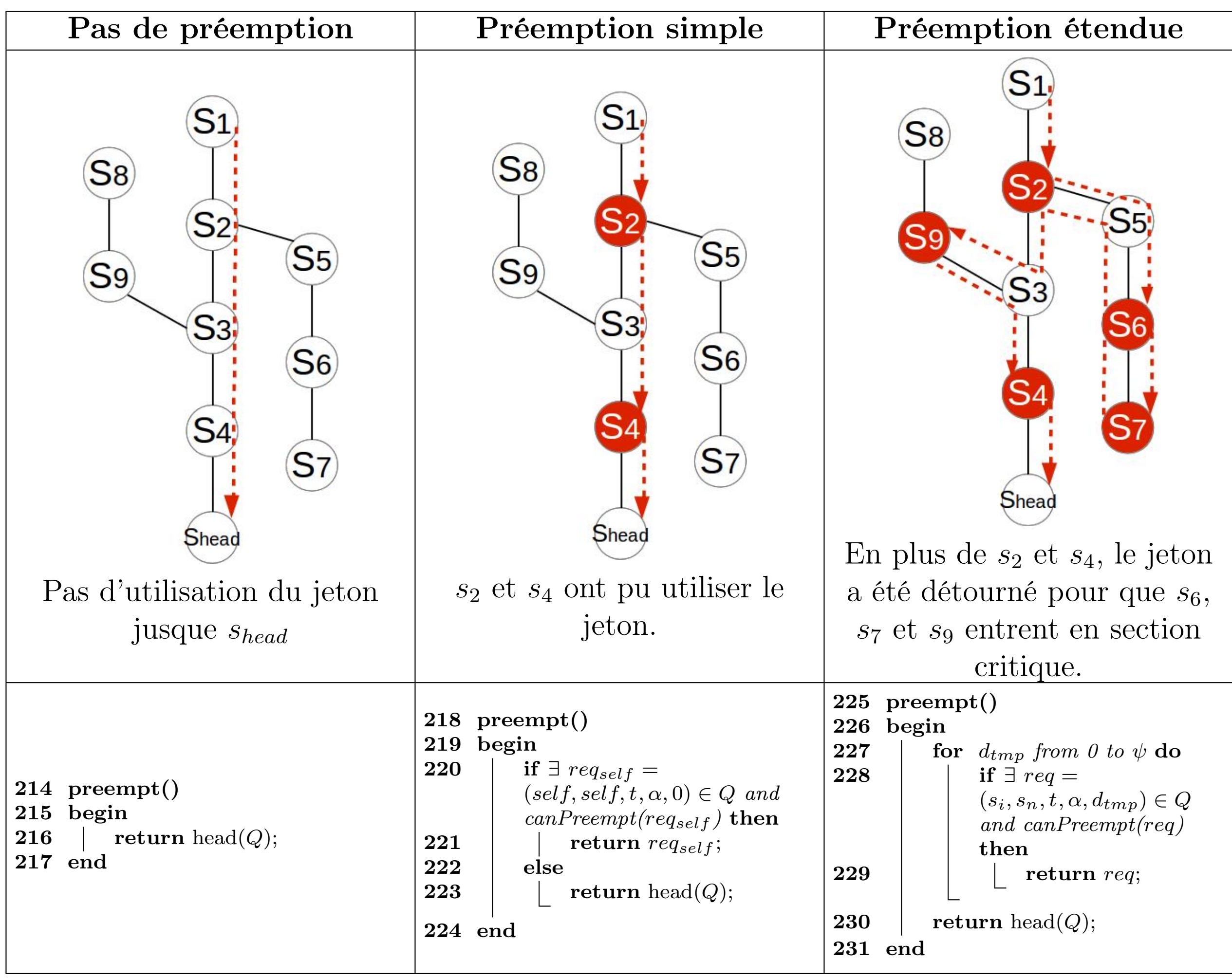


FIGURE 5.6 – Description des différentes politiques de préemption

types de préemption : pour la version sans préemption $\psi < 0$ et pour la version simple $\psi = 0$.

5.4 Évaluation des performances

Dans cette section, nous allons analyser les performances de notre algorithme. Dans notre protocole, nous prédéfinissons plusieurs niveaux de SLA où chaque niveaux correspond à un temps de réponse requis. Plus ce niveau est grand, plus les contraintes temporelles sont grandes et par conséquent plus le temps de réponse associé est petit. Chaque processus choisira donc un niveau de SLA avant d'émettre une nouvelle requête.

Comme à notre connaissance il n'existe pas dans la littérature d'autres algorithmes d'exclusion mutuelle à passage de messages se basant sur l'exclusion mutuelle à date d'échéance, nous comparons les deux versions de notre algorithme ("avec acquittement" et "sans acquittement") avec l'algorithme de Raymond (voir section 2.5.1) et notre algorithme à priorités "Awareness" associé à une fonction de retard $\mathcal{F}(p) = 2^p$ (voir section 4.5.3). Nous pouvons ainsi comparer le comportement des ordonnancements FIFO et à priorités avec notre algorithme qui applique une stratégie EDF. Ces deux algorithmes étant dépourvus de contrôle d'admission, toute requête émise est supposée satisfiable. Ces deux algorithmes ont été légèrement adapté : un site peut détecter que sa date d'échéance a été dépassée et dans ce cas annuler sa requête en l'effaçant de sa propre file locale. De plus, pour l'algorithme "Awareness", nous avons fait une correspondance directe entre priorité et niveau de SLA : plus la priorité est élevée, plus le niveau de SLA est haut.

5.4.1 Protocole d'évaluation

Les expériences ont été réalisées sur une grappe de 32 machines de Grid 5000 avec un processus par machine. Ceci permet d'éviter les effets de bord de contention au niveau des cartes réseau puisqu'il y a un seul processus par carte. Chaque nœud a deux processeurs AMD 1.7GHz et 47 Go de mémoire RAM, s'exécute sur un noyau Linux 2.6 (cluster Grid5000 Reims). Les nœuds sont reliés par un switch de 1 Gbit/s en ethernet. Les algorithmes ont été implémentés en C++ en utilisant l'intergiciel OpenMPI.

Une expérience se caractérise par

- N : le nombre de processus fixé à 32.
- α : le temps d'une section critique. Cette valeur est comprise entre 25 ms et 50 ms. Bien que la plate-forme de test possède un mécanisme NTP de synchronisation des nœuds, il peut persister un décalage d'horloge. C'est pour cela que les valeurs de α ont été choisies volontairement hautes afin de rendre ce décalage négligeable dans le calcul du contrôle d'admission.
- γ : la latence réseau,i.e., le temps d'acheminement d'un message entre deux processus voisins fixée à 20 ms. Nous avons choisi de rendre le temps réseau non négligeable par rapport au temps de section critique car dans un Cloud, les nœuds physiques peuvent être séparés par une grande distance.
- θ : le temps de l'expérience est fixée à 35 secondes. Cette valeur est très élevée par rapport au temps de section critique et permet de réaliser un nombre important de

section critique par expérience.
- ρ : la charge du système (voir section 4.4.1 pour plus de détails sur ce paramètre)
- $NbSLA$: Le nombre de niveaux de SLA considéré fixé à 4. Un identifiant de SLA est compris entre 0 et $NbSLA - 1$. Plus cet identifiant est haut et plus le niveau de contrainte est grand.
- $WaitMin$: le temps de réponse requis minimal qui correspond au plus haut niveau de SLA (identifiant $NbSLA - 1$). Ce paramètre est fixé à 600 ms.
- ψ : la taille de préemption (voir section 5.3.3)

À chaque nouvelle requête, un site choisit selon une loi aléatoire uniforme une valeur α de temps de section critique et un identifiant de SLA noté sla compris entre 0 et $NbSLA-1$. Un site peut alors déterminer le temps d'attente maximum requis à la requête par $WaitMin * (NbSLA - sla)$. La date d'échéance associée est donc la date à laquelle la requête a été émise en y ajoutant le temps maximum d'attente requis.

5.4.2 Métriques

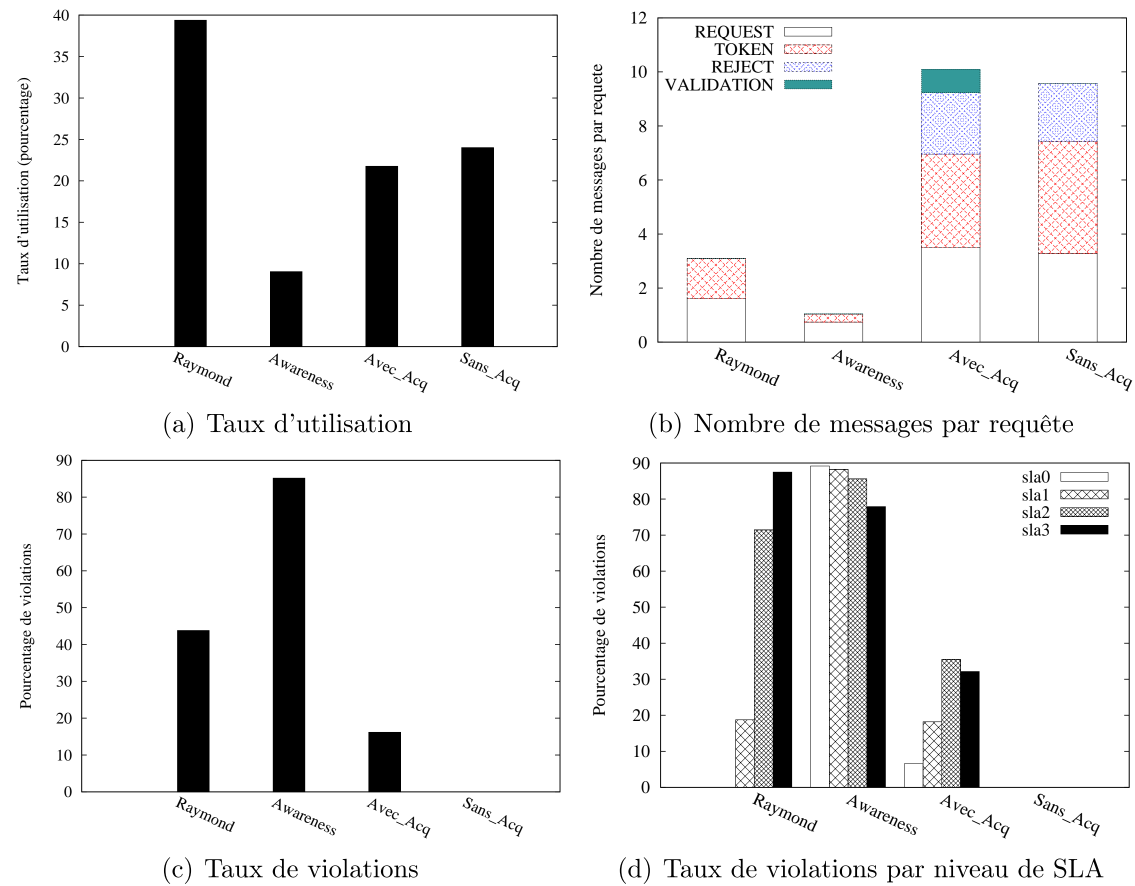
Nous considérons les métriques habituelles de l'exclusion mutuelle :
- **le nombre de messages moyen par requête** : pour un type de message donné, cette métrique se calcule en faisant le ratio entre le nombre total de messages de ce type émis normalisé par le nombre total de requêtes à l'exception du type $TOKEN$ qui est normalisé par le nombre total de requêtes acceptées par le contrôle d'admission (les requêtes rejetées n'ont pas lieu de recevoir le jeton).
- le **taux d'utilisation** : le pourcentage de temps passé en section critique.

En plus de ces métriques propres à l'exclusion mutuelle classique, nous ajoutons une métrique supplémentaire propre à l'exclusion mutuelle à dates d'échéances qui est **le taux de violations de SLA** qui est le pourcentage de requêtes acceptées n'ayant pas été satisfaites avant leur date d'échéance. Nous ne traiterons pas le temps de réponse car il est compris entre 0 et le temps d'attente maximum requis.

5.4.3 Impact global

Dans cette section nous considérons une charge constante avec $\rho = 0.5N$ qui correspond à une charge intermédiaire (environ 50% des sites en attente). Le mécanisme de préemption est désactivé sur nos algorithmes ($\psi = -1$) afin de pouvoir analyser uniquement l'impact du contrôle d'admission. Les résultats sont donnés en figure 5.7.

Algorithme de Raymond : L'algorithme de Raymond exploite implicitement la localité des requêtes lorsque la latence réseau n'est pas négligeable. En effet, les sites éloignés du porteur du jeton sont naturellement pénalisés par l'ordonnancement FIFO : les messages de requêtes ont peu de chances d'arriver jusqu'au porteur du jeton (grâce au mécanisme de non-retransmission). De ce fait, les trajets du jeton entre deux sections critiques seront moins importants mais les sites éloignés du jeton ont plus de chances d'annuler leur requête lors du dépassement de leur date d'échéance. Ce phénomène se confirme sur la figure 5.7(a) où l'on peut remarquer que le nombre de messages de jeton par requête est beaucoup plus important pour nos algorithmes que pour l'algorithme de Raymond.



(a) Taux d'utilisation

(b) Nombre de messages par requête

(c) Taux de violations

(d) Taux de violations par niveau de SLA

FIGURE 5.7 – Comparaison globale

Bien que son taux d'utilisation soit supérieur à nos algorithmes (figure 5.7(c)), 40 % des requêtes acceptées ne sont pas satisfaites dans les temps impartis (figure 5.7(c)).

Algorithme à priorités : L'ordonnancement à priorité de l'algorithme "Awareness" privilégie les requêtes de niveau 3 proche du porteur du jeton dans le graphe (grâce au mécanisme de distance, voir section 4.3.4) ce qui explique son faible nombre de messages de jeton (figure 5.7(b)). Mais la correspondance priorités-niveaux de SLA n'est pas une approche satisfaisante car on remarque dans la figure 5.7(c) que 86 % des requêtes ne sont pas satisfaites dans les temps impartis.

Algorithmes avec et sans acquittement : On peut remarquer sur la figure 5.7(a) que nos deux algorithmes ont un taux d'utilisation plus faible que celui de Raymond. Cette différence est due aux politiques d'ordonnancement. L'ordonnancement EDF de nos algorithmes sans préemption ne tient pas compte de la localité des requêtes impliquant potentiellement de longs trajets du jeton. Il est notable sur les figures 5.7(c) et 5.7(d) que le contrôle d'admission réduit le nombre de violations de SLA au prix d'un surcoût en messages (figure 5.7(b)) à cause des mécanismes de notification qui envoient des messages de validation ou de rejet. De plus les messages de requêtes sont plus nombreux car ils sont

systématiquement retransmis au processus père tant que la décision locale est positive. La version avec acquittement génère cependant des violations de SLA alors que l'on observe aucune violation dans la version sans acquittement. Ceci s'explique par le fait que la version avec acquittement ne tient pas compte de son propre surcoût en temps dans son évaluation. En effet le temps de contrôle n'est pas négligeable puisque le site doit systématiquement attendre la notification de son père pour être accepté définitivement. En revanche, dans la version sans acquittement, les requêtes satisfiables sont validées systématiquement impliquant un contrôle d'admission gratuit en temps.

Synthèse

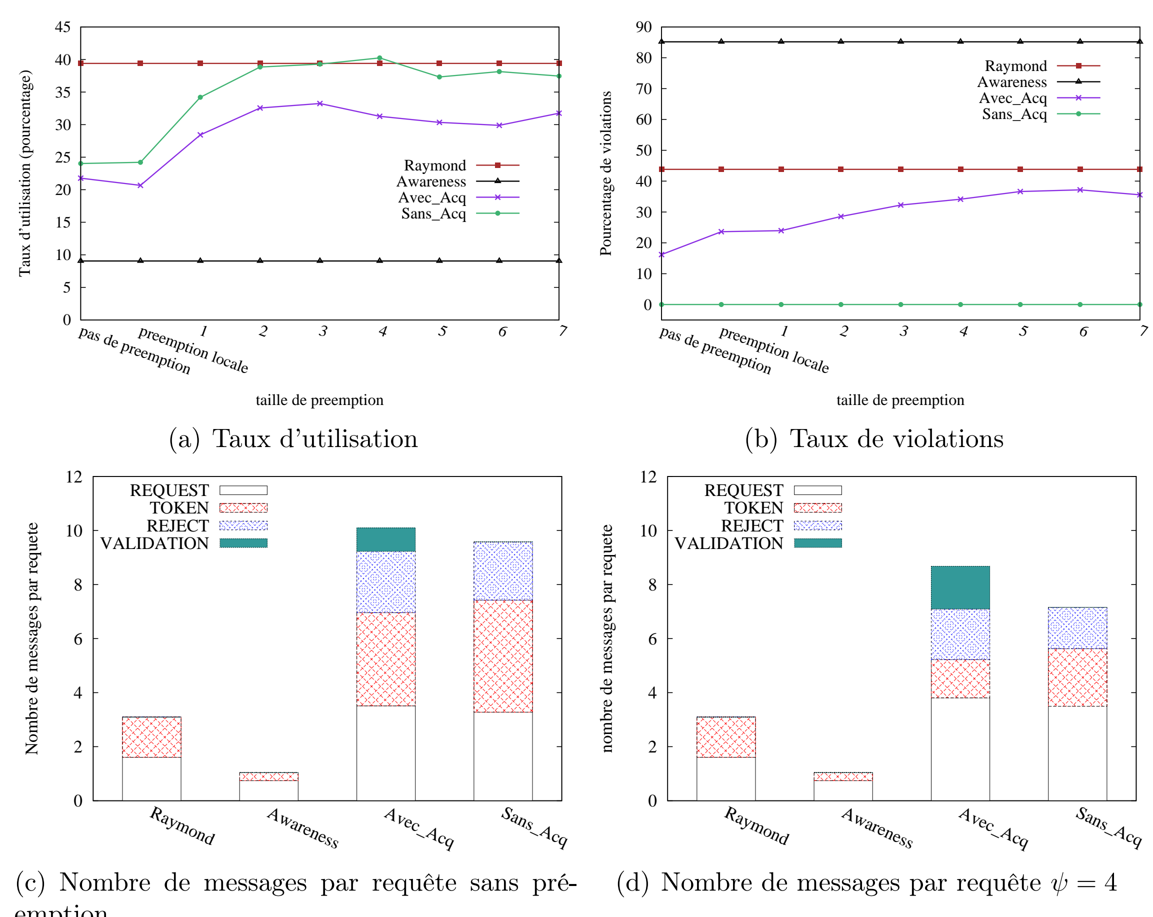
Notre mécanisme de contrôle d'admission associé à l'ordonnancement EDF permet de réduire fortement la quantité de requêtes n'ayant pas été satisfaites à temps. Nous pouvons remarquer que le gain théorique de taux d'utilisation de notre algorithme avec acquittement sur l'algorithme sans acquittement n'est pas mis en évidence dans nos graphiques. De plus, le taux d'utilisation de nos algorithmes (25 %) reste inférieur à l'algorithme de Raymond (40 %) qui favorise la localité des requêtes.

5.4.4 Impact de la préemption pour une charge donnée

Nous avons remarqué que sans préemption le taux d'utilisation de la ressource critique est réduit. Nous analysons maintenant le comportement de nos algorithmes lorsque l'on augmente le paramètre de taille de préemption ψ (figure 5.8). Nous comparons le nombre de messages quand il n'y a pas de préemption (figure 5.8(c)) et une valeur intermédiaire de ψ (figure 5.8(d)). Cette valeur intermédiaire (égale à 4) est la taille de préemption où nous pouvons avoir les meilleures performances en termes de taux d'utilisation (voir figure 5.8(a)).

Impact sur le taux d'utilisation : Nous pouvons observer dans la figure 5.8(a) que l'algorithme de Raymond est plus efficace que nos algorithmes quand ces derniers sont privés du mécanisme de préemption. Lorsque le niveau de préemption augmente, le taux d'utilisation augmente puis se dégrade légèrement après une valeur seuil (3 pour la version avec acquittement et 4 pour la version sans acquittement). En effet quand la taille de préemption augmente, l'importance de la déviation du jeton de son chemin initial a plus de chances d'augmenter. Lorsque la latence réseau n'est pas négligeable, une longue déviation est trop coûteuse car elle empêche l'utilisation du jeton sur le chemin initial vers s_{head}.

Impact sur le taux de violations : On remarque sur la figure 5.8(b) que la taille de préemption n'a aucun impact sur la version sans acquittement : le taux de violation est toujours nul. En revanche, dans la version avec acquittement la quantité de violation augmente avec la taille de préemption. Le contrôle d'admission est d'autant plus coûteux en temps car le temps de validation augmente. En effet on peut remarquer dans les figures 5.8(c) et 5.8(d) que lorsque la taille de préemption augmente le nombre de messages de validation par requête augmente également.



(a) Taux d'utilisation

(b) Taux de violations

(c) Nombre de messages par requête sans préemption

(d) Nombre de messages par requête $\psi = 4$

FIGURE 5.8 – Impact de la taille de préemption ψ

Impact sur le nombre de messages : On note dans les figures 5.8(c) et 5.8(d) que l'on peut économiser des messages de jeton par requête satisfaite lorsque la taille de préemption augmente. En effet, grâce au mécanisme de préemption, le jeton satisfait davantage de requêtes pendant son trajet rentabilisant ainsi son utilisation. Ceci est cohérent avec la figure 5.8(a).

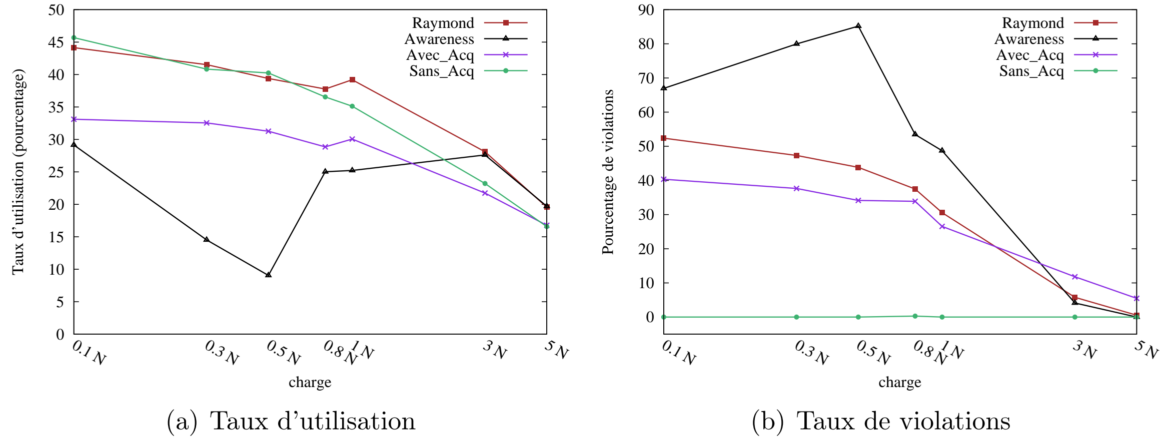
De manière globale, la figure 5.8 montre que notre algorithme sans acquittement égale le taux d'utilisation de l'algorithme de Raymond pour une valeur seuil de taille de préemption tout en gardant un taux de violation nul.

Synthèse

Nous remarquons que le gain théorique de l'algorithme avec acquittement n'est toujours pas mis en évidence. Lorsque l'on active le mécanisme de préemption, nous pouvons obtenir avec notre algorithme sans acquittement un taux d'utilisation équivalent à celui de Raymond tout en gardant un taux de violation faible. Cette amélioration se fait avec une petite valeur du paramètre ψ (2 ou 3).

5.4.5 Impact de la charge

Dans la figure 5.9, nous montrons l'impact de la valeur de la charge sur le taux d'utilisation et le taux de violation. Chaque point est évalué à partir d'une charge donnée ρ statique que l'on a fait varier de $5N$ (faible charge) à $0.1N$ (haute charge).



(a) Taux d'utilisation (b) Taux de violations

FIGURE 5.9 – Impact de la charge pour $\psi = 4$

La figure 5.9(a) montre que notre algorithme sans acquittement suit le même comportement que l'algorithme de Raymond en termes de taux d'utilisation : en faible charge, notre algorithme est légèrement moins performant que l'algorithme de Raymond alors qu'en moyenne et en forte charge, le taux d'utilisation de notre algorithme est au moins équivalent à l'algorithme de Raymond. Cependant le taux de violation de notre algorithme sans acquittement reste toujours nul quelle que soit la valeur de la charge. On peut donc en déduire que notre algorithme sans acquittement possède des performances équivalentes à l'algorithme de Raymond en termes de taux d'utilisation mais sans générer de violation de SLA grâce à son contrôle d'admission et son mécanisme de préemption.

5.5 Conclusion

Dans ce chapitre, nous avons présenté une nouvelle extension de l'exclusion mutuelle permettant de satisfaire des requêtes avant une date d'échéance requise.

Nous avons conçu un nouvel algorithme pour ce type d'exclusion mutuelle. Cet algorithme se base sur l'algorithme de Raymond et possède deux mécanismes principaux : un contrôle d'admission et un mécanisme de préemption. Le contrôle d'admission permet de minimiser les violations de SLA et le mécanisme de préemption permet de prendre en compte la topologie du système afin d'utiliser au mieux le jeton.

Les performances ont montré que le contrôle d'admission associé à un ordonnancement EDF permettait de limiter la quantité de requêtes qui n'ont pas été satisfaites à temps. Pour améliorer le taux d'utilisation, nous avons introduit un mécanisme de préemption qui permet de prendre en compte la localité des requêtes sans induire de violation. Ce mé-

canisme permet de rentabiliser l'utilisation du jeton en satisfaisant des requêtes pendantes de processus se trouvant sur son trajet.

Il serait cependant possible d'améliorer davantage le taux d'utilisation en ajoutant une fourchette de tolérance dans la décision locale du contrôle d'admission mais il y aurait alors un risque d'avoir une augmentation du nombre de violations de SLA. La limite de notre approche réside dans le fait que le temps maximal d'acheminement réseau entre deux voisins (paramètre γ_{max}) est supposé connu ce qui apporte une hypothèse relativement forte.

Troisième partie

Contribution dans la généralisation de l'exclusion mutuelle

Chapitre 6

Présentation de l'exclusion mutuelle généralisée

Sommaire

6.1 Introduction **91**
6.2 Généralités et notations **92**
6.3 Le modèle à une ressource en plusieurs exemplaires . . . **92**
6.3.1 Section critique à entrées multiples ou k-mutex 92
6.3.2 Plusieurs exemplaires par demande 93
6.4 Le modèle à plusieurs ressources en un seul exemplaire . **94**
6.4.1 Les conflits 94
6.4.2 Propriétés à respecter 95
6.4.3 Algorithmes incrémentaux. 95
6.4.4 Algorithmes simultanés 96
6.5 Le modèle à plusieurs ressources en plusieurs exemplaires **98**
6.6 Conclusion et synthèse **99**

6.1 Introduction

Dans les chapitres précédents, nous avons considéré le problème de l'exclusion mutuelle simple. Cependant, il existe plusieurs généralisations de ce problème. La généralisation peut se faire soit en augmentant le nombre d'exemplaires de la ressource soit en en considérant plusieurs types de ressource ou bien les deux.

Ce chapitre présente un état de l'art sur la généralisation de l'exclusion mutuelle afin d'introduire notre contribution décrite au chapitre suivant. Dans la section 6.2 nous donnons un aperçu des différents modèles. Dans les sections 6.3, 6.4 et 6.5 nous présenterons pour chaque modèle, les différents algorithmes existants dans la littérature qui sont bien souvent une extension d'un des algorithmes d'exclusion mutuelle classique décrits en section 2.4. En section 6.6, le lecteur pourra trouver dans la conclusion un schéma récapitulatif des différents modèles.

6.2 Généralités et notations

Il est possible de généraliser l'exclusion mutuelle classique selon deux critères [Ray92] :

- le nombre d'exemplaires de la ressource critique : la ressource existe en k exemplaires.
- le nombre de types de ressource : le système possède non pas une seule ressource mais un ensemble $\mathcal{R}= \{r_1, r_2, ...r_M\}$ de M types de ressources

Les deux critères étant orthogonaux, il est possible de définir un troisième modèle ultime de généralisation en faisant l'union des deux précédents. On peut donc définir un multi-ensemble de ressources noté $(\mathcal{R}, \kappa)$, où κ défini pour chaque élément de $\mathcal{R}$ un nombre d'exemplaires existants.

Nous notons $D^t_{s_i} \subseteq \mathcal{R}$ l'ensemble de ressources demandées par le site $s_i \in \Pi$ à l'instant $t \in T$ et $\delta^t_{s_i}$ la fonction définissant à l'instant t le nombre d'exemplaires demandés par s_i pour chaque ressource $r \in D^t_{s_i}$. L'ensemble de ressources demandées à l'instant t par le site s_i est donc le multi-ensemble noté $(D^t_{s_i}, \delta^t_{s_i})$ que l'on notera $(D, \delta)^t_{s_i}$.

La généralisation implique que désormais tout processus s_i devra préciser lors de l'appel à la primitive *Request_CS* à l'instant t, le multi-ensemble $(D, \delta)^t_{s_i}$ associé à sa requête pour entrer en section critique. Cette primitive sera bloquante jusqu'à l'acquisition de l'ensemble des exemplaires demandés de chaque type de ressource. Lors de l'appel à la primitive *Release_CS*, les exemplaires obtenus seront libérés.

6.3 Le modèle à une ressource en plusieurs exemplaires

Dans ce modèle $M = 1$ et $\kappa = k$ où $k \geq 1$. Il y a donc au plus k processus en section critique à un instant t.

Il est possible de sous-diviser ce modèle en fonction de $(D, \delta)^t_{s_i}$.

6.3.1 Section critique à entrées multiples ou k-mutex

Dans ce modèle de demande, les processus ne peuvent demander qu'un seul exemplaire de la ressource ($\delta^t_{s_i} = 1, \forall t \in T$). Ceci est similaire à un sémaphore. Il existe de nombreux algorithmes qui peuvent être classés de la manière suivante :

- **Permissions** : les processus attendent un message de permission de chaque processus du système pour entrer en section critique. On peut citer l'algorithme de Raymond [Ray89a] qui s'inspire de l'algorithme de Ricart-Agrawala [RA81]. Pour entrer en section critique, un site envoie $N - 1$ messages de requête et attend $N - k$ messages de permission.
- **Jetons** : Il existe k jetons dans le système. Les processus attendent l'acquisition d'un des jetons pour entrer en section critique. Comme dans l'exclusion mutuelle classique, l'obtention d'un jeton peut reposer sur une diffusion ou sur l'exploitation d'une structure logique :
  - ∗ diffusion : Srimani-Reddy [SR92] se sont basés sur l'algorithme de Suzuki-Kasami [SK85]. Baldoni-Ciciani [BC94] ont étendu [SR92] aux requêtes à priorités.
  - ∗ arbre statique : basé sur la circulation de k jetons dans la topologie, chaque site maintient k liens qui indiquent la direction de la racine qui est soit le détenteur

d'un jeton (DeMent-Srimani [DS94] et Satyanarayanan-Muthukrishnan [SM94]) soit le dernier demandeur (Naimi [Nai93]). À chaque nouvelle requête les sites envoient une requête à chacun de leurs pères et se mettent en attente d'un des jetons. Les requêtes sont ainsi propagées jusqu'aux racines. Le routage des jetons se fait toujours à la manière de l'algorithme classique de Raymond [Ray89b]. Afin d'éviter la réception multiple de jetons venant des différents liens, des mécanismes d'annulation de requêtes peuvent être déclenchés ([DS94]) lorsque le site demandeur entre en section critique.

∗ arbre dynamique : Dans ce genre de topologie, les différents liens évoluent au cours de l'exécution de l'algorithme contrairement à l'arbre statique où seule la direction des liens peut s'inverser. Dans cette catégorie nous pouvons citer [MBB+92] qui utilise un unique jeton contenant une file globale des requêtes pendantes, alors que [BV95] et [RMG08] se basent sur l'algorithme de Naimi-Tréhel [NT87a] et utilisent k jetons où l'accès à chaque jeton est géré par une file distribuée.

- **Hiérarchique centralisé** : Dans cette catégorie, les processus sont partitionnés en plusieurs groupes. Chaque processus ne peut communiquer qu'avec un processus du même groupe. Dans chaque groupe il existe un processus mandataire permettant de communiquer avec les autres mandataires et de facto avec le reste du système. Une telle répartition permet de répartir les processus en fonction de leur localité physique. Nous pouvons citer l'algorithme de Chaudhuri-Edward [CE08] qui répartit les processus en $\sqrt{N}$ groupes de $\sqrt{N}$ processus. Chaque groupe est associé à un processus coordinateur qui fait l'intermédiaire entre les deux niveaux de hiérarchie.

6.3.2 Plusieurs exemplaires par demande

Désormais les processus peuvent demander au plus k exemplaires de la ressource (formellement $\delta_{s_i}^t \geq 1, \forall t \in T$). Dans ce sous-modèle on ne trouve que des algorithmes à permissions :

- L' **algorithme de Raynal** [Ray91a] se base sur l'algorithme de Ricart-Agrawala [RA81] qui ordonnance les requêtes selon une horloge logique de Lamport [Lam78]. Chaque site maintient un vecteur de N compteurs où chaque entrée e indique du point de vue du site courant le nombre d'exemplaires possédés par le site s_e. Chaque nouvelle requête induit une diffusion de messages de requête contenant le nombre d'exemplaire voulu et une mise à jour pessimiste du vecteur en incrémentant de k chaque compteur. À la réception d'un message de requête, un site s_j envoie une réponse contenant le nombre d'exemplaires non utilisés (k si la requête de s_j est moins prioritaire, $k - \delta_{s_j}^t$ sinon). À chaque réception de réponse un site décrémente du nombre reçu le compteur correspondant et peut entrer en section critique lorsque la somme des compteurs du vecteur et de $\delta_{s_i}^t$ est inférieure à k.
- L'**algorithme de Baldoni** [Bal94] est une extension de l'algorithme à permissions d'arbitres de Maekawa [Mae85] se basant sur des quorums. En construisant judicieusement les quorums, sa complexité en messages est de $\mathcal{O}(N^{k/(k+1)})$ (pour $k = 1$, on retrouve la complexité de l'algorithme de Maekawa en $\mathcal{O}(\sqrt{N})$).

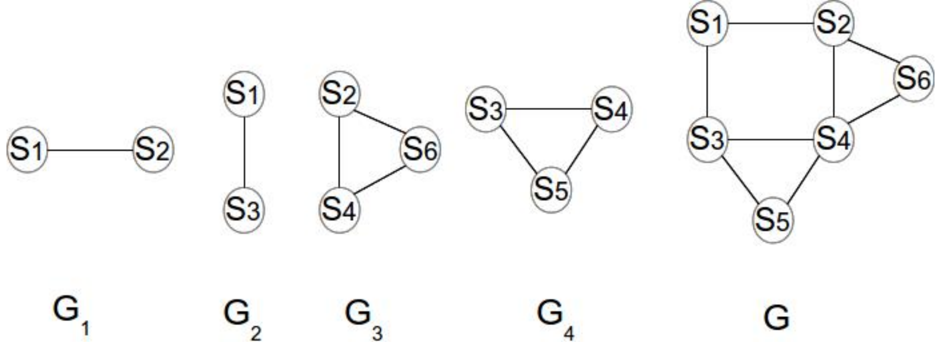


FIGURE 6.1 – Exemple de construction de graphe de conflit

6.4 Le modèle à plusieurs ressources en un seul exemplaire

Dans ce modèle $M \geq 1$ et $\kappa = 1$ impliquant que le multi-ensemble de ressources $(\mathcal{R}, \kappa)$ se réduit à l'ensemble $\mathcal{R}$. Par conséquent, le multi-ensemble de demande de ressources $(D^t_{s_i}, \delta^t_{s_i})$ sera réduit à l'ensemble $D^t_{s_i}$. Ce problème multi-ressource, aussi appelé "AND-synchronisation" a été introduit par Dijkstra [Dij71] avec le problème du "Dîner des philosophes" où les processus demandent un ensemble de ressources statiques ($\forall\, t, t' \in T$, $D^t_{s_i} = D^{t'}_{s_i}$). Ce problème a été étendu par Chandy et Misra [CM84] sous le nom du problème du "cocktail des philosophes" où les processus peuvent demander à chaque nouvelle requête un ensemble de ressources différent.

6.4.1 Les conflits

Dans le cadre de ce modèle, les requêtes peuvent être conflictuelles si leurs ensembles de ressources ne sont pas disjoints. Autrement dit, un conflit existe entre deux processus s_i et s_j si $D^t_{s_i} \cap D^t_{s_j} \neq \emptyset$. Il est possible alors de définir un graphe de conflits noté G où les nœuds correspondent aux processus du système et un lien modélise le partage d'une ressource entre deux nœuds. La construction d'un tel graphe se fait par l'union des graphes de conflits complets G_i associé à chaque ressource $r_i \in \mathcal{R}$. La figure 6.1 montre la construction d'un graphe de conflits à partir de 4 ressources et de 6 processus. Dans le graphe de conflits G, chaque arrête représente désormais une ressource virtuelle qui est partagée par exactement deux processus. Dans le problème du dîner des philosophes, pour qu'un site s_k verrouille la ressource réelle $r_i \in \mathcal{R}$, il lui faut donc verrouiller toutes les ressources virtuelles correspondantes aux arrêtes (s_k, s'_k) du graphe G_i. Dans notre exemple, si s_5 souhaite verrouiller la ressource r_4, il a besoin des ressources virtuelles (s_5,s_3) et (s_5,s_4). Un graphe de conflits peut s'appliquer à plusieurs ensembles de ressources $\mathcal{R}$ de départ : dans notre exemple, pour un graphe de conflits équivalent, on aurait pu aussi considérer un ensemble de ressources réelles égal à l'ensemble des arrêtes du graphe G.

6.4.2 Propriétés à respecter

Sûreté. Contrairement à l'exclusion mutuelle classique, la propriété de sûreté doit être modifiée afin d'autoriser que deux processus non-conflictuels puissent être tous les deux à un instant t à l'état $inCS$. Par conséquent, à un instant donné, une ressource peut être utilisée par au plus un processus. Formellement : $\forall t \in T$,

$$\exists s_i \in \Pi, inCS^t_{s_i} \Rightarrow \nexists s_j \in \Pi, inCS^t_{s_j} \wedge D^t_{s_i} \cap D^t_{s_j} \neq \emptyset$$

Vivacité. La définition de la propriété de vivacité reste inchangée et permet d'éviter la famine. Cependant, pour qu'elle soit respectée, il faut que deux sous-propriétés soient vérifiées :

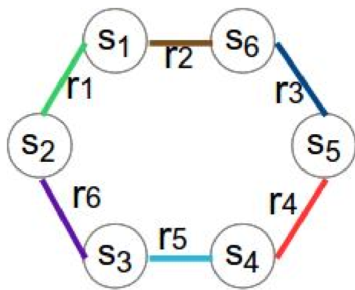
- **La distinguabilité :** cette propriété assure que si deux requêtes sont conflictuelles alors il est toujours possible de déterminer dans un temps fini un ordre de satisfaction entre ces deux requêtes. Assurer un accès exclusif à chaque ressource du système n'est pas suffisant pour garantir cette propriété car les accès aux ressources seraient gérés de manière indépendante les uns des autres sans prendre en compte l'ensemble $D^t_{s_i}$ qui les lie. Par conséquent, le non-respect de cette propriété peut conduire à un état d'**interblocage** dans lequel aucune progression n'est possible. Dans notre modèle, un interblocage peut se produire par exemple lorsque deux processus sont chacun en train d'attendre la libération d'une ressource verrouillée par l'autre.
- **l'équité :** permet d'assurer que dans un temps fini, toute requête émise sera satisfaite (propriété de base pour la vivacité).

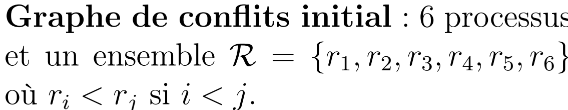
Concurrence. Cette propriété assure que deux processus ayant des requêtes non conflictuelles peuvent exécuter leurs sections critiques simultanément. Cette propriété exclut ainsi toute solution qui utiliserait un algorithme d'exclusion mutuelle classique où l'ensemble des ressources $\mathcal{R}$ représenterait une seule ressource.

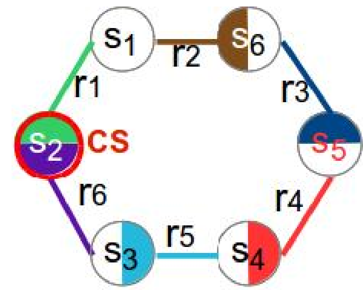
6.4.3 Algorithmes incrémentaux.

Dans cette famille d'algorithme chaque processus verrouille de manière incrémentale ses ressources suivant un ordre préalablement défini sur l'ensemble des ressources du système $\mathcal{R}$. Chaque verrou peut être implémenté avec un algorithme d'exclusion mutuelle classique. Cependant, une telle stratégie peut être inefficace puisqu'un effet domino des attentes peut se produire : un processus attend des ressources qui ne sont pas en cours d'utilisation mais qui sont verrouillées par des processus qui attendent l'acquisition d'autres ressources. L'effet domino dégrade la propriété de concurrence et par conséquent réduit fortement le taux d'utilisation des ressources. La figure 6.2 illustre cet effet dans un graphe de conflits en anneau à 7 processus.

Pour éviter l'effet domino, Lynch [Lyn81] propose de construire un graphe dual au graphe de conflit : les nœuds sont les ressources et il existe un lien entre deux nœuds si les deux ressources correspondantes sont susceptibles d'être demandées au sein d'une même requête. En coloriant ce graphe et en minimisant le nombre de couleurs, il est possible




Graphe de conflits initial : 6 processus et un ensemble $\mathcal{R} = \{r_1, r_2, r_3, r_4, r_5, r_6\}$ où $r_i < r_j$ si $i < j$.



s_2 est en section critique avec r_6 et r_1. Les processus s_3 et s_5 verrouillent respectivement r_5 et r_3 et attendent respectivement r_6, et r_4. Les processus s_4 et s_6 verrouillent r_4, r_2 et attendent respectivement r_5 et r_3 à cause de s_3 et s_5 alors qu'ils auraient pu entrer en section critique en même temps que s_2. L'effet domino séquentialise les requêtes.

FIGURE 6.2 – Exemple illustrant l'effet domino

alors de définir un ordre partiel sur l'ensemble des ressources du système si on définit un ordre total sur l'ensemble des couleurs. Les processus demanderont alors les ressources dans l'ordre des couleurs associées. Ceci réduit l'effet domino et améliore l'exploitation du parallélisme. Cependant le coloriage de graphe est un problème NP-complet et il est difficile de trouver un coloriage optimal.

Styer et Peterson [SP88] ont proposé une solution en considérant un coloriage quelconque (de préférence optimisé) pour réduire le temps d'attente avec un mécanisme d'annulation de verrouillage : un processus peut libérer une ressource (ou une couleur) même si il ne l'a pas encore utilisée. Ceci permet de casser dynamiquement les possibles chaînes de processus en attente causées par l'effet domino. Ainsi, un processus peut libérer toutes ou une partie de ses ressources acquises et essayer ensuite de les réacquérir jusqu'à satisfaction de la requête.

6.4.4 Algorithmes simultanés

Dans cette famille d'algorithme, les ressources ne sont plus ordonnées. Les algorithmes ont des mécanismes internes pour éviter les interblocages et permettre de verrouiller l'ensemble des ressources requises de manière atomique.

Chandy et Misra [CM84] ont décrit le problème du cocktail des philosophes où les processus (les philosophes) partagent un ensemble de ressources (les bouteilles). Ce problème est une extension du problème du dîner des philosophes ([Dij71]) où les processus partagent un ensemble de fourchettes. Le problème du cocktail des philosophes permet au site de demander un ensemble de ressources différent à chaque nouvelle requête contrai-

rement au problème du dîner des philosophes où les processus demandent en permanence le même ensemble de ressources. Le graphe de communication correspond directement au graphe des conflits et implique de le connaître a priori. Chaque processus partage une bouteille ou une fourchette (en fonction du problème considéré) avec chaque voisin. En orientant le graphe des conflits, il en résulte un graphe de précédence. Si les circuits sont évités dans ce graphe de précédence, les interblocages ne peuvent pas se produire. Il a été montré que le problème du dîner des philosophes respecte cette acyclicité. Cependant ceci n'est pas le cas pour le problème du cocktail. Chandy et Misra ont adapté le problème du cocktail en utilisant les procédures du dîner : pour verrouiller un sous-ensemble de bouteilles, un processus doit acquérir préalablement toutes les fourchettes de ses voisins avant de demander les bouteilles requises. Les fourchettes peuvent être vues comme des ressources auxiliaires et sont libérées lorsque le processus a obtenu toutes ses bouteilles. La phase d'acquisition des fourchettes sert à sérialiser les requêtes de bouteilles dans le système. Cette sérialisation évite les circuits dans le graphe de précédence et supprime par conséquent les interblocages.

Ginat et al. [GSA89] ont remplacé la phase d'acquisition des fourchettes de l'algorithme de Chandy-Misra par une horloge logique [Lam78]. Lorsqu'un processus demande des ressources, il estampille sa requête par une horloge logique et envoie un message à chaque voisin concerné. À la réception d'une requête, la bouteille associée est envoyée immédiatement au demandeur si l'estampille de la requête est plus petite que l'horloge du receveur. L'association d'une horloge logique et d'un ordre total sur les identifiants des processus permet de définir un ordre total sur les requêtes évitant ainsi les interblocages.

Dans [Rhe95, Rhe98] **Rhee** propose un ordonnanceur où chaque processus gère l'ordre d'attribution d'une des ressources. Chaque processus gérant une ressource maintient une file d'attente qui peut être réordonnancée en fonction des nouvelles requêtes pendantes. Les interblocages sont ainsi évités.

Maddi [Mad97] a proposé un algorithme basé sur un mécanisme de diffusion. Chaque ressource est représentée par un unique jeton. À chaque demande, un processus diffuse aux autres processus un message de requête estampillé par une horloge logique. À sa réception, la requête est stockée dans une file locale triée selon les estampilles temporelles. Cet algorithme peut être vu comme plusieurs instances de l'algorithme d'exclusion mutuelle classique de Suzuki-Kasami [SK85].

L'algorithme de Bouabdallah-Laforest [BL00] est plus détaillé que les précédents car il nous servira par la suite de base de comparaison. Cet algorithme est basé sur la circulation de jetons entre les processus. Chaque ressource est représentée par un unique jeton et est associée à une file d'attente distribuée dont le premier élément est le possesseur du jeton. Avant de demander un ensemble de ressources, le processus doit d'abord obtenir un unique jeton de contrôle. L'algorithme d'exclusion mutuelle gérant ce jeton de contrôle est l'algorithme de Naimi-Tréhel [NT87a]. Un jeton protégeant une ressource peut être possédé par un processus ou être directement stocké dans le jeton de contrôle si il n'est pas utilisé. Le jeton de contrôle contient donc un vecteur de M entrées. Si un jeton de

ressource n'est pas dans le jeton de contrôle, l'entrée correspondante du vecteur indique l'identifiant du dernier processus ayant demandé cette ressource. Ainsi, lorsqu'un site reçoit le jeton de contrôle, il prend les jetons associés aux ressources requises inutilisées et envoie pour chaque jeton manquant un message INQUIRE au dernier demandeur. Avant de libérer le jeton de contrôle, le processus attend un message d'acquittement (message INQACK1) de chaque processus à qui un message INQUIRE a été envoyé. Un message d'acquittement, peut contenir plusieurs jetons de ressource si l'envoyeur ne les utilise pas. Un processus recevant un message INQUIRE affecte l'identifiant de l'envoyeur dans les variables $next$ permettant de maintenir les différentes files distribuées. Ainsi lorsqu'un processus libère les jetons de ressources en sortant de section critique, il envoie directement la ressource dans un message INQACK2 aux sites pointés par les variables $next$ des ressources correspondantes. Le jeton de contrôle permet de sérialiser les requêtes ce qui assure qu'une requête sera enregistrée de manière atomique dans les différentes files d'attente distribuées. Ainsi, aucun cycle n'est possible dans l'union des files. La complexité en messages de cet algorithme est de $\mathcal{O}(Log(N))$.

Autres algorithmes. Des extensions du problème multi-ressource ont été proposées pour prendre en compte des systèmes dynamiques où l'ensemble des processus peut changer durant l'exécution de l'algorithme. Awerbuch et Saks [AS90] proposent le maintien d'une file d'attente globale qui peut être implémentée de manière centralisée ou distribuée. BarIlan et Peleg [BIP92] ont étendu la solution de Awerbuch-Saks pour améliorer le temps d'attente dans une version centralisée. Enfin, Weidman et al. [WPP91] ont adapté l'algorithme de Chandy-Misra.

6.5 Le modèle à plusieurs ressources en plusieurs exemplaires

Ce modèle de verrouillage de ressources est le plus général. Il fait l'union des deux modèles précédemment décrits en sections 6.3 et 6.4.

Il est généralement trivial de transformer un algorithme du modèle de plusieurs ressources à un seul exemplaire vers le modèle à plusieurs ressources en plusieurs exemplaires. En effet, la principale difficulté du multi-ressource est de résoudre le problème des interblocages. Comme le modèle de plusieurs ressources à un seul exemplaire résout ce problème, il suffit en général pour les algorithmes à jeton de transformer des booléens (présence ou pas du jeton) en compteurs (représentant un nombre d'exemplaires). Ainsi, il existe une extension possible sans dégradation de la complexité en messages de l'algorithme de Ginat et al. [GSA89] qui prend en compte plusieurs bouteilles par lien implémentée grâce à des compteurs. De même [Mad97] et [BL00] peuvent être étendus en remplaçant leurs jetons par des compteurs. Cependant, il n'est pas certain que ces transformations triviales résultent en une solution optimale.

Il existe aussi une extension possible à partir du modèle de plusieurs exemplaires d'une seule ressource. Cette extension se fait à partir de l'algorithme de Raynal [Ray91a] où chaque site diffuse pour chaque type de ressource voulue, un message de requête contenant

le nombre d'exemplaires requis. Il y a cependant une dégradation de la complexité en messages qui devient $\mathcal{O}((2N-1)*M)$.

6.6 Conclusion et synthèse

Nous avons présenté dans ce chapitre les différents modèles de généralisation de l'exclusion mutuelle. Il est possible de généraliser soit par le nombre d'exemplaires de la ressource soit par le nombre de types de ressource. La figure 6.3 schématise les différents modèles de généralisation recensés dans ce chapitre.

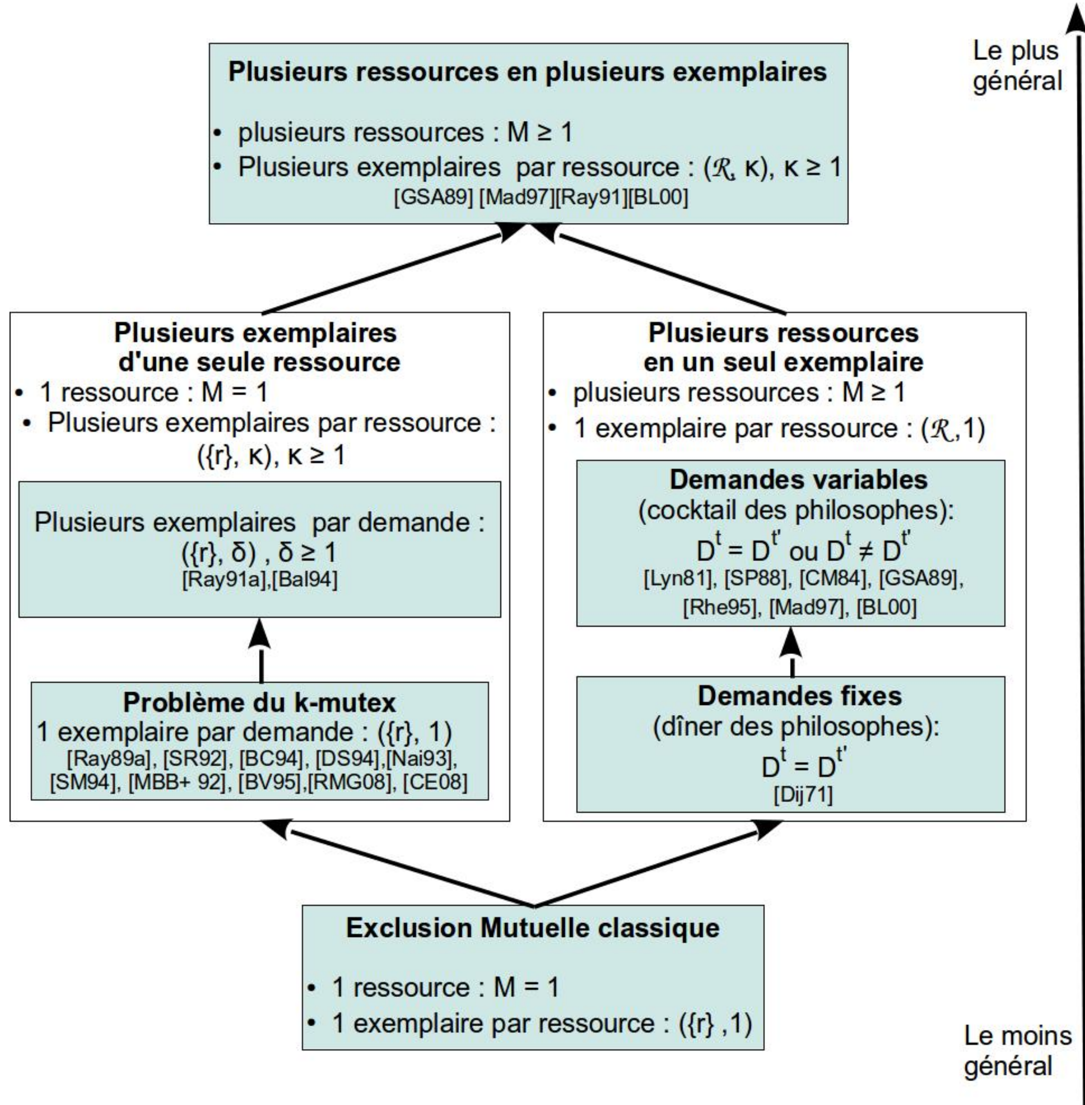


FIGURE 6.3 – Schéma récapitulatif de la généralisation de l'exclusion mutuelle distribuée

Chapitre 7

Verrouiller efficacement les ressources sans connaissance préalable des conflits

Sommaire

7.1 Introduction **101**
7.2 Objectifs **102**
7.3 Suppression du verrou global **104**
7.3.1 Mécanisme de compteurs 104
7.3.2 Ordonnancement total des requêtes 104
7.4 Évaluation **105**
7.4.1 Protocole d'évaluation 105
7.4.2 Résultats sur le taux d'utilisation 107
7.4.3 Résultats sur le temps d'attente 109
7.5 Ordonnancement dynamique **109**
7.5.1 Principes 109
7.5.2 Évaluation 111
7.6 Conclusion **112**

7.1 Introduction

Le chapitre précédent a présenté les différentes façons de généraliser le problème de l'exclusion mutuelle. Ce chapitre décrit une contribution dans le modèle à plusieurs ressources en un seul exemplaire (voir section 6.4) qui est facilement généralisable au cas d'exemplaires multiples. Dans ce modèle, la plupart des solutions de la littérature supposent que le graphe des conflits est connu a priori et ne change jamais pendant l'exécution de l'algorithme ce qui induit une hypothèse forte et irréaliste sur le système. Ces solutions peuvent néanmoins fonctionner sans connaissance préalable du graphe de conflits en considérant de façon pessimiste un graphe complet. Cependant une telle considération induit un coût de synchronisation élevé et dégrade le taux d'utilisation des ressources :

- Les algorithmes de la famille incrémentale sont très pénalisés par un effet domino inévitable des attentes car il est impossible de colorier de manière optimale un graphe complet (théorème de Brooks [Bro87], théorie des graphes).
- Les algorithmes de la famille simultanée font communiquer des processus qui ne rentrent pas en conflit et qui n'ont donc aucune raison d'interagir entre eux. Les algorithmes du cocktail des philosophes de Chandy-Misra [CM84] et de Ginat-Shankar-Agrawala [GSA89] utilisent dans le cas d'un graphe complet, un algorithme de diffusion (acquisition des ressources auxiliaires pour l'un et enregistrement des requêtes pour l'autre) impliquant en plus d'un coût de synchronisation élevé, une forte complexité en messages. Enfin, l'algorithme de Bouabdallah-Laforest [BL00] utilise une section critique de contrôle (verrou global) impliquant un goulot d'étranglement.

Dans ce chapitre nous proposons donc une solution permettant de verrouiller efficacement des ressources en réduisant le coût des synchronisations sans connaître a priori le graphe des conflits. Notre solution n'utilise pas de mécanisme de diffusion et limite la communication entre processus non conflictuels.

Dans la section 7.2, nous présentons les objectifs de cette contribution. La section 7.3 décrit un mécanisme supprimant la communication entre processus non conflictuels. L'implémentation distribuée de ce mécanisme est donnée en annexe A du manuscrit. L'évaluation de performance de ce mécanisme est donnée en section 7.4. La section 7.5 présente l'idée d'un mécanisme de préemption permettant d'améliorer le taux d'utilisation des ressources.

7.2 Objectifs

Puisque l'on considère un graphe de conflits inconnu, il est difficile de s'appuyer sur un algorithme de la famille *incrémentale* (cf. section 6.4.3) car leurs performances dépendent des techniques coûteuses de coloration de graphe. Par conséquent, notre choix s'est porté sur la famille *simultanée* (cf. section 6.4.4).

Ces algorithmes ont un point commun : ils possèdent un mécanisme permettant d'ordonner totalement les requêtes du système évitant ainsi les cycles dans le graphe de précédence et donc les interblocages. Ce mécanisme associe un identifiant unique à toute requête quel que soit son ensemble de ressources. En définissant un ordre total sur ces identifiants, on obtient facilement un ordonnancement sans interblocage. Chandy-Misra [CM84] s'appuie sur un algorithme de "Dîner des philosophes", [GSA89] et [Mad97] utilisent des horloges logiques et Bouabdallah-Laforest [BL00] utilise une section critique de contrôle. Cependant, l'utilisation d'horloge logique est généralement associée à un mécanisme de diffusion impliquant une forte complexité en messages, ce qui est problématique pour un passage à l'échelle. Les solutions de Chandy-Misra et de Bouabdallah-Laforest utilisent une section critique de contrôle : pour Chandy-Misra, le "dîner" dans un graphe complet est équivalent à l'algorithme d'exclusion mutuelle classique de Ricart-Agrawala [RA81], et l'algorithme de Bouabdallah-Laforest utilise l'algorithme de Naimi-Tréhel [NT87a]. L'algorithme de Ricart-Agrawala est basé sur des permissions et utilise aussi un mécanisme de diffusion (complexité en messages de $\mathcal{O}(2N-1)$) alors que l'algorithme de Naimi-Tréhel est basé sur la circulation d'un jeton dans un arbre dynamique (complexité en messages de

$\mathcal{O}(log(N))$). L'algorithme de Bouabdallah-Laforest apparaît aujourd'hui comme le plus efficace. On peut le décomposer en deux grandes étapes :

- *Première étape* : obtenir la section critique de contrôle.
- *Deuxième étape* : demander toutes les ressources nécessaires, attendre un acquittement et enfin libérer la section critique de contrôle de la première étape.

Bien que cet algorithme ait une bonne complexité en messages, il possède néanmoins deux limites qui dégradent le taux d'utilisation des ressources :

- deux processus non conflictuels doivent communiquer ensemble par le biais du jeton de contrôle impliquant un surcoût en synchronisation,
- l'ordonnancement des requêtes est statique : il dépend uniquement de l'ordre d'acquisition de la première étape (cette remarque est aussi valable pour les algorithmes avec horloge logique). En effet, il est impossible de revenir sur l'ordre de verrouillage. Une requête ne peut pas préempter une autre requête qui a eu le jeton de contrôle avant ce qui peut être problématique si on souhaite changer l'ordonnancement dynamiquement.

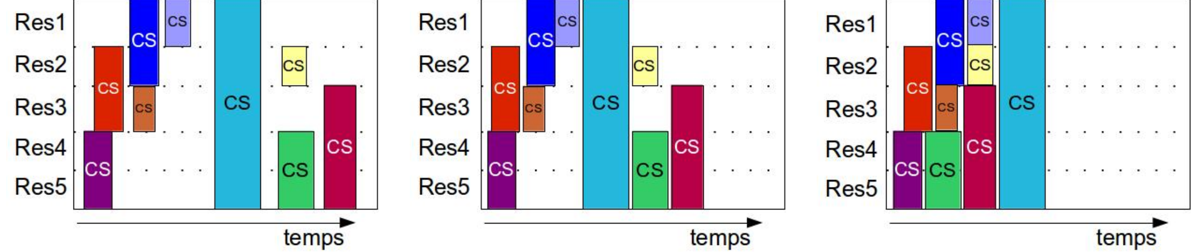
Nous avons donc deux objectifs :

- éviter l'utilisation d'un verrou global pour éviter la communication entre deux processus non conflictuels,
- pouvoir ordonnancer dynamiquement les requêtes.

La figure 7.1 illustre sous la forme d'un diagramme de Gantt l'impact de nos deux objectifs sur le taux d'utilisation des ressources dans un système à 5 ressources :

- la suppression du verrou global permet de réduire le temps entre deux sections critiques conflictuelles successives.
- le mécanisme d'ordonnancement dynamique permet de donner l'accès aux ressources à des processus dans les intervalles de temps où les ressources ne seraient pas utilisées dans le cas d'un ordonnancement statique (Figure 7.1(c))

On peut ainsi satisfaire le même ensemble de requêtes dans un intervalle de temps plus petit.



(a) Avec verrou global avec ordonnancement statique (b) Sans verrou global avec ordonnancement statique (c) Sans verrou global avec ordonnancement dynamique

FIGURE 7.1 – Illustration de l'impact des objectifs sur le taux d'utilisation

7.3 Suppression du verrou global

Dans cette section, nous décrivons de manière globale notre mécanisme permettant de supprimer le verrou global tout en assurant une sérialisation des requêtes.

7.3.1 Mécanisme de compteurs

Le but du jeton de contrôle est de donner un ordre de passage commun sur l'ensemble des files d'attentes associées aux ressources. Pour supprimer le verrou global, nous proposons d'utiliser un compteur par ressource. Ces derniers donneront un ordre de passage par ressource pour chacune des requêtes. Le système maintient donc M compteurs à accès exclusif. Dans notre cas, les compteurs sont stockés dans les jetons, un jeton étant associé à chaque ressource.

La première étape de notre algorithme consiste à demander la valeur de chaque compteur associé aux ressources requises et de les incrémenter de un : ceci assure une valeur différente à chaque nouvelle lecture.

Une fois que le site connaît l'ensemble des compteurs, sa requête peut être associée à un vecteur de M entiers dans l'ensemble $\mathbb{N}^M$ (les ressources non demandées ont une valeur nulle dans le vecteur). Par conséquent, la requête est définie de manière unique quel que soit l'instant où elle a été émise et quel que soit son ensemble de ressources requises. Ainsi, lors de la seconde étape, le processus pourra demander chaque ressource indépendamment les unes des autres en indiquant ce vecteur. Notons que ce mécanisme de compteur et le mécanisme de verrouillage sont décorrellés. Il est toujours possible de demander la valeur d'un compteur pendant que la ressource associée est utilisée.

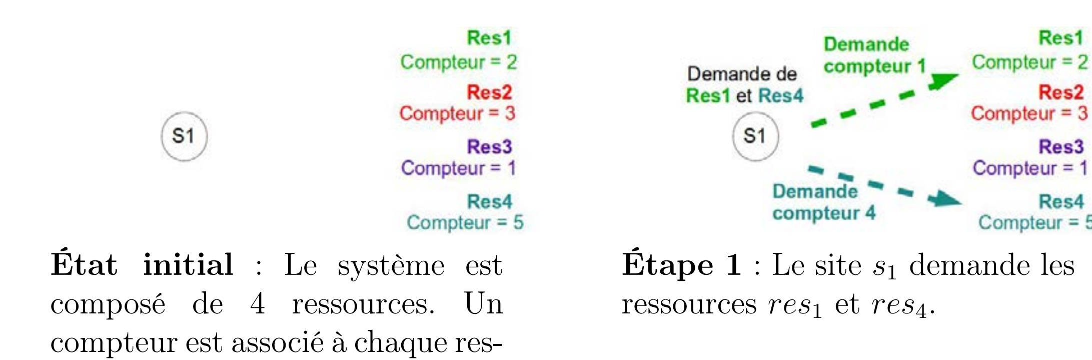
La figure 7.2 illustre ce mécanisme pour quatre ressources.

7.3.2 Ordonnancement total des requêtes

Une requête req_i émise par le site $s_i \in \Pi$ pour une ressource donnée est associée à deux informations : le site initiateur de la requête (s_i) et un vecteur $v_i \in \mathbb{N}^M$ (voir section 7.3.1). Les interblocages sont évités si nous définissons un ordre total sur les requêtes. Nous allons d'abord utiliser un ordre partiel sur l'ensemble des vecteurs en définissant une fonction $\mathcal{A} : \mathbb{N}^m \to \mathbb{R}$ qui transforme un vecteur d'entiers en un réel. Puisque $\mathcal{A}$ donne un ordre partiel (deux vecteurs peuvent avoir le même réel résultant), nous devons utiliser un ordre total arbitraire $\prec$ sur Π pour ordonner totalement les requêtes.

L'ordre total sur les requêtes est noté $\lhd$ où $req_i \lhd req_j$ ssi $\mathcal{A}(v_i) < \mathcal{A}(v_j) \vee (\mathcal{A}(v_i) = \mathcal{A}(v_j) \wedge s_i \prec s_j)$. Ainsi, en cas de valeur égale par $\mathcal{A}$, le site de plus petit identifiant d'après l'ordre total $\prec$ sera le plus prioritaire. Bien que ce mécanisme évite les interblocages en assurant que toutes les requêtes sont différentiables, le respect total de la propriété de vivacité dépend aussi de la définition de $\mathcal{A}$. En effet, la famine est évitée si la définition de $\mathcal{A}$ assure que toute requête sera dans un temps fini la plus petite dans l'ordre $\lhd$.

La fonction $\mathcal{A}$ permet de définir une heuristique donnant une politique d'ordonnancement sur les requêtes. C'est un paramètre de l'algorithme qui peut permettre de favoriser les requêtes en fonction du nombre de ressources utilisées. Notons que si $\mathcal{A}$ est bien choisie,



État initial : Le système est composé de 4 ressources. Un compteur est associé à chaque ressource.

Étape 1 : Le site s_1 demande les ressources res_1 et res_4.

Étape 2 : À chaque consultation de compteur, la valeur du compteur est renvoyée au demandeur puis incrémentée. Le vecteur est complété à chaque valeur reçue.

Étape 3 : Le site s_1 demande indépendamment chaque ressource voulue en indiquant le vecteur.

FIGURE 7.2 – Exemple d'exécution d'obtention des compteurs

les cas d'égalité seront peu probables. L'utilisation de l'identifiant du site ne posera donc pas de problème d'équité.

7.4 Évaluation

Dans cette section , nous présentons une évaluation de performances de notre algorithme que nous comparons avec deux algorithmes de l'état de l'art. Nous nous intéressons à deux métriques : le taux d'utilisation des ressources et le temps d'attente pour obtenir la section critique.

7.4.1 Protocole d'évaluation

Algorithmes considérés

Cette section d'évaluation de performances compare :

- un algorithme incrémental fixant un ordre prédéfini d'accès aux ressources et utilisant M instances d'algorithme de Naimi-Tréhel avec files locales [NT87b] (cf. section 2.5.2)
- l'algorithme de Bouabdallah-Laforest [BL00]

- l'implémentation distribuée de notre mécanisme basé sur les compteurs (voir annexe A)

Notre algorithme nécessitant la définition de la fonction $\mathcal{A}$, nous avons choisi de comparer deux politiques :

- **Politique équitable :** La première politique calcule la moyenne des compteurs non nuls : le résultat donne donc un compteur moyen par ressource demandée.
- **Politique favorisant les petites requêtes** Une solution naïve pour améliorer le taux d'utilisation serait d'avoir une définition de $\mathcal{A}$ qui favoriserait les requêtes requérant peu de ressources. En effet, ces requêtes ont moins de chances d'être en conflit avec une autre du fait de leurs petites tailles. Dans cette politique, $\mathcal{A}$ est égale à la moyenne de l'ensemble des valeurs du vecteur de la requête. Puisque les ressources non requises ont une valeur nulle dans le vecteur, les petites requêtes demandant peu de ressources en seront privilégiées.

Dans les deux cas, la famine est impossible car les compteurs augmentent à chaque nouvelle requête impliquant que la valeur minimum de $\mathcal{A}$ augmentera également à chaque nouvelle requête. La propriété de vivacité est ainsi toujours respectée. L'avantage du mécanisme de la fonction $\mathcal{A}$ réside dans le fait que la famine est évitée seulement grâce à un calcul qui n'implique aucun surcoût en communication.

Plate-forme et paramètres d'expérimentation

Les expériences ont été menées sur un cluster de 32 machines (Grid5000 Lyon) avec un processus par machine pour éviter les effets de bords de contention sur les cartes réseau. Chaque machine a deux processeurs Xeon 2.4GHz, 32 GB de mémoire RAM et fonctionne sous Linux 2.6. Les machines sont reliées par un switch Ethernet 10 Gbit/s. Les algorithmes ont été implémentées en C++ et OpenMPI avec la version 4.7.2 de gcc.

Une application est caractérisée par :

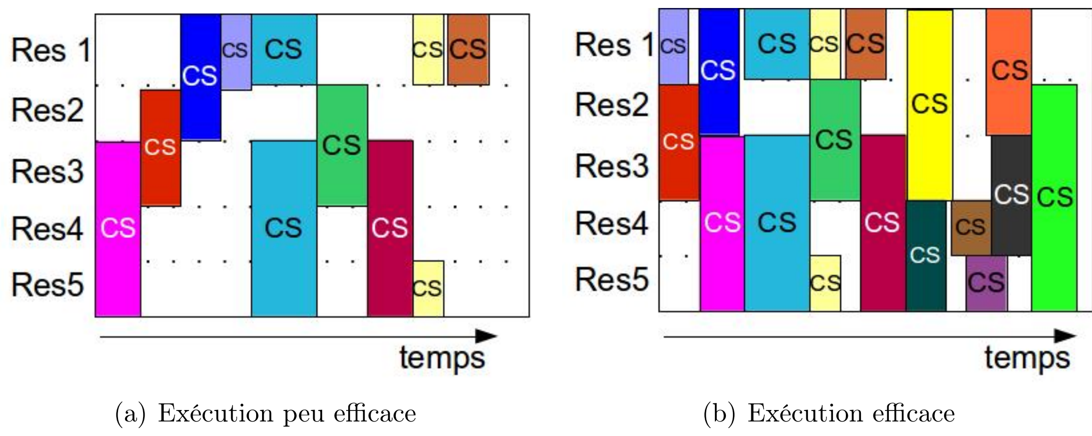
- N : le nombre de processus (32 dans notre cas).
- M : le nombre total de ressources dans le système (80 dans notre cas).
- α : le temps d'exécution de la section critique (variable entre 5 ms et 35 ms selon le nombre de ressources demandées).
- β : l'intervalle de temps entre le moment où un processus libère la section critique et le moment où il la redemande.
- γ : la latence réseau pour envoyer un message entre deux processus.
- ρ : le rapport $\beta/(\alpha+\gamma)$, qui exprime la fréquence à laquelle la section critique est demandée. La valeur de ce paramètre est inversement proportionnelle à la charge en requêtes : une valeur basse donne une charge haute et vice-versa. Dans nos expériences, nous avons considéré une charge haute et moyenne.
- $SizeReq$: le nombre maximum de ressources qu'un site peut demander. Ce paramètre est compris entre 1 et M.

À chaque nouvelle requête, un processus choisit x ressources uniformément entre 1 et $SizeReq$. Le temps de section critique de la requête dépend alors de la valeur de x : plus cette valeur est grande et plus le temps de section critique risque d'être grand car nous considérons qu'une requête demandant beaucoup de ressources doit en pratique avoir un

plus grand temps de calcul en section critique.

7.4.2 Résultats sur le taux d'utilisation

La métrique principale pour l'évaluation des algorithmes est le taux d'utilisation. Cette métrique globale est le pourcentage de temps où les ressources sont utilisées. On peut la voir comme le pourcentage de l'aire colorée des diagrammes de la figure 7.3. Ainsi, la figure 7.3(a) donne un exemple d'exécution où les ressources ne sont pas utilisées efficacement alors que la figure 7.3(b) illustre une exécution avec un meilleur taux d'utilisation (zone blanche moins importante).



(a) Exécution peu efficace (b) Exécution efficace

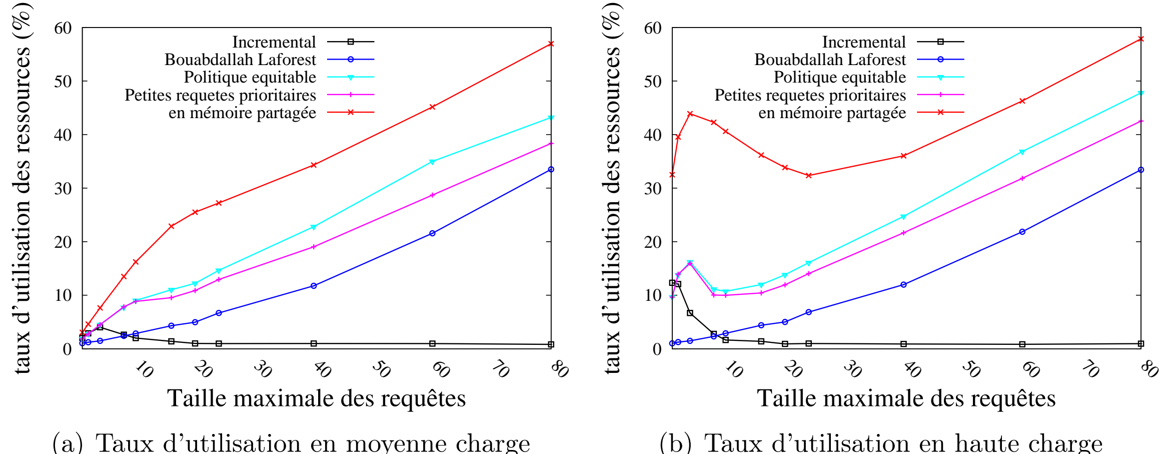
FIGURE 7.3 – Illustration du taux d'utilisation pour l'exclusion mutuelle généralisée

Sur les graphiques de la figure 7.4, nous faisons varier en abscisse la taille maximale des requêtes (paramètre $SizeReq$). Les figures 7.4(a) et 7.4(b) montrent l'impact de ce paramètre sur le taux d'utilisation respectivement en moyenne et haute charge. L'augmentation de la taille maximale fait varier deux facteurs :

- **la taille moyenne des requêtes** qui aura un effet positif sur la métrique : on utilise plus de ressources à chaque section critique
- **le nombre de conflits** qui aura un effet négatif sur la métrique : la parallélisation des requêtes est plus difficile.

Performances globales

Dans la figure 7.4 nous avons ajouté une courbe témoin représentant un algorithme en mémoire partagée ayant une connaissance globale des nouvelles requêtes émises et un coût de synchronisation nul. Cette connaissance globale lui permet d'ordonnancer les requêtes afin d'utiliser au mieux les ressources. Ainsi, l'écart avec cette courbe donnera le coût de synchronisation des algorithmes. En cas de charge moyenne, nous pouvons voir que l'impact des conflits est toujours moins important que le facteur taille (les courbes augmentent en permanence). En revanche, nous remarquons qu'en forte charge le comportement global passe par trois phases :



(a) Taux d'utilisation en moyenne charge (b) Taux d'utilisation en haute charge

FIGURE 7.4 – Impact sur le taux d'utilisation

- en cas de petites requêtes : la courbe augmente car le facteur de la taille moyenne est plus important que le facteur conflit.
- en cas requêtes de taille inférieure à la moyenne : le facteur conflit reprend le dessus sur le facteur taille.
- en cas de requête de taille supérieure à la moyenne : la courbe augmente de nouveau grâce au facteur taille qui redevient plus important que le facteur conflit.

Les requêtes de taille moyenne sont donc les plus coûteuses en matière de taux d'utilisation car leur taille est assez grande pour générer des conflits mais trop insuffisante pour avoir un taux d'utilisation important.

Performances des algorithmes distribués

Une première remarque sur ces résultats, est que l'algorithme incrémental ne suit en aucun cas le comportement global du taux d'utilisation. En effet, il ne fait que diminuer à cause de l'effet domino. L'algorithme de Bouabdallah-Laforest ne profite pas du parallélisme potentiel lorsqu'il y a peu de conflits à cause de sa section critique de contrôle qui est très coûteuse en synchronisation.

De manière générale, notre algorithme permet d'avoir un meilleur taux d'utilisation car il réduit les communications entre processus non conflictuels. On peut remarquer une différence entre les deux politiques : la deuxième politique donnant une priorité aux petites requêtes a un taux d'utilisation moins important que la première politique lorsque la taille maximale augmente. Ceci s'explique par le fait que les requêtes de tailles moyennes (les plus coûteuses en termes de dégradation du taux d'utilisation comme nous avons pu le voir précédemment) sont privilégiés par rapport aux requêtes de taille plus importante qui sont dans cette politique moins prioritaires alors qu'elles maximisent l'utilisation des ressources. Ainsi, on pouvait penser que favoriser les requêtes de petites tailles améliorerait le taux d'utilisation grâce à leurs faibles impacts en termes de conflits. Cette solution n'est donc pas satisfaisante.

7.4.3 Résultats sur le temps d'attente

Dans la figure 7.5, nous montrons le temps d'attente moyen pour entrer en section critique (figures 7.5(b) et 7.5(a)) ainsi que le temps d'attente par taille de requêtes (figures 7.5(d) et 7.5(c)). Nous n'indiquons pas les performances de l'algorithme incrémental car l'effet domino le pénalise énormément comme nous avons pu le remarquer dans les précédentes courbes : le temps d'attente moyen des points considérés est trop important par rapport au temps de l'expérience.

Nous pouvons remarquer dans les figures 7.5(b) et 7.5(a) que nos deux politiques ont un temps d'attente moyen plus faible que l'algorithme de Bouabdallah-Laforest grâce à leur coût de synchronisation plus faible. Cependant ce dernier a une variance moins importante. Ceci s'explique par le fait que cet algorithme est très équitable : on peut en effet remarquer sur les figures 7.5(d) et 7.5(c) que le temps d'attente est similaire entre les différentes tailles de requête. Cette équité est due au mécanisme d'acquisition du jeton de contrôle qui ne permet aucune préemption entre les différentes requêtes.

La politique favorisant les petites requêtes a une grande variance comparée à la politique équitable. En effet, puisqu'elle se base sur un ordonnancement similaire à un algorithme à priorité, nous pouvons remarquer dans les figures 7.5(d) et 7.5(c) que les grandes requêtes ont un temps d'attente très grand par rapport aux petites requêtes. Bien que la politique équitable soit la plus performante, on peut remarquer qu'elle ne favorise pas les petites requêtes dans son ordonnancement. En effet, on peut constater que ces requêtes sont plus pénalisées : leur temps d'attente moyen est le plus important. Cependant, leur variance est également importante. En effet, dans cette politique, l'ordre d'accès d'une requête concernant une seule ressource dépend de la valeur du compteur correspondant. La moyenne n'est donc calculée qu'à partir de cette unique valeur qui peut varier selon la popularité de la ressource demandée : une ressource très populaire aura une valeur de compteur importante par rapport à d'autres ressources moins populaires.

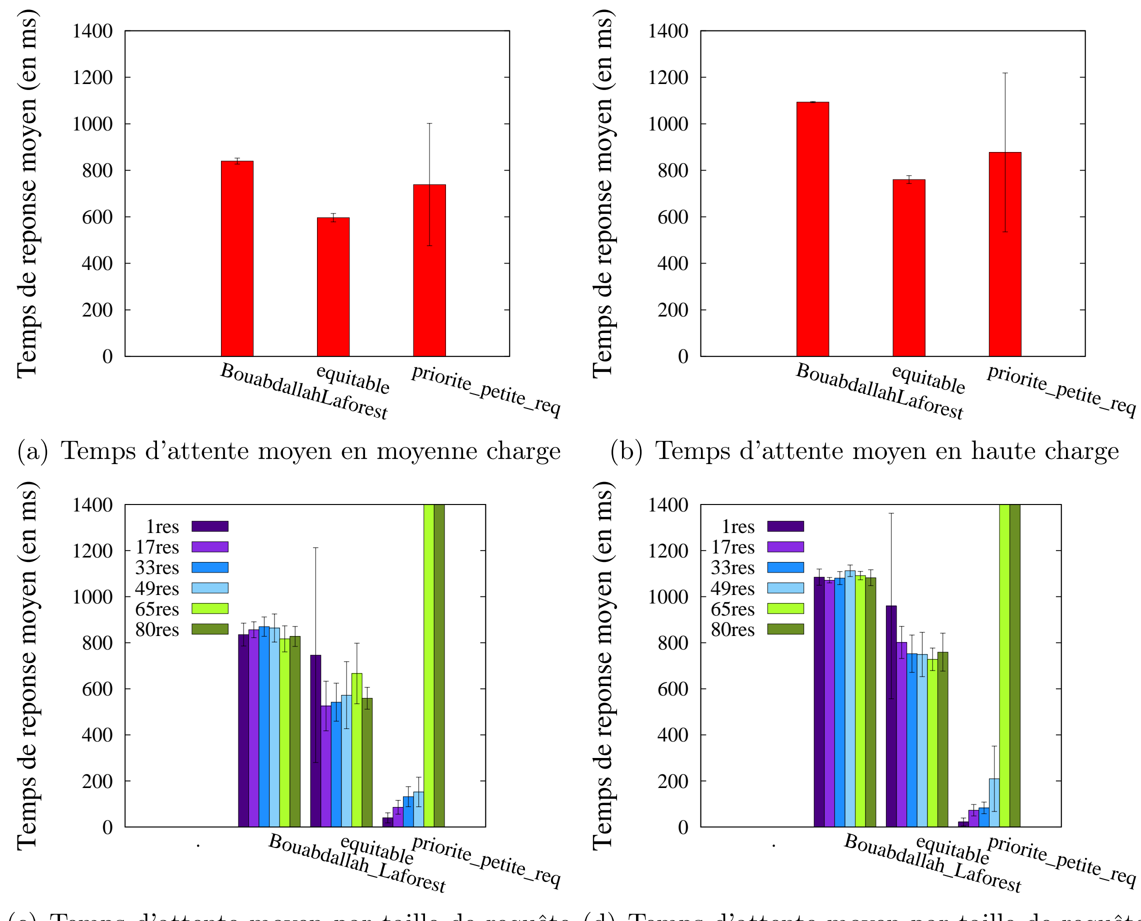
7.5 Ordonnancement dynamique

Dans cette section nous décrivons le principe du mécanisme de préemption permettant d'ordonnancer les requêtes dynamiquement.

7.5.1 Principes

Pour améliorer le taux d'utilisation des ressources, il peut paraître intéressant d'introduire un mécanisme de "prêt". En effet dans bien des cas les ressources sont acquises progressivement pour n'être réellement utilisées qu'une fois l'ensemble des jetons acquis. De nombreuses ressources sont ainsi verrouillées par des sites ne pouvant pas entrer en section critique, limitant d'autant le taux d'utilisation des ressources partagées.

L'idée de l'ordonnancement dynamique serait de limiter la détention des ressources aux seules sections critiques, en offrant la possibilité de prêter des ressources inutilement possédées. L'introduction de "prêt" n'est cependant pas triviale, puisqu'elle remet en cause la propriété de vivacité : une ressource acquise puis prêtée ne permet plus d'entrer en section critique. L'algorithme nécessitera donc des mécanismes complexes qui pourront



(a) Temps d'attente moyen en moyenne charge

(b) Temps d'attente moyen en haute charge

(c) Temps d'attente moyen par taille de requête en moyenne charge

(d) Temps d'attente moyen par taille de requête en haute charge

FIGURE 7.5 – Impact sur le temps d'attente moyen

peser sur les performances. Il nous faudra donc veiller à en limiter le surcoût pour espérer gagner globalement sur le taux d'utilisation.

Pour assurer la vivacité, le mécanisme de "prêt" doit assurer qu'un site finisse à terme par disposer simultanément de ses ressources précédemment acquises. Il devra ainsi éviter l'éparpillement des ressources et les interblocages dus aux ressources prêtées.

Éviter l'éparpillement des ressources

Le fait qu'on ne puisse pas assurer qu'un processus possédera l'ensemble des ressources requises induit des famines. Pour palier à ce problème, on peut introduire un mécanisme de compteurs qui permet de borner le nombre de prêts et ainsi assurer dans un temps fini que l'on ne prêtera plus de ressource. Cependant, les latences induites pourraient rendre inefficace l'algorithme.

Nous proposons donc un mécanisme simple en restreignant le prêt à un seul site à la fois. Le processus prêteur est ainsi assuré de réacquérir et en temps fini toutes les ressources prêtées puisque le temps de section critique de l'emprunteur est supposé fini.

Éviter les interblocages dus aux ressources prêtées

Le fait d'emprunter des ressources à plusieurs sites peut induire des cycles dans les files d'attente ce qui conduit à des interblocages. Pour régler le problème il serait envisageable d'avoir un mécanisme de prêt à deux phases mais ceci serait trop complexe et induirait des latences trop grandes.

Nous proposons donc un mécanisme simple en restreignant l'emprunt à un seul site uniquement s'il possède toutes les ressources manquantes. Un site rentrera directement en section critique à la réception des ressources prêtées.

7.5.2 Évaluation

Ce mécanisme n'a pas été implémenté dans notre algorithme distribué. Nous l'avons cependant développé dans un simulateur en mémoire partagée,i.e., sans mécanisme distribué de synchronisation. Cette évaluation a pour but de mesurer le potentiel d'un tel mécanisme de prêt. Nous considérons le même nombre de processus et le même nombre de ressources que la section 7.4 (32 processus et 80 ressources).

La figure 7.6 montre les résultats de cette étude pour le taux d'utilisation en fonction de la taille maximale des requêtes en forte charge. Nous comparons l'approche basée sur les prêts avec :

- un ordonnancement avec vue globale correspondant à l'algorithme témoin de la section 7.4.
- un ordonnancement avec vue locale correspondant à une implémentation en mémoire partagée de l'ordonnancement statique.

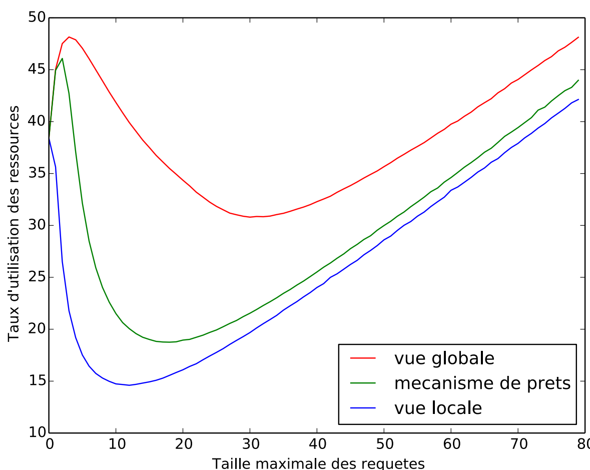


FIGURE 7.6 – Évaluation du mécanisme de prêt en mémoire partagée

Nous pouvons remarquer que le mécanisme de prêt améliore le taux d'utilisation quelle que soit la taille maximale de requête par rapport à l'ordonnancement à vue locale. Ce gain est le plus important lorsque la taille maximale de requête est comprise entre 5 et 20

ressources. On peut en effet constater dans ces valeurs d'abscisse un gain compris entre 100% et 15%. La taille des petites requêtes rend plus facile la préemption. C'est ce qui explique qu'en cas de grande requête (à partir d'une taille maximale de 40), le gain est réduit et ne dépasse pas 8 %. Cette évaluation donne donc des résultats prometteurs.

7.6 Conclusion

Nous avons présenté dans ce chapitre, un nouveau schéma pour verrouiller un ensemble de ressources différentes dans un système distribué. Ce schéma ne nécessite pas de connaître a priori le graphe de conflits et limite la communication entre les processus qui n'ont pas de conflit en remplaçant le verrou global par un mécanisme de compteur. Notre solution améliore le taux d'utilisation des ressources et réduit le temps d'attente moyen. L'ordonnancement des requêtes peut être modifié de manière modulaire grâce à la définition de la fonction $\mathcal{A}$. Nous avons comparé deux politiques d'ordonnancement : une politique classique et une politique donnant priorité aux petites requêtes dans l'idée de privilégier les requêtes qui génèrent moins de conflit. Nous avons pu nous rendre compte que cette politique n'est pas satisfaisante car elle augmente le temps d'attente moyen des grandes requêtes et réduit le taux d'utilisation des ressources par rapport à la première politique. Le mécanisme de compteur n'est cependant pas suffisant pour casser complètement l'effet domino : des attentes en cascades peuvent toujours se produire.

Nous proposons d'associer un nouveau mécanisme de prêt en permettant de réordonner dynamiquement les requêtes. Il permet de réduire la probabilité que l'effet domino se produise. Une première évaluation par simulation montre une amélioration du taux d'utilisation. Il serait intéressant d'introduire ce mécanisme dans notre implémentation distribuée afin d'évaluer son coût en synchronisation dans un environnement réel.

Quatrième partie

Conclusion générale

Chapitre 8

Conclusion générale

Sommaire

8.1 Conclusion 115
8.2 Perspectives 116
8.2.1 Perspectives spécifiques aux contributions 117
8.2.2 Perspectives globales 117

8.1 Conclusion

Cette thèse a abordé le problème fondamental de la synchronisation pour l'accès à des ressources partagées dans les systèmes répartis. Nous avons fait trois contributions sur des extensions du problème de l'exclusion mutuelle distribuée. Nous avons proposé des algorithmes pour gérer la priorité de demande d'accès, prendre en compte des contraintes de temps et traiter plusieurs types de ressources. Les évaluations de ces algorithmes en environnement réel ont montré leur efficacité.

Requêtes à priorités : Dans l'étude de l'état de l'art des algorithmes d'exclusion mutuelle à priorité (chapitre 3) nous avons distingué deux familles d'algorithmes : les algorithmes à priorités statiques et ceux à priorités dynamiques. Les approches statiques respectent de manière stricte les priorités fixées par les applications et peuvent donc induire des famines. En revanche, les algorithmes à priorités dynamiques assurent la propriété de vivacité mais génèrent beaucoup d'inversions de priorités dues à l'augmentation trop rapide des petites priorités. Nous avons donc conçu un algorithme reposant sur la circulation d'un jeton dans une topologie d'arbre statique qui lui permet d'avoir une complexité moyenne en messages en $\mathcal{O}(Log(N))$. L'évaluation en environnement réel permet de constater que notre algorithme réduit considérablement le nombre d'inversions de priorités grâce à un mécanisme de retard d'incrémentation paramétrable par une fonction de palier. De plus, nous avons réussi à réduire le surcoût en messages de notre mécanisme d'ajustement dynamique des priorités en prenant en compte la localité des requêtes dans la topologie reliant les nœuds. Dans un second temps, nous avons mis en évidence l'impact

de cette topologie sur le temps de réponse des différentes priorités. Nous avons montré que dans certains cas les temps d'attente étaient énormes pour les requêtes à petites priorités. Nous avons donc proposé un algorithme qui incrémente les priorités en considérant l'ensemble des requêtes émises. Les expériences ont montré que notre algorithme présentait un bon compromis entre le temps d'attente maximum et le nombre d'inversions de priorités.

Requêtes à contraintes temporelles : Nous avons ensuite conçu un algorithme se basant également sur la circulation d'un jeton dans une topologie d'arbre statique et permettant de satisfaire des requêtes avant une date d'échéance. Cette extension de l'exclusion mutuelle permet de répondre à des contraintes temporelles (SLA) pour l'accès aux ressources que l'on peut trouver dans les systèmes de réservation de ressources des Clouds. Pour éviter la violation de SLA, i.e., les requêtes satisfaites après leur date d'échéance, notre algorithme fait un contrôle d'admission des requêtes et les ordonnance selon une politique *Earliest Deadline First*. Nos évaluations ont permis de constater que la politique d'admission choisie minimisait la quantité de violations. Cependant, nous avons pu constater que la politique d'ordonnancement EDF dans une topologie statique dégradait le taux d'utilisation de la ressource critique. Un mécanisme de préemption a donc été ajouté. Il permet d'utiliser au mieux le jeton lors de son transfert dans la topologie sans générer de violations. Cette préemption est paramétrable par un entier et permet d'améliorer jusqu'à 70% le taux d'utilisation.

Requêtes à plusieurs ressources : En dernier lieu, nous avons proposé un algorithme d'exclusion mutuelle généralisée permettant de verrouiller un ensemble de ressources hétérogènes. Par rapport à une grande majorité des solutions que l'on peut trouver dans la littérature, notre algorithme ne nécessite pas de connaître à l'avance le graphe des conflits liant les requêtes. De plus notre solution réduit les coûts de synchronisation en évitant que deux processus non conflictuels communiquent entre eux. Les évaluations d'une implémentation distribuée de notre mécanisme sur la plateforme nationale Grid5000 ont montré qu'il était possible d'améliorer le taux d'utilisation des ressources d'un facteur 1 à 10 par rapport aux algorithmes de l'état de l'art et de réduire le temps d'attente moyen des requêtes. Dans un second temps nous avons simulé en mémoire partagée un ordonnancement dynamique. Les expériences ont montré qu'un gain en termes de taux d'utilisation est possible en particulier lorsque la taille moyenne des requêtes reste faible.

8.2 Perspectives

Cette section traite des différentes perspectives de travail de cette thèse. Nous aborderons les perspectives spécifiques aux contributions dans un premier temps et nous traiterons des perspectives globales de la thèse dans un second temps.

8.2.1 Perspectives spécifiques aux contributions

Requêtes à contraintes temporelles

Dans notre étude nous avons proposé un contrôle d'admission avec des dates d'échéance strictes qui ne faisait pas de surréservation dans les décisions locales. Il serait alors possible de considérer une politique plus souple qui accepterait plus de requêtes avec une marge d'erreur paramétrable. Il serait donc intéressant d'étudier l'impact de cette marge sur le taux de violations et le taux d'utilisation de la ressource critique. On pourrait également envisager une négociation lors de la soumission d'une requête. En cas de réponse négative du contrôle d'admission, il serait possible d'estimer les modifications de contraintes nécessaires (retardement de la date d'échéance ou bien diminution du temps de section critique par exemple) pour que la requête puisse être acceptée. Ces estimations pourraient être proposées par le système à l'utilisateur qui pourra à sa guise les accepter ou les refuser. Concernant le mécanisme de préemption de requête pour augmenter le taux d'utilisation, il serait possible d'exploiter plus finement la topologie reliant les nœuds pour router plus efficacement le jeton . En effet, le porteur du jeton connaît l'ensemble des requêtes à satisfaire dans le système. Il lui est alors possible de tirer parti de cette connaissance globale : on peut privilégier par exemple le sous-arbre qui donnera le plus de sections critiques tout en assurant que la première requête de la file d'attente sera satisfaite à temps. Enfin, il reste possible de ne plus considérer l'hypothèse de connaissance du temps de transmission γ_{max} entre deux sites voisins en incluant des mécanismes d'estimation et de calcul de latence réseau.

Requêtes à plusieurs ressources

Notre mécanisme de préemption par prêts de ressources a été évalué uniquement sur un simulateur en mémoire partagée. Les résultats sont particulièrement prometteurs et une perspective à court terme est donc d'ajouter ce mécanisme dans notre implémentation distribuée et d'en évaluer le coût de synchronisation dans un environnement réel. Une autre perspective serait de généraliser notre algorithme à plusieurs exemplaires par ressource.

D'autre part, puisque notre solution limite la communication entre processus non conflictuels, il serait particulièrement intéressant de tester notre algorithme sur une topologie physique hiérarchique telle qu'on peut la trouver dans les grandes infrastructures de Clouds. En effet, la suppression du verrou global permet maintenant d'éviter la communication inutile entre les différents sites géographiques. Il y a ainsi moins de risques à ce qu'un processus attende un message d'un site très éloigné. Cette étude permettrait de mettre en évidence nos gains de performances en termes de taux d'utilisation et de temps d'attente.

8.2.2 Perspectives globales

Combinaison

Cette thèse a abordé trois problèmes différents dans la réservation de ressources : les priorités, les contraintes de temps et l'hétérogénéité des ressources. L'idée d'étudier une

solution qui combinerait plusieurs de ces problèmes semble assez naturelle. On pourrait ainsi prendre en compte dans une même requête un ensemble de ressources requises avec une priorité ou/et une date d'échéance. Ce type de requête a de l'intérêt dans les systèmes de réservation de ressources tel que OAR [OAR] qui gère l'accès des utilisateurs à des ressources physiques dans une grille de calcul mais de manière centralisée. Chaque utilisateur peut ainsi verrouiller un ensemble de ressources hétérogènes en indiquant éventuellement une date de réservation requise (dates d'échéance) ainsi qu'un niveau de priorité pour différencier par exemple les administrateurs des utilisateurs.

Élasticité

Enfin, dans cette thèse nous avons traité deux particularités des systèmes large échelle : l'hétérogénéité des ressources et l'hétérogénéité des requêtes que l'on peut associer à un niveau de SLA. Cependant, les Nuages informatiques ont une particularité supplémentaire : l'élasticité. Cette particularité est due à la virtualisation des machines qui peuvent s'ajouter, se supprimer et se déplacer d'un site physique à l'autre en fonction des pics de charge. Cela implique des changements dynamiques dans la topologie qu'il serait intéressant d'étudier.

Bibliographie

[AFG+09] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy H. Katz, Andrew Konwinski, Gunho Lee, David A. Patterson, Ariel Rabkin, and Matei Zaharia. Above the clouds : A berkeley view of cloud computing. Technical report, University of California at Berkeley, 2009.

[App] Google app engine. "`https://developers.google.com/appengine/`".

[AS90] B. Awerbuch and M. Saks. A dining philosophers algorithm with polynomial response time. In *Foundations of Computer Science, 1990. Proceedings., 31st Annual Symposium on*, pages 65 –74 vol.1, oct 1990.

[AWCR13] Aoxueluo, Weigang Wu, Jiannong Cao, and Michel Raynal. A generalized mutual exclusion problem and its algorithm. In *Parallel Processing (ICPP), 2013 42nd International Conference on*, pages 300–309, 2013.

[Bal94] Roberto Baldoni. An $\mathrm{o}(\mathrm{n}^{\mathrm{m}/(\mathrm{m}+1)})$ distributed algorithm for the kth-out of-m resources allocation problem. In *ICDCS*, pages 81–88, 1994.

[BAS04] Marin Bertier, Luciana Bezerra Arantes, and Pierre Sens. Hierarchical token based mutual exclusion algorithms. In *CCGRID*, pages 539–546, 2004.

[BAS06] Marin Bertier, Luciana Bezerra Arantes, and Pierre Sens. Distributed mutual exclusion algorithms for grid applications : A hierarchical approach. *J. Parallel Distrib. Comput.*, 66(1) :128–144, 2006.

[BC94] Roberto Baldoni and Bruno Ciciani. Distributed algorithms for multiple entries to a critical section with priority. *Inf. Process. Lett.*, 50(3) :165–172, 1994.

[BIP92] Judit Bar-Ilan and David Peleg. Distributed resource allocation algorithms (extended abstract). In *WDAG*, pages 277–291, 1992.

[BL00] Abdelmadjid Bouabdallah and Christian Laforest. A distributed token/based algorithm for the dynamic resource allocation problem. *Operating Systems Review*, 34(3) :60–68, 2000.

[Bro87] Rowland Leonard Brooks. On colouring the nodes of a network. In *Classic Papers in Combinatorics*, pages 118–121. Springer, 1987.

[Bur06] Mike Burrows. The chubby lock service for loosely-coupled distributed systems. In *OSDI*, pages 335–350. USENIX Association, 2006.

[BV95] Shailaja Bulgannawar and Nitin H. Vaidya. A distributed k-mutual exclusion algorithm. In *ICDCS*, pages 153–160, 1995.

[CE08] Pranay Chaudhuri and Thomas Edward. An algorithm for k-mutual exclusion in decentralized systems. *Computer Communications*, 31(14) :3223–3235, 2008.

[Cha94] Ye-In Chang. Design of mutual exclusion algorithms for real-time distributed systems. *J. Inf. Sci. Eng.*, 11(4) :527–548, 1994.

[CM84] K. Mani Chandy and Jayadev Misra. The drinking philosopher's problem. *ACM Trans. Program. Lang. Syst.*, 6(4) :632–646, 1984.

[CR83] O.S.F. Carvalho and G. Roucairol. On mutual exclusion in computer networks. *Comm. ACM*, 26 :145–147, 1983.

[CSL90] Ye-In Chang, Mukesh Singhal, and Ming T. Liu. A fault tolerant algorithm for distributed mutual exclusion. In *SRDS*, pages 146–154, 1990.

[CSL91] Y.-I. Chang, M. Singhal, and M.T. Liu. A dynamic token-based distributed mutual exclusion algorithm. In *Computers and Communications, 1991. Conference Proceedings., Tenth Annual International Phoenix Conference on*, pages 240–246, 1991.

[Dij65] Edsger W. Dijkstra. Solution of a problem in concurrent programming control. *Commun. ACM*, 8(9) :569, 1965.

[Dij71] E. W. Dijkstra. Hierarchical ordering of sequential processes. *Acta Informatica*, 1 :115–138, 1971. 10.1007/BF00289519.

[DS94] Naomi S. DeMent and Pradip K. Srimani. A new algorithm for k mutual exclusions in distributed systems. *Journal of Systems and Software*, 26(2) :179–191, 1994.

[EC2] Amazon ec2. "`http://aws.amazon.com/ec2/`".

[Erc04] Kayhan Erciyes. Cluster based distributed mutual exclusion algorithms for mobile networks. In *Euro-Par*, pages 933–940, 2004.

[g5k] Grid5000. "`https://www.grid5000.fr/mediawiki/index.php/Grid5000:Home`".

[Gos90] Andrzej M. Goscinski. Two algorithms for mutual exclusion in real-time distributed computer systems. *J. Parallel Distrib. Comput.*, 9(1) :77–82, 1990.

[GSA89] David Ginat, A. Udaya Shankar, and Ashok K. Agrawala. An efficient solution to the drinking philosophers problem and its extension. In *WDAG (Disc)*, pages 83–93, 1989.

[HPR88] Jean-Michel Helary, Noel Plouzeau, and Michel Raynal. A distributed algorithm for mutual exclusion in an arbitrary network. *Comput. J.*, 31(4) :289–295, 1988.

[HT01] Ahmed Housni and Michel Trehel. Distributed mutual exclusion token-permission based by prioritized groups. In *AICCSA*, pages 253–259, 2001.

[JNW96] Theodore Johnson and Richard E. Newman-Wolfe. A comparison of fast and low overhead distributed priority locks. *J. Parallel Distrib. Comput.*, 32(1) :74–89, 1996.

[Jou98] Yuh-Jzer Joung. Asynchronous group mutual exclusion (extended abstract). In *PODC*, pages 51–60, 1998.

[Jou03] Yuh-Jzer Joung. Quorum-based algorithms for group mutual exclusion. *Parallel and Distributed Systems, IEEE Transactions on*, 14(5) :463–476, 2003.

[JPT03] Prasad Jayanti, Srdjan Petrovic, and King Tan. Fair group mutual exclusion. In *Proceedings of the Twenty-second Annual Symposium on Principles of Distributed Computing*, PODC '03, pages 275–284, New York, NY, USA, 2003. ACM.

[KC10] Sukhendu Kanrar and Nabendu Chaki. Fapp : A new fairness algorithm for priority process mutual exclusion in distributed systems. *JNW*, 5(1) :11–18, 2010.

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21 :558–565, July 1978.

[Lan78] Gerard Le Lann. Algorithms for distributed data-sharing systems which use tickets. In *Berkeley Workshop*, pages 259–272, 1978.

[LH89] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems (TOCS)*, 7(4) :321–359, 1989.

[LKN+09] Alexander Lenk, Markus Klems, Jens Nimis, Stefan Tai, and Thomas Sandholm. What's inside the cloud ? an architectural map of the cloud landscape. *Software Engineering Challenges of Cloud Computing, ICSE Workshop on*, 0 :23–31, 2009.

[Lyn81] Nancy A. Lynch. Upper bounds for static resource allocation in a distributed system. *J. Comput. Syst. Sci.*, 23(2) :254–278, 1981.

[Mad97] Aomar Maddi. Token based solutions to m resources allocation problem. In *SAC*, pages 340–344, 1997.

[Mae85] Mamoru Maekawa. A $\sqrt{N}$ algorithm for mutual exclusion in decentralized systems. *ACM Trans. Comput. Syst.*, 3 :145–159, May 1985.

[Mar85] Alain J. Martin. Distributed mutual exclusion on a ring of processes. *Sci. Comput. Program.*, 5(3) :265–276, 1985.

[MBB+92] Kia Makki, Paul Banta, Ken Been, Niki Pissinou, and E. K. Park. A token based distributed k mutual exclusion algorithm. In *SPDP*, pages 408–411, 1992.

[MN06] Quazi Ehsanul Kabir Mamun and Hidenori Nakazato. A new token based protocol for group mutual exclusion in distributed systems. In *Parallel and Distributed Computing, 2006. ISPDC'06. The Fifth International Symposium on*, pages 34–41. IEEE, 2006.

[Mue98] Frank Mueller. Prioritized token-based mutual exclusion for distributed systems. In *IPPS/SPDP*, pages 791–795, 1998.

[Mue99] F. Mueller. Priority inheritance and ceilings for distributed mutual exclusion. In *Real-Time Systems Symposium, 1999. Proceedings. The 20th IEEE*, pages 340 –349, 1999.

[MyC] Mycloud project. "`http://mycloud.inrialpes.fr/`".

[Nai93] Mohamed Naimi. Distributed algorithm for k-entries to critical section based on the directed graphs. *SIGOPS Oper. Syst. Rev.*, 27(4) :67–75, October 1993.

[NM91] Mitchell L. Neilsen and Masaaki Mizuno. A dag-based algorithm for distributed mutual exclusion. In *ICDCS*, pages 354–360, 1991.

[NT87a] Mohamed Naimi and Michel Trehel. How to detect a failure and regenerate the token in the log(n) distributed algorithm for mutual exclusion. In *WDAG*, pages 155–166, 1987.

[NT87b] Mohamed Naimi and Michel Trehel. An improvement of the log(n) distributed algorithm for mutual exclusion. In *ICDCS*, pages 371–377, 1987.

[NTA96] Mohamed Naimi, Michel Trehel, and André Arnold. A log(n) distributed mutual exclusion algorithm based on path reversal. *J. Parallel Distrib. Comput.*, 34(1) :1–13, 1996.

[OAR] Oar. "`http://oar.imag.fr/`".

[oMP] Openmpi. "`http://www.open-mpi.org/`".

[RA81] Glenn Ricart and Ashok K. Agrawala. An optimal algorithm for mutual exclusion in computer networks. *Commun. ACM*, 24 :9–17, January 1981.

[Ray89a] Kerry Raymond. A distributed algorithm for multiple entries to a critical section. *Inf. Process. Lett.*, 30(4) :189–193, 1989.

[Ray89b] Kerry Raymond. A tree-based algorithm for distributed mutual exclusion. *ACM Trans. Comput. Syst.*, 7(1) :61–77, 1989.

[Ray91a] Michel Raynal. A distributed solution to the k-out of-m resources allocation problem. In *ICCI*, pages 599–609, 1991.

[Ray91b] Michel Raynal. A simple taxonomy for distributed mutual exclusion algorithms. *Operating Systems Review*, 25(2) :47–50, 1991.

[Ray92] Michel Raynal. *Synchronisation et état global dans les systèmes répartis*. Eyrolles, 1992.

[RCL09] Bhaskar Prasad Rimal, Eunmi Choi, and Ian Lumb. A taxonomy and survey of cloud computing systems. In *Proceedings of the 2009 Fifth International Joint Conference on INC, IMS and IDC*, NCM '09, pages 44–51, Washington, DC, USA, 2009. IEEE Computer Society.

[Rhe95] Injong Rhee. A fast distributed modular algorithm for resource allocation. In *ICDCS*, pages 161–168, 1995.

[Rhe98] Injong Rhee. A modular algorithm for resource allocation. *Distributed Computing*, 11(3) :157–168, 1998.

[RMG08] Vijay Anand Reddy, Prateek Mittal, and Indranil Gupta. Fair k mutual exclusion algorithm for peer to peer systems. In *ICDCS*, pages 655–662, 2008.

[S3] Amazon s3. "`http://aws.amazon.com/s3/`".

[San87] Beverly A. Sanders. The information structure of distributed mutual exclusion algorithms. *ACM Trans. Comput. Syst.*, 5(3) :284–299, August 1987.

[Sin92] Mukesh Singhal. A dynamic information-structure mutual exclusion algorithm for distributed systems. *IEEE Trans. Parallel Distrib. Syst.*, 3(1) :121–125, 1992.

[Sin93] Mukesh Singhal. A taxonomy of distributed mutual exclusion. *J. Parallel Distrib. Comput.*, 18(1) :94–101, 1993.

[SK85] Ichiro Suzuki and Tadao Kasami. A distributed mutual exclusion algorithm. *ACM Trans. Comput. Syst.*, 3(4) :344–349, 1985.

[SLAAS07] Julien Sopena, Fabrice Legond-Aubry, Luciana Bezerra Arantes, and Pierre Sens. A composition approach to mutual exclusion algorithms for grid applications. In *ICPP*, page 65, 2007.

[SM94] R. Satyanarayanan and C. R. Muthukrishnan. Multiple instance resource allocation in distributed computing systems. *J. Parallel Distrib. Comput.*, 23(1) :94–100, 1994.

[Sop08] Julien Sopena. *Algorithmes d'exclusion mutuelle : tolerance aux fautes et adaptation aux grilles.* PhD thesis, Université Pierre et Marie Curie, 2008.

[SP88] Eugene Styer and Gary L. Peterson. Improved algorithms for distributed resource allocation. In *PODC*, pages 105–116, 1988.

[SR92] Pradip K. Srimani and Rachamallu L. N. Reddy. Another distributed algorithm for multiple entries to a critical section. *Inf. Process. Lett.*, 41(1) :51–57, 1992.

[SRL90] Lui Sha, Ragunathan Rajkumar, and John P. Lehoczky. Priority inheritance protocols : An approach to real-time synchronization. *IEEE Trans. Computers*, 39(9) :1175–1185, 1990.

[SS07] Abhishek Swaroop and Awadhesh Kumar Singh. A distributed group mutual exclusion algorithm for soft real time systems. In *in Proc. WASET International Conference on Computer, Electrical and System Science and Engineering CESSE07*, pages 138–143, 2007.

[Vel93] Martin G. Velazquez. A survey of distributed mutual exclusion algorithms. Technical report, Colorado state university, 1993.

[Vid03] K. Vidyasankar. A simple group mutual l-exclusion algorithm. *Information Processing Letters*, 85(2) :79 – 85, 2003.

[WJ00] K-P Wu and Y-J Joung. Asynchronous group mutual exclusion in ring networks. *IEE Proceedings-Computers and Digital Techniques*, 147(1) :1–8, 2000.

[WPP91] Elizabeth B. Weidman, Ivor P. Page, and William J. Pervin. Explicit dynamic exclusion algorithm. In *SPDP*, pages 142–149, 1991.

[ZCB10] Qi Zhang, Lu Cheng, and Raouf Boutaba. Cloud computing : state-of-the-art and research challenges. *J. Internet Services and Applications*, 1(1) :7–18, 2010.

Publications associées à cette thèse

Revue internationale

En cours de soumission

[JPDC14] Jonathan Lejeune, Luciana Arantes, Julien Sopena, and Pierre Sens. A Fair Starvation-free Prioritized Mutual Exclusion Algorithm for Distributed Systems In Journal of Parallel and Distributed Computing.

Conférences internationales

[ICPP13] Jonathan Lejeune, Luciana Arantes, Julien Sopena, and Pierre Sens. A prioritized distributed mutual exclusion algorithm balancing priority inversions and response time In International Conference on Parallel Processing (ICPP-2013).

[CCgrid13] Damián Serrano, Sara Bouchenak, Yousri Kouki, Thomas Ledoux, Jonathan Lejeune, Julien Sopena, Luciana Arantes, and Pierre Sens. Towards QoS-Oriented SLA Guarantees for Online Cloud Services. In *13th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing (CC-GRID'13)*. IEEE Computer Society Press, May 2013.

[CCgrid12] Jonathan Lejeune, Luciana Arantes, Julien Sopena, and Pierre Sens. Service level agreement for distributed mutual exclusion in cloud computing. In *12th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing (CCGRID'12)*. IEEE Computer Society Press, May 2012.

Conférence francophone

[Compas14] Jonathan Lejeune, Luciana Arantes, Julien Sopena, and Pierre Sens. Un algorithme distribué efficace d'exclusion mutuelle généralisée sans connaissance préalable des conflits. In Conférence d'informatique en Parallélisme, Architecture et Système (Renpar), April 2014, (Meilleur article Renpar).

[Compas13] Jonathan Lejeune, Luciana Arantes, Julien Sopena, and Pierre Sens. Un algorithme équitable d'exclusion mutuelle distribuée avec priorité. In *9ème*

Conférence Française sur les Systèmes d'Exploitation (CFSE'13), Chapitre français de l'ACM-SIGOPS, GDR ARP, January 2013.

Liste des notations

N	Nombre de processus du système, i.e. le cardinal de Π	8
Π	Ensemble des processus du système	8
s_i	$1 \leq i \leq N$, site appartenant à Π	9
T	Temps d'exécution du système.	9
t_0	Instant appartenant à T correspondant à l'initialisation du système	9
t_{now}	Instant courant appartenant à T	9
P	nombre de priorités dans le système	26
p_{max}	priorité maximale du système	26
p_{min}	priorité minimale du système	26
$\mathcal{P}$	ensemble totalement ordonné des priorités	26
$\mathcal{F}$(p)	fonction de palier permettant de retarder l'incrémentation de priorité	39
ρ	charge : exprime la fréquence à laquelle la section critique est demandée	40
θ	temps d'une expérience d'évaluation	40
α	le temps d'une section critique	40
γ	le temps d'acheminement d'un message entre deux processus voisins	40
β	le temps où un site reste à l'état *tranquil*	40
$NbSLA$	le nombre de niveaux de SLA considérés.	82
ψ	taille de préemption maximum.	79
$WaitMin$	temps de réponse requis minimal (correspond au plus haut niveau de SLA).	82
$\mathcal{A}$	Fonction calculant un réel à partir d'un vecteur de M entiers	104
$D_{s_i}^t$	Ensemble des ressources $\in \mathcal{R}$ que le site s_i demande à l'instant t	92
G	Graphe de conflits	94
M	Nombre de types de ressources dans le système	92
$\mathcal{R}$	Ensemble des ressources du système $\{r_1, r_2, ..., r_M\}$	92

Annexe A

Implémentation distribuée de l'algorithme d'exclusion mutuelle généralisée

Sommaire

A.1 Généralités **129**
A.2 Les messages **130**
A.2.1 Information véhiculée 130
A.2.2 Mécanisme d'agrégation 130
A.3 États des processus **131**
A.4 Variables locales **131**
A.5 Description **132**

Cette annexe décrit l'implémentation distribuée de l'algorithme de verrouillage de ressources basé sur des compteurs (chapitre 7). Sur le même principe que l'algorithme de Bouabdallah-Laforest, notre implémentation distribuée se base sur la transmission de jetons. Son pseudo-code est donné en figures A.3, A.4 et A.5.

A.1 Généralités

Chaque ressource est associée à un unique jeton. Les compteurs sont stockés dans le jeton de la ressource associée. Ainsi le possesseur du jeton est le seul à pouvoir accéder à la valeur du compteur assurant ainsi un accès exclusif. Chaque jeton est géré par une instance d'une version simplifiée de l'algorithme à priorité de Mueller (cf section 3.4.1). Par conséquent, les messages de requêtes pour une ressource (ou pour un compteur) sont acheminés jusqu'au porteur de jeton grâce à une structure d'arbre dynamique. Le choix d'un algorithme à priorité permet de réordonnancer de manière dynamique la file d'attente d'un jeton si une requête de précédence plus importante, i.e., avec un vecteur plus prioritaire, est émise. Ainsi plus la valeur résultante de $\mathcal{A}$ est basse plus la priorité de la requête est haute.

A.2 Les messages

A.2.1 Information véhiculée

Nous définissons quatre types de messages : $Request1$, $Counter$, $Request2$ et $Token$. Chaque type de message véhicule une ou plusieurs structures de données qui lui est propre (déclarations données en figure A.1). Ces structures de données sont :

- $Request1$: demande de la valeur du compteur associé à r. Elle contient l'identifiant du site initiateur, l'identifiant de la requête concernée et un booléen drapeau qui vaut vrai si la requête concerne une seule ressource, faux sinon.
- $Counter$: indication de la valeur du compteur associé à r. Cette émission fait suite à une réception d'un message de type $Request1$ sur le porteur du jeton correspondants.
- $Request2$: demande de la ressource r. Elle contient l'identifiant du site initiateur, l'identifiant de la requête concernée et un réel égal au résultat de $\mathcal{A}$ à partir du vecteur originel.
- $Token$: jeton de la ressource r. Il stocke la valeur du compteur associé, une liste de requêtes de type 2 à satisfaire qui représente la file d'attente de la ressource r et deux tableaux de N identifiants de requêtes :
  - $lastReq1$: indique pour chaque site s_j l'identifiant de la dernière requête de s_j auquel le porteur du jeton a envoyé la valeur du compteur.
  - $lastCS$: pour chaque site s_j l'identifiant de la dernière requête de s_j satisfaite par ce jeton.

A.2.2 Mécanisme d'agrégation

Afin d'économiser des envois de messages, chaque message peut véhiculer plusieurs structures qui lui correspondent. En effet, les structures du même type adressées au même destinataire seront agrégées dans un seul message. Par conséquent, la réception d'un message concerne non pas une seule ressource mais un ensemble de ressources. Ce mécanisme implique donc des tampons locaux : un par type de message et par destinataire.

Nous définissons la fonction **buffer(**site s_{dest}, type t, data**)** qui stocke temporairement la structure de type t à destination du site s_{dest}.

Nous définissons alors pour chaque type de message une fonction qui consomme les structures stockées par la fonction **buffer** en les envoyant à leur destinataire :

- **SendBufReq1(** *visited* **: set of sites)** : envoie les structures de type $Request1$ à leur destinataire. Comme ce type de message n'est pas directement transmis au destinataire final, i.e. le porteur du jeton, il stocke au fur et à mesure de son transfert les nœuds qu'il a déjà visités afin d'éviter les cycles de messages dans la structure dynamique.
- **SendBufCnt()** : envoie les structures de type $Counter$ à leur destinataire.
- **SendBufReq2(** *visited* **: set of sites)** : même principe que le type $Request1$.
- **SendBufTok()** : envoie les structures de type $Token$ à leur destinataire.

```
1  Type Request1 :
2  begin
3  |  s_init : site;
4  |  r : resource;
5  |  id : integer ;
6  |  isOne : boolean ;
7  end
```

```
8  Type Counter :
9  begin
10 |  r : resource;
11 |  val : integer ;
12 end
```

```
13 Type Request2 :
14 begin
15 |  s_init : site;
16 |  r : resource;
17 |  id : integer ;
18 |  mark : float;
19 end
```

```
20 Type Token :
21 begin
22 |  r : resource;
23 |  counter : integer;
24 |  lastReq1 : array of
   |  N integers;
25 |  lastCS : array of N
   |  integers;
26 |  req2 : sorted list of
   |  Request2 ;
27 end
```

Figure A.1 – Structures véhiculées par les messages de l'implémentation distribuée

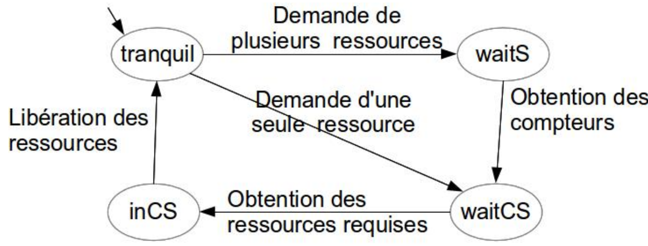


Figure A.2 – Machine à états des processus

A.3 États des processus

L'état *requesting* a été divisé en deux sous-états : *waitS* et *waitCS*. Les processus ont donc quatre états possibles :

- *tranquil* : le processus ne demande rien.
- *waitS* : le processus attend les compteurs requis (première étape).
- *waitCS* : le processus attend les ressources requises (seconde étape).
- *inCS* : le processus utilise les ressources requises (section critique).

La figure A.2 représente la machine à états des processus.

Nous avons ajouté une optimisation permettant aux requêtes ne demandant qu'une seule ressource de passer directement de l'état *tranquil* à l'état *waitCS* (lignes 67 à 70). En effet, ces requêtes ne demandent qu'un seul compteur. Puisque cet unique compteur est stocké dans le jeton, le nœud racine peut calculer le résultat de $\mathcal{A}$ et ainsi transformer directement la requête de type 1 en requête de type 2, la stocker dans la file d'attente du jeton et éventuellement renvoyer le jeton si la nouvelle requête est plus prioritaire. Ce mécanisme permet de réduire le coût de synchronisation des requêtes demandant une seule ressource.

A.4 Variables locales

Chaque processus maintient les variables locales suivantes :

- *tokDir* : un tableau de M sites, où chaque entrée indique le père dans l'arbre dynamique gérant la ressource correspondante. L'entrée vaut *nil* si le processus est

la racine de l'arbre correspondant (possession du jeton).
- $MyVector$: le vecteur correspondant à la requête en cours du site courant.
- $lastToken$: tableau de M structures de jetons. Cette variable permet de stocker localement les informations d'un jeton lors de sa dernière réception.
- $TRequired$: ensemble de ressources que le site demande.
- $TOwned$: ensemble de ressources que le site a en sa possession.
- $CntNeeded$: ensemble de ressources requises dont le processus n'a pas encore reçu le compteur associé.
- $curId$: compteur local permettant d'identifier et de dater une requête. Sa valeur s'incrémente à chaque nouvelle requête (ligne 65).
- $pendingReq1$ et $pendingReq2$: tableaux de M ensembles de requêtes de type 1 (respectivement de type 2) reçues pour chaque ressource. Ces ensembles sont utiles pour éviter que des requêtes ne soient perdues lorsque le jeton est en transit dans le réseau. Puisque les structures de requête contiennent la date locale de la requête (champs id), il est possible de sauvegarder uniquement la requête la plus récente pour un site donné. Si le processus possède une ressource, les entrées correspondantes sont égales à l'ensemble vide.

A.5 Description

Lors de l'appel à $Request_CS$ (ligne 63), le processus demande les compteurs concernés (ligne 71). S'il possède des jetons requis alors il sauvegarde dans son vecteur la valeur du compteur et incrémente ce dernier (lignes 76 à 78). Sinon, il sauvegarde dans $CntNeeded$ les ressources pour lesquels une requête de type 1 a été envoyée (lignes 72 à 75). Si la requête concerne une seule ressource, il est inutile de sauvegarder la ressource dans $CntNeeded$ puisque ces requêtes ne recquièrent pas d'attendre un compteur. Une fois les requêtes de type 1 envoyées (ligne 79), le processus se met en attente jusqu'à ce que l'ensemble des ressources requises soient possédé (lignes 80 et 81).

À la réception d'un message de requêtes de type 1 (ligne 97), le site traite l'ensemble de $Request1$ contenu dans le message. Si une requête est obsolète (ligne 102) alors elle est ignorée. Si le site ne possède pas la ressource concernée, la requête est ajoutée dans l'ensemble $pendingReq1$ et est envoyée au père dans l'arbre correspondant (lignes 132 et 133) en s'assurant que ce dernier n'a pas déjà transmis la requête (ligne 131). Si au contraire, le site possède la ressource (il est le site racine de l'arbre correspondant), l'identifiant de la requête est sauvegardé dans le tableau $lastReq1$ du jeton (ligne 105). Trois cas peuvent alors se produire :
- **le site n'a pas besoin de la ressource** (ligne 106) : le jeton est directement envoyé au site demandeur (ligne 107)
- **le site a besoin de la ressource et la requête concerne plusieurs ressources** (ligne 108) : la valeur du compteur est envoyée au demandeur et incrémentée (lignes 110 et 111).
- **le site a besoin de la ressource et la requête concerne une seule ressource** (ligne 112) : la requête est considérée comme une requête de type 2. Sa valeur réelle peut alors être calculée avec la fonction $\mathcal{A}$ et la valeur du compteur (lignes 113 à 118

```
28 Local variables :
29 begin
30     state ∈ {tranquil, waitS, waitCS, inCS};
31     tokDir : array of M sites;
32     MyVector : array of M integers;
33     lastTok : array of M Token;
34     TRequired : set of resources;
35     TOwned : set of resources;
36     CntNeeded : set of resources;
37     curId : integer ;
38     pendingReq1 : array of M sets of Request1;
39     pendingReq2 : array of M sorted lists of Request2;
40 end
41 Initialization
42 begin
43     if self = elected_node then
44         tokDir[r] ← nil ∀r ∈ R;
45         TOwned ← R ;
46     else
47         tokDir[r] ← elected_node ∀r ∈ R;
48         TOwned ← ∅ ;
49     TRequired ← ∅;
50     CntNeeded ← ∅;
51     state ← tranquil;
52     curId ← 0;
53     foreach resource r ∈ R do
54         MyVector[r] ← 0;
55         lastTok[r].r ← r;
56         lastTok[r].counter ← 1;
57         lastTok[r].lastReq[s] ← 0 ∀s ∈ Π;
58         lastTok[r].lastCS[s] ← 0 ∀s ∈ Π;
59         lastTok[r].req2 ← ∅;
60         pendingReq1[r] ← ∅;
61         pendingReq2[r] ← ∅;
62 end
```

```
63 Request_CS(D : set of resources)
64 begin
65     curId ← curId + 1;
66     TRequired ← D;
67     if |TRequired| = 1 then
68         state ← waitCS;
69     else
70         state ← waitS;
71     foreach resource r ∈ TRequired do
72         if tokDir[r] ≠ nil then
73             if |TRequired| ≠ 1 then
74                 CntNeeded ← CntNeeded ∪ {r};
75             buffer(tokDir[r],Request1,
               < self, r, curId, |TRequired| = 1 > );
76         else
77             MyVector[r] ← lastTok[r].counter;
78             lastTok[r].counter ←
               lastTok[r].counter + 1;
79     SendBufReq1({self});
80     if TRequired ⊈ TOwned then
81         wait(TRequired ⊆ TOwned);
82     state ← inCS;
83     /* CRITICAL SECTION                                  */
84 end
85 Release_CS
86 begin
87     state ← tranquil;
88     foreach ressource r ∈ TRequired do
89         lastTok[r].lastCS[self] ← curId;
90         if lastTok[r].req2 ≠ ∅ then
91             < s, r', seq, m >←
               dequeue(lastTok[r].req2);
92             SendToken(s,r);
93     TRequired ← ∅;
94     MyVector[r] ← 0 ∀r ∈ R;
95     SendBufTok();
96 end
```

FIGURE A.3 – Implémentation distribuée : procédures d'initialisation, de demande et de libération de section critique

). Si le site est en attente de compteurs (état *waitS*), alors la requête en question est prioritaire : le jeton est alors envoyé (ligne 120). Sinon si le site est en attente de jetons (état *waitCS*), sa requête et la requête reçue sont alors comparées (ligne 123). Si la requête distante est plus prioritaire, alors le site ajoute sa propre requête dans la file locale du jeton et le transmet au site prioritaire (lignes 124 et 125). Dans le cas contraire si le site courant est en section critique, c'est la requête distante qui est ajoutée dans la file d'attente du jeton (lignes 127 et 130).

À la réception d'un message de compteur (ligne 138), la valeur du compteur est sauvegardée dans l'entrée correspondante du vecteur (ligne 141) et l'ensemble des compteurs requis est mis à jour (ligne 142). Comme le site envoyeur est le porteur du jeton le plus récent du point de vue du site courant, le pointeur de parenté est alors mis à jour (ligne 143) : des messages de requêtes peuvent alors être économisés. Si le site n'attend plus de compteur (ligne 144) la fonction **processCntNeededEmpty** est appelée. Cette fonction fait passer le site courant à l'état *waitCS*. Le site peut ensuite envoyer pour toute ressource requise non possédée, un message de requête de type 2 (ligne 181).

La réception un message de requêtes de type 2 (ligne 147), implique dans un premier temps une vérification de l'obsolescence de la requête (ligne 152). Si le site possède la ressource demandée (lignes 155 à 164) alors le traitement est le même que la réception d'une requête de type 1 concernant une seule ressource . Si le site ne possède pas la ressource et si la requête n'a pas déjà visité le processus père du site courant (ligne 165) alors la requête est enregistrée dans la file locale *pendingReq*2. Le test de la ligne 167 permet de décider si le message nécessite d'être retransmis au père ou non (principe de l'algorithme de Mueller). Si le site courant et le site demandeur sont en conflit sur la ressource en question et s'ils sont tous les deux en état *waitCS* alors il est possible de différencier les deux requêtes. Si la requête du site courant est plus prioritaire que le site distant alors il est inutile de retransmettre le message car le site courant obtiendra le jeton avant le site distant. Il subsiste néanmoins une exception où la requête sera retransmise : la requête du site courant concerne une seule ressource. En effet, puisque dans ce cas le site courant ne connaît pas le réel de sa requête, il est donc impossible pour lui de la différencier de celle du site distant.

À la réception d'un message de jeton (ligne 184) le processus va dans en premier temps mettre à jour ses variables locales pour chaque jeton contenu dans le message en appelant la fonction *processUpdate* (ligne 188). Cette fonction met à jour dans un premier temps les variables *TOwned*, *tokDir* puis sauvegarde d'éventuels compteurs manquants (lignes 226 à 229). Ensuite on traite les requêtes de type 1 enregistrées dans *pendingReq*1 et qui n'ont pas été prises en compte (lignes 230 à 249)). Le traitement est similaire à la réception d'une requête de type 1 lorsque le site possède la ressource concernée. Chaque requête de type 2 enregistrée dans *pendingReq*2 est ensuite copiée dans la file d'attente du jeton si elle n'a pas encore été prise en compte (lignes 251 à 257). Une fois les variables locales mises à jour, le site contrôle s'il peut entrer en section critique (lignes 189 à 191). S'il ne peut pas encore entrer en section critique, il contrôle d'abord si l'ensemble des compteurs manquants ont été acquis dans la fonction *processUpdate* et le cas échéant appelle la fonction *processCntEmpty* permettant de passer à l'état *waitCS* et d'envoyer les messages de requête de type 2 (lignes 177 à 181). Il contrôle ensuite si le

```
97  Receive request1 (visitedNodes : set of sites,
    Req1sRecv : set of Request1) from s_j
98  begin
99      foreach Request1 req1 ∈ Req1sRecv do
100         ressource r ← req1.r;
101         site s_i ← req1.s_init;
102         if req1.id ≤ lastTok[r].lastReq1[s_i] or
            req1.id ≤ lastTok[r].lastCS[s_i] then
103             continue;
104         if r ∈ TOwned then
105             lastTok[r].lastReq1[s_i] ← req1.numSeq;
106             if r ∉ TRequired then
107                 SendToken(s_i, r);
108             else if ¬req1.isOne then
109                 /* send Counter                         */
110                 buffer(s_i,Counter,<
                    r, lastTok[r].counter > );
111                 lastTok[r].counter ←
                    lastTok[r].counter + 1;
112             else
113                 Vtmp : array of M integer;
114                 Vtmp[r_k] ← 0 ∀r_k ∈ R;
115                 Vtmp[r] ← lastTok[r].counter;
116                 lastTok[r].counter ←
                    lastTok[r].counter + 1;
117                 float mark ← A(Vtmp);
118                 Request2 newReq ←< s_i, r, seq, mark >;
119                 if state = waitS then
120                     SendToken(s_i, r);
121                 else if state = waitCS then
122                     Request2 myReq ←<
                        self, r, curId, A(MyVector) > ;
123                     if newReq ◁ myReq then
124                         add(lastTok[r].req2,myReq);
125                         SendToken(s_i, r);
126                     else
127                         add(lastTok[r].req2,newReq);
128                 else
129                     /* inCS                             */
130                     add(lastTok[r].req2,newReq);
131         else if tokDir[r] ∉ visitedNodes then
132             add(pendingReq1[r],req1);
133             buffer(tokDir[r],Request1, req1 );
134     SendBufReq1(visitedNodes ∪ {self});
135     SendBufTok();
136     SendBufCnt();
137 end

138 Receive Counter(CntsRcv : sets of Counter) from s_j
139 begin
140     foreach Counter cnt ∈ CntsRcv do
141         MyVector[cnt.r] ← cnt.val;
142         CntNeeded ← CntNeeded − {cnt.r};
143         tokDir[r] ← s_j;
144     if CntNeeded = ∅ then
145         processCntNeededEmpty();
146 end
```

```
147 Receive Request2(visitedNodes : set of sites,
    Req2sRcv : sets of Request2) from s_j
148 begin
149     foreach Request2 req2 ∈ Req2sRecv do
150         ressource r ← req2.r;
151         site s_i ← req2.s_init;
152         if req2.id < lastTok[r].lastReq1[s_i] or
            req2.id ≤ lastTok[r].lastCS[s_i] then
153             continue;
154         Request2
            myReq2 ←< self, r, curId, A(MyVector) >;
155         if r ∈ TOwned then
156             if r ∉ TRequired or state = waitS
                then
157                 SendToken(s_i, r);
158             else if req2 ∉ lastTok[r].req2 then
159                 if state = waitCS ∧ req2 ◁ myReq2
                    then
160                     add(lastTok[r].req2,myReq2);
161                     SendToken(s_i, r);
162                 else
163                     /* (waitCS ∧ myReq2 ◁
                           req2) ∨ inCS                 */
164                     add(lastTok[r].req2,req2);
165         else if tokDir[r] ∉ visitedNodes then
166             add(pendingReq2[r],req2);
167             if state = waitCS ∧ r ∈ TRequired ∧
                myReq2 ◁ req2 ∧ |TRequired| ≠ 1 then
168                 /* Do not forward                       */
169             else
170                 buffer(tokDir[r],Request2, req2
                    );
171     SendBufReq2(visitedNodes ∪ {self});
172     SendBufTok();
173 end

174 processCntNeededEmpty()
175 begin
176     /* precond : state = waitS ∧ CntNeeded = ∅ */
177     state ← waitCS;
178     foreach resource r ∈ TRequired do
179         Request2
            myReq2 ←< self, r, curId, A(MyVector) >;
180         if r ∉ TOwned then
181             buffer(tokDir[r],Request2, myReq2 );
182     SendBufReq2({self});
183 end
```

FIGURE A.4 – Implémentation distribuée : procédures de réception de requêtes 1 et 2 et de compteur

```
184 Receive Token (TokˢRcv : sets of Token) from s_j
185 begin
186     /* t.r must be in TRequired                                   */
187     foreach Token t ∈ ToksRcv do
188         processUpdate(t);
189     if TRequired ⊆ TOwned then
190         state ← inCS;
191         notify(TRequired ⊆ TOwned);
192     else
193         if state = waitS ∧ CntNeeded = ∅ then
194             processCntNeededEmpty();
195         foreach resource r ∈ TOwned do
196             if lastTok[r].req2 ≠ ∅ then
197                 Request2 req2 ← Head(lastTok[r].req2);
198                 site s_i ← req2.s_init;
199                 if state = waitS then
200                     dequeue(lastTok[r].req2);
201                     SendToken(s_i, r);
202                 else if state = waitCS then
203                     Request2 myReq2 ←<
                        self, r, curId, A(MyVector) >;
204                     if req2 ◁ myReq then
205                         dequeue(lastTok[r].req2);
206                         add(lastTok[r].req2,myReq2);
207                         SendToken(s_i, r);
208                 else
209                     /* IMPOSSIBLE                                 */
210     SendBufCounters();
211     SendBufTokens();
212 end

213 SendToken(s_dest : site, r : resource)
214 begin
215     /* precond : r∈TOwned                                         */
216     buffer(s_dest,Token, lastTok[r] );
217     tokDir[r] ← s_dest;
218     TOwned ← TOwned − {r};
219 end
```

```
220 processUpdate(t : Token)
221 begin
222     ressource r ← t.r;
223     lastTok[r] ← t;
224     TOwned ← TOwned ∪ {r};
225     tokDir[t.r] ← nil;
226     if r ∈ CntNeeded then
227         CntNeeded ← CntNeeded − {r};
228         MyVector[r] ← lastTok[r].counter;
229         lastTok[r].counter ← lastTok[r].counter + 1;
230     foreach Request1 req1 ∈ pendingReq1[r] do
231         site s_i ← req1.s_init;
232         if req1.id ≤ lastTok[r].lastReq1[s_i] or
            req1.id ≤ lastTok[r].lastCS[s_i] then
233             /* req1 is out of date                        */
234             continue;
235         lastTok[r].lastReq1[s_i] ← req1.id;
236         if pendingReq1[r].isOne then
237             /* transform it in req2                       */
238             Vtmp : array of M integer;
239             Vtmp[r_k] ← 0 ∀r_k ∈ R;
240             Vtmp[r] ← lastTok[r].counter;
241             lastTok[r].counter ←
                lastTok[r].counter + 1;
242             float mark ← A(Vtmp);
243             Request2
                newReq2 ←< s_i, r, req1.id, mark >;
244             if newReq2 ∉ lastTok[r].req2 then
245                 add(lastTok[r].req2,newReq2);
246         else
247             /* send Counter                               */
248             buffer(s_i,lastTok[r].counter);
249             lastTok[r].counter ←
                lastTok[r].counter + 1;
250     pendingReq1[r] ← ∅;
251     foreach Request2 req2 ∈ pendingReq2[r] do
252         site s_i ← req2.s_init;
253         if req2.id < lastTok[r].lastReq1[s_i] or
            req2.id ≤ lastTok[r].lastCS[s_i] then
254             /* req2 is out of date                        */
255             continue;
256         if req2 ∉ lastTok[r].req2 then
257             add(lastTok[r].req2,req2);
258     pendingReq2[r] ← ∅;
259 end
```

Figure A.5 – Implémentation distribuée : procédure de réception et d'envoi de jeton

jeton en question doit être retransmis à un éventuel site plus prioritaire présent dans la file d'attente (lignes 195 à 207).

Enfin lorsque le site sort de la section critique (ligne 85), il repasse à l'état *tranquil* et transmet éventuellement les jetons possédés aux prochains processus demandeurs (lignes 88 à 92).