Université Pierre et Marie Curie – Paris VI
UFR 919 (Ingénierie)

Centre National de la Recherche Scientifique

Algorithmes d'exclusion mutuelle : tolérance aux fautes et adaptation aux grilles

THÈSE

soutenance prévue le 8 Décembre 2008

pour l'obtention du

Doctorat de l'Université Pierre et Marie Curie - Paris VI

Spécialité : Informatique

par

Julien Sopena

Composition du jury

Directeur :	**Pierre SENS**, Professeur à l'Université Paris VI
Encadrant :	**Luciana ARANTES**, Maître de conférences à l'Université Paris VI
Rapporteurs :	**Roberto BALDONI**, Professeur à l'Université Rome I **Frédéric DESPREZ**, Directeur de recherche à l'INRIA
Examinateurs :	**Marin BERTIER**, Maître de conférences à l'INSA de Rennes **André SCHIPER**, Professeur à l'École Polytechnique Fédérale de Lausanne **Sebastien TIXEUIL**, Professeur à l'Université Paris VI

LIP6 – Laboratoire d'Informatique de Paris VI

Mis en page avec la classe thloria.

Je dédie cette thèse
à Mathilde et Nathéo.

Table des matières

Partie I
Algorithme d'exclusion mutuelle tolérant aux défaillances

Partie II
Algorithme d'exclusion mutuelle adapté aux topologies de type grille

Chapitre 6
Un algorithme générique de composition

Chapitre 7
Évaluation de performance de notre approche générique

Chapitre 8
Conclusion

Chapitre 1

Introduction

Sommaire

"L'enfer c'est les autres" écrivait Jean Paul Sartre dans [Sar44]. Loin de faire l'apologie de la misanthropie, il exprime ici la nécessité des autres et les problèmes que cela engendre. Cette phrase résume très bien la problématique des systèmes distribués à grande échelle. Si la mutualisation des ressources disponibles devient nécessaire pour satisfaire les nouveaux besoins en puissance de calcul et en capacité de stockage, elle engendre de nouveaux problèmes inhérents à cette pluralité grandissante. Ainsi, ces systèmes se caractérisent par le grand nombre de ressources à gérer, l'hétérogénéité des réseaux d'inter-connexion et la nécessité de considérer les pannes dont la fréquence augmente proportionnellement au nombre de machines.

Cette thèse prend pour cadre ces grands systèmes distribués et y étudie le paradigme de l'*exclusion mutuelle*. Les algorithmes distribués d'exclusion mutuelle sont l'une des briques de base pour de nombreuses applications réparties. Elles les utilisent pour protéger la partie du code (*section critique*) qui accède à des ressources partagées. Ainsi, on retrouve ce mécanisme dans tous les niveaux des applications réparties. Le middleware *Globus* [GLB] offre, par exemple, une *API* d'exclusion mutuelle, le service distribué de partage de données JuxMem [ABJ05] utilise un algorithme d'exclusion mutuelle distribué pour permettre à un client de verrouiller une donnée et le micro-noyau distribué *Chorus* [MVF+92] est doté d'une synchronisation distribuée des processus [Era95].

Le paradigme de l'exclusion mutuelle a été introduit pour la première fois en 1965 par Edsger Dijkstra [Dij65] et depuis de nombreuses solutions ont été proposées pour des systèmes répartis. On distingue généralement deux classes d'algorithmes d'exclusion mutuelle distribués. Dans les approches à base de permissions, un nœud peut entrer en section critique uniquement s'il a obtenu la permission de tous les nœuds ou d'une majorité d'entre eux [Lam78, RA81, Mae85]. Dans les autres approches, un jeton unique donne le

droit d'accès à la ressource critique [SK85, Ray89, NT96, Mar85].

Mais face aux contraintes liées aux environnements à grande échelle, nombre des algorithmes de la littérature se montrent inadaptés, ou tout du moins, difficiles à mettre en œuvre. Parmi ces difficultés, nous avons choisi de nous intéresser plus particulièrement à celle de la gestion des défaillances ainsi que celle de l'adaptation aux topologies de type grille.

1.1 Problématiques

Tolérance aux défaillances : Dans cette thèse, nous nous concentrons sur les algorithmes à base de jeton car ils sont souvent moins coûteux en nombre de messages et sont intrinsèquement plus extensibles que les algorithmes à permission. En contrepartie, ils sont particulièrement sensibles aux défaillances.

Plusieurs auteurs ont défini des extensions tolérantes aux fautes d'algorithmes à jeton, mais la plupart de ces algorithmes reposent sur l'utilisation de la diffusion de messages ([NLM90]), ce qui limite leur mise en œuvre dans des systèmes à grande échelle. De plus les hypothèses faites par ces algorithmes rendent souvent difficile leurs implémentations : limitation forte du nombre de défaillances simultanées, arrêt de l'algorithme pendant la durée de la défaillance, pas de défaillance pendant le recouvrement, possesseur du jeton considéré comme sûr, jetons temporairement multiples...

Notre objectif est de proposer un algorithme d'exclusion mutuelle tolérant les défaillances et permettant une mise en œuvre sur des systèmes répartis à grande échelle. Cet algorithme devra, entre autres, limiter tant que possible l'utilisation de la diffusion de messages, tant pour le recouvrement que pour les mécanismes de détection. Il devra aussi éviter l'annulation des requêtes pendantes, permettant ainsi de limiter le coût des recouvrements tout en offrant de bonnes propriétés d'équité.

Adaptation aux topologies de type grille : Ces dernières années ont vu l'émergence des *grilles de calculs*. Ces structures informatiques de grandes tailles font coopérer un grand nombre de machines regroupées dans des grappes (*"cluster"*). Si elles permettent de répondre à moindre coût aux demandes toujours croissantes des applications, les grilles de calculs posent le problème de l'hétérogénéité des réseaux de communication : plusieurs ordres de grandeur séparent la latence et le débit internes aux clusters, de ceux de leur interconnexion. Or la plupart des algorithmes d'exclusion mutuelle de la littérature utilisent des abstractions du réseau physique et ne prennent pas en compte ces différences.

La prise en compte de cette topologie peut se faire par deux moyens. Le premier consiste à utiliser un algorithme d'exclusion mutuelle à priorité permettant de favoriser les requêtes locales par rapport aux requêtes distantes. Si cette approche permet de réduire le temps moyen d'obtention de la section critique, elle ne résout pas le problème de l'absence de service de diffusion sur le réseau d'interconnexion et de la complexité en termes de nombre de messages. La deuxième méthode consiste à assembler des algorithmes classiques d'exclusion mutuelle pour créer un nouvel algorithme hiérarchique. Les nœuds d'un même cluster sont alors rassemblés au sein d'un même groupe logique où s'exécute une instance d'un algorithme d'exclusion mutuelle. Entre ces clusters s'exécute un algorithme global

ordonnant leur accès à la section critique. Cette approche a pour intérêt de limiter le temps d'attente moyen de la section critique mais aussi de réduire la complexité en nombre de messages. L'utilisation, au sein des groupe, d'algorithmes inadaptés au système à large échelle devient dès lors possible.

Notre objectif est de proposer un algorithme permettant de composer génériquement les algorithmes présents dans la littérature. Un tel algorithme permettra d'évaluer différentes compositions sur des grilles réelles, afin de trouver la composition la mieux adaptée à la topologie physique et à l'application. En limitant l'intrusion dans les algorithmes composés, cette approche générique permettra aussi de factoriser les preuves des compositions : la preuve de l'algorithme de composition devenant à elle seule suffisante.

1.2 Contributions

Tolérance aux défaillances : Dans une première partie nous proposons un nouvel algorithme d'exclusion mutuelle tolérant les fautes. Notre algorithme est une extension de l'algorithme de Naimi-Tréhel [NT96] basé sur la circulation d'un jeton dans une file d'attente répartie et la transmission des requêtes dans un arbre dynamique. Nous avons choisi d'étendre cet algorithme car il présente de très bonnes performances en absence de faute. Par rapport aux autres approches, nous visons, d'une part, à minimiser le nombre de diffusions lors des recouvrements et, d'autre part, à conserver l'équité des requêtes en présence de fautes. En effet la plupart des algorithmes existants ne préservent pas l'ordre de la file des requêtes après un recouvrement de fautes. Ainsi, tous les nœuds attendant d'entrer en section critique sont dans l'obligation de ré-émettre leur requête, ce qui engendre un important surcoût en messages. Notre algorithme répare la file répartie en ré-assemblant dans l'ordre les morceaux encore intacts. Cette approche permet de minimiser les ré-émissions et de préserver l'équité en dépit des fautes. Dans certains cas, le temps du recouvrement peut être complètement masqué grâce à la file d'attente encore active. Par ailleurs, en l'absence de faute, notre algorithme est en $\mathcal{O}(log(N))$, ce qui lui permet de passer à l'échelle [SABS05].

Nous avons ensuite implémenté une plate-forme générique répartie *G-Mutex* permettant de tester puis d'évaluer les algorithmes d'exclusion mutuelle. Après avoir implémenté notre algorithme dans *G-Mutex*, nous avons conduit des expériences comparant notre extension et celle proposée par Naimi-Tréhel [NT87b] en environnement réel. Dans la plupart des cas, notre algorithme s'est montré à la fois plus rapide et moins coûteux en messages. Nous avons aussi observé que le comportement de notre extension n'était pas dépendant du degré de parallélisme de l'application. Ceci est particulièrement appréciable pour des applications dont le degré de parallélisme n'est pas constant. Enfin, nous avons montré que notre algorithme résistait à l'usage de temporisateurs agressifs, permettant ainsi d'optimiser le temps de recouvrement [SAS06a].

Adaptation aux topologies de type grilles : Nous nous sommes aussi intéressés à adapter les algorithmes d'exclusion mutuelle à l'hétérogénéité des réseaux des environnements de type grille. Nous avons par proposé un nouvel algorithme *générique* de composition. Cet algorithme permet de composer plusieurs algorithmes d'exclusion mutuelle,

sans les modifier, pour former une nouvelle solution *hiérarchique* [SLAS07].

Nous avons implémenté et testé cette solution en grandeur nature sur 9 clusters de la grille française GRID'5000. Les résultats de ces expériences ont permis de vérifier qu'une approche hiérarchique était très efficace. Ils ont aussi mis en évidence l'importance du taux de parallélisme de l'application dans le choix des algorithmes à composer. Ainsi nous proposons trois types de composition adaptés à différents degrés de parallélisme des applications.

En émulant de multiples topologies, nous avons également étudié le comportement de notre approche hiérarchique suivant la répartition des nœuds dans la grille. Nous montrons que le gain apporté par la composition était optimal pour une grille de N machines réparties en $\sqrt{N}$ clusters (avec une répartition uniforme). Enfin, une étude théorique a permis de montrer que ces résultats obtenus avec une répartition homogène représentaient une limite rapidement atteinte sitôt qu'on s'écarte du cas extrême d'une grille composée d'un cluster et de plusieurs machines individuelles [SALS08, SALS09].

Parallèlement à ces résultats, nous avons modélisé notre algorithme de composition en utilisant le formalisme des Réseaux de Petri Bien-Formés Stochastiques (RdPBFS) [CDFH93]. À partir de ce modèle, nous avons vérifié par des techniques de *model checking* l'expression des propriétés de l'exclusion mutuelle en *Logique Temporelle Linéaire* (LTL) [Pnu77]. Pour limiter l'explosion combinatoire, nous avons cherché à utiliser les propriétés de symétrie. La conservation de ces symétries dans le modèle et dans les propriétés, nous a permis d'optimiser le processus de vérification par l'utilisation d'algorithmes spécifiques de *model cheking* dite de symétrie partielle [BHI04]. Ce résultat présenté en annexe a été conduit en collaboration avec Souheib Baarir et Fabrice Legond-Aubry. S'il vient en complément d'une preuve formelle, il est une première étape vers une vérification quantitative des propriétés d'équités ainsi que des garanties offertes par la composition [BSL08, SBL09].

1.3 Organisation du manuscrit

Ce manuscrit s'articule autour de deux parties précédées d'un chapitre 2 introduisant le problème de l'exclusion mutuelle et ses propriétés.

Le chapitre 2 définit formellement le paradigme de l'exclusion mutuelle, puis le compare aux paradigmes du consensus et de l'élection, eux aussi liés au concept d'unicité. Enfin, nous donnons une taxonomie des algorithmes répartis d'exclusion mutuelle.

La première partie adresse le problème de la tolérance aux défaillances dans les algorithmes d'exclusion mutuelle :

- Le chapitre 3 présente les différents types de modèle pour les systèmes répartis. Nous y étudions les algorithmes à jeton tolérant les défaillances et les systèmes qu'ils considèrent. Nous nous attachons alors à dégager dans une synthèse les différentes classes d'hypothèses faites par ces algorithmes.
- Le chapitre 4 introduit l'algorithme de Naimi-Tréhel. Nous y présentons ensuite son extension tolérante aux défaillances proposée par les auteurs et analysons ses limites. Enfin, nous présentons notre extension et la comparons théoriquement à celle de Naimi-Tréhel.

– Le chapitre 5 est consacré à une étude comparative des performances de notre algorithme avec celles de l'algorithme de Naimi-Tréhel. Dans un premier temps, nous y analysons le comportement des deux approches en fonction du degré de parallélisme des applications. Puis, nous étudions l'impact du dimensionnement des temporisateurs sur les performances.

Dans une deuxième partie nous présentons notre approche pour l'adaptation des algorithme d'exclusion mutuelle aux topologies des grilles de calcul :

– Le chapitre 6 commence par une présentation de l'impact des grilles sur les algorithmes d'exclusion mutuelle. Nous y étudions ensuite les algorithmes de la littérature qui permettent une mise en œuvre dans ces topologies. Enfin, nous y présentons notre approche par composition générique.
– Le chapitre 7 présente une évaluation de notre algorithme dans la grille académique française *GRID'5000*. Nous y comparons l'efficacité de différentes topologies suivant le degré de parallélisme des applications. Nous étudions ensuite l'impact de la répartition des nœuds dans la grille. Puis, nous faisons une étude théorique des gains obtenus grâce à la composition par rapport à différentes topologies de grille. Enfin, nous proposons trois extensions pour prendre en compte les variations dans la fréquence des demandes d'entrée en section critique.

Nous terminons par une conclusion générale qui fait la synthèse de l'ensemble de nos contributions et qui met notre travail en perspective.

L'annexe *A* présente la modélisation de notre algorithme de composition en RdPBFS ainsi que celle de différentes abstractions des algorithmes d'exclusion mutuelle. Nous exprimons les propriétés de l'exclusion mutuelle à vérifier en LTL et terminons par la présentation des résultats du *model checking*.

Chapitre 2

L'exclusion mutuelle

Sommaire

L'exclusion mutuelle est une brique de base de l'algorithmique répartie assurant qu'à un instant donné : seul un processus puisse exécuter une partie d'un code concurrent, appelé section critique. Ce chapitre a pour but de présenter et d'analyser ce paradigme de la concurrence ainsi que les solutions qui y ont été apportées.

Dans un premier temps, nous définirons ce problème, pour ensuite comparer sa spécification à celle d'autres paradigmes de la programmation répartie adressant des problèmes d'unicité. De cette étude, nous ferons ressortir les singularités du problème de l'exclusion mutuelle.

Puis, dans un deuxième temps, nous présenterons un historique des premières solutions apportées à ce paradigme, avant de donner une taxonomie des algorithmes répartis d'exclusion mutuelle (avec communication par envoi de messages) présents dans la littérature.

2.1 Définition historique

La première définition du problème de l'exclusion mutuelle a été donnée par Edsger Dijkstra en 1965 dans l'article [Dij65]. On peut résumer son propos dans les cinq assertions suivantes :

1. À tout moment, il n'y a qu'un site du système qui puisse exécuter la section critique.
2. La solution doit être symétrique, c'est-à-dire que l'on ne "peut pas introduire de priorité statique".
3. On ne peut pas faire d'hypothèse sur la vitesse des participants.
4. En dehors de la section critique, tout site peut quitter le système sans pour autant bloquer les autres.
5. Si plus d'un site désire entrer en section critique, on doit pouvoir décider en un temps fini de l'identité du site qui accédera à celle-ci.

Cette première définition a fait date et le sens du paradigme de l'exclusion mutuelle n'a depuis que peu évolué. Ainsi, on retrouve dans cette définition les principales propriétés des algorithmes d'exclusion mutuelle tels que définis de nos jours dans la littérature. Mais on y trouve aussi des contraintes qui caractérisent plus le type des systèmes répartis considérés, que le problème d'exclusion mutuelle lui même. Ainsi, la deuxième assertion impose une solution décentralisée. La troisième est elle, une des hypothèses classiques sur le synchronisme du système. Enfin, la quatrième peut être vue comme une contrainte sur le dynamisme du système.

Au fil des années, le problème de l'exclusion mutuelle s'est alors généralisé en ne conservant de cet article fondateur que la notion d'unicité (assertion 1) à la demande (assertion 5). La section suivante présente de façon formelle le résultat de cette généralisation.

2.2 Définition formelle

Définir formellement le paradigme de l'exclusion mutuelle équivaut à définir l'ensemble des propriétés qui doivent être vérifiées par tout algorithme prétendant résoudre celui-ci.

2.2.1 Automates de l'exclusion mutuelle

Avant d'énoncer ses propriétés, nous allons définir l'automate de l'exclusion mutuelle. Cet automate est constitué des trois états suivants (voir figure 2.1(a)) :

NO_REQ : état dans lequel se trouve un processus quand il exécute du code ne nécessitant pas l'accès à la section critique.

REQ : état dans lequel se trouve un processus quand il attend de pouvoir exécuter une section critique.

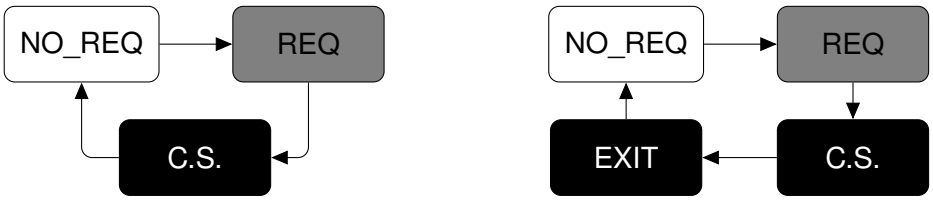
CS : état dans lequel se trouve un processus quand il exécute une section critique.

Les changements d'état se font alors circulairement de la façon suivante :

$NO\_REQ \rightarrow REQ$: À la demande de l'application, quand celle-ci veut exécuter une section critique.

$REQ \rightarrow CS$ **:** Induit par l'algorithme, quand celui-ci autorise l'application à exécuter sa section critique.

$CS \rightarrow NO\_REQ$ **:** Sur permission de l'algorithme, quand l'application lui signifie qu'elle a terminé d'exécuter sa section critique.



(a) Automate classique à 3 états (b) Automate avec l'état $EXIT$

FIGURE 2.1 – Automates de l'exclusion mutuelle

Dans cet automate (figure 2.1(a)), on remarque que la troisième transition se fait à la fois sur demande de l'application et sur contrôle de l'algorithme. C'est pour dissocier ces deux évènements de nature différente et simplifier les démonstrations que Nancy Lynch a introduit dans son livre [Lyn96] un quatrième état à l'automate : l'état $EXIT$ (voir figure 2.1(b)). Elle peut dès lors séparer les évènements liés au comportement de l'application ($CS \rightarrow EXIT$) du contrôle propre aux algorithmes d'exclusion mutuelle ($EXIT \rightarrow NO\_REQ$).

Bien qu'étant convaincu du bien fondé de cette nouvelle représentation, nous avons choisi, pour une raison de lisibilité, de considérer dans cette thèse la modélisation sur trois états. C'est en effet la modélisation la plus largement répandue à ce jour. À charge pour le lecteur et nous-même de garder à l'esprit la simplification que représente la transition ($CS \rightarrow NO\_REQ$).

2.2.2 Propriétés de l'exclusion mutuelle

À partir de l'automate précédemment décrit, nous allons maintenant définir formellement le concept d'exclusion mutuelle. Ainsi, tout algorithme réparti résout le paradigme de l'*exclusion mutuelle* définition si et seulement s'il respecte les propriétés suivantes :

Validité : Les séquences d'exécution des processus respectent l'automate défini précédemment.

Exclusion mutuelle : À tout moment il n'y a au plus qu'un processus dans l'état CS.

Progression : Si un ou plusieurs processus sont bloqués dans l'état REQ alors qu'aucun processus ne se trouve dans l'état CS, l'un d'eux doit accéder à l'état CS au bout d'un temps fini.

.

Suivant la taxonomie proposée par Leslie Lamport dans [Lam77], on peut classer ces propriétés en deux grandes familles. Les deux premières propriétés apparaissent alors

comme des propriétés de sûreté (*"safety"*) - *i.e.*, il n'arrive jamais quelque chose de mauvais. La troisième en revanche est du type vivacité (*"liveness"*) - *i.e.*, quel que soit l'état du système il peut toujours arriver quelque chose de bien.

Cette notion de vivacité est ici à prendre au sens global : il peut toujours arriver quelque chose de bien à l'ensemble du système - *i.e.*, l'exécution d'une section critique par l'un des sites demandeurs. En revanche, en ne considérant pas ici la situation particulière de chacun des sites, on n'assure pas qu'un site désireux d'exécuter une section critique finira par accéder à l'état CS. On dit alors, qu'il y a un risque de famine dans l'algorithme.

La propriété de *progression* reprend donc strictement la définition originelle d'Edsger Dijkstra. En effet, la cinquième condition ("*Si plus d'un site désire entrer en section critique, on doit pouvoir décider en un temps fini de l'identité du site qui accédera à celle-ci*") ne pose pas de condition sur le choix de l'identité du site élu.

2.2.3 Équité

La définition de l'exclusion mutuelle n'exclut donc pas la famine pour un site particulier. Assurer la "vivacité" site par site est une propriété de vivacité plus forte appelée "équité faible" :

Équité faible : Si un processus est bloqué dans l'état REQ et si les temps d'exécution des sections critiques sont finis, ce site finira par accéder à l'état CS au bout d'un temps fini.

Il est évident que la propriété d'équité faible implique celle de progression. Autrement dit, la vivacité "individuelle" implique une vivacité "collective" du système. Or, bien des applications réelles ne peuvent se contenter de la seule "progression" : Elles ne peuvent tolérer le fait qu'un site ne puisse jamais accéder à la section critique. Tant et si bien que nombre d'articles considèrent un problème d'exclusion plus fort où la progression fait place à l'équité faible en termes de vivacité.

Si l'équité faible assure à un site d'accéder à la section critique, elle n'implique pas que cet accès se fasse égalitairement. Il est par exemple possible de voir certains sites, plus prioritaires que d'autres, accéder plus souvent à la ressource critique. Pour éviter cette écueil, on peut donc être amené à utiliser une troisième propriété de vivacité, appelée "équité forte". Cette propriété est quelques fois appelée propriété de *compassion*, par opposition au terme de *justice* utilisé pour décrire l'équité faible. Si l'on trouve plusieurs définitions de cette propriété dans la littérature, toutes partagent la notion de respect d'une relation d'ordre. Dans cette thèse nous avons choisi de ne considérer que la définition suivante :

Équité forte : Un site qui demande d'entrer en section critique a la garantie qu'aucun autre site du système n'y accédera plus de k fois avant lui.

Cette définition induit la "notion" d'une file d'attente plus ou moins respectée suivant la valeur du paramètre k. Par la suite, on dira d'un algorithme qu'il est équitable s'il respecte l'ordre de cette file d'attente.

Si l'on compare maintenant l'ensemble des propriétés de vivacité, on obtient la hiérarchie suivante :

$$\text{Équité forte} \Longrightarrow \text{Équité faible} \Longrightarrow \text{Progression}$$

Dans cette thèse, nous considérons ces trois propriétés de façon séparée en précisant le cas échéant le type de vivacité considéré.

Pour finir, le lecteur intéressé par la relation Vivacité - Équité, pourra lire avec intérêt les réflexions non publiées de Dijkstra sur le sujet [Dij87] ainsi que la réponse de Lamport dans [LS88].

2.3 Comparaison du problème de l'exclusion à d'autres paradigmes de la programmation répartie

En algorithmique répartie, il existe trois grands paradigmes liés au concept d'unicité, à savoir :

- l'élection : Unicité de processus
- le consensus : Unicité de valeur
- l'exclusion mutuelle : Unicité d'accès

Pour mieux appréhender l'exclusion mutuelle, il nous est paru intéressant de la comparer à ces deux autres paradigmes. Les comparaisons présentées dans cette section n'ont pour autant pas la prétention de démontrer formellement les équivalences et les différences entre ces paradigmes. L'idée est ici de comparer intuitivement ces différents problèmes pour dégager les spécificités de l'exclusion mutuelle ainsi que ces principales contraintes. De même, on se restreindra aux problématiques intrinsèques à l'exclusion mutuelle. On ne considérera donc pas celles liées à la tolérance aux fautes dans tel ou tel de ces paradigmes.

Pour faire cette comparaison, nous introduisons le concept de réductibilité stricte. Ainsi un problème $\mathcal{P}$ est S-Réductible à un problème $\mathcal{P}'$ si et seulement si pour toutes solutions $\mathcal{A}'$ de $\mathcal{P}'$, on peut transformer **localement** les sorties de chaque processus pour obtenir une solution du problème $\mathcal{P}$. Autrement dit, on doit pouvoir transformer tout algorithme résolvant $\mathcal{P}'$ en algorithme résolvant $\mathcal{P}$ sans ajouter de message.

Ce type de réductibilité peut permettre entre autre d'étudier les complexités et la minimalité des hypothèses sur les systèmes. Elle permet aussi de raisonner en termes de diffusion des connaissances.

2.3.1 Paradigme de l'élection

En algorithmique répartie, l'élection correspond à la désignation parmi l'ensemble des sites d'un système d'un unique site du système communément appelé "leader". Contrairement à son sens commun, l'élection n'implique pas forcement de votes. Ce problème a été posé pour la première fois en 1977 par Gérard Le Lann dans son article [Lan77]. Dans sa version faible il se définit formellement de la façon suivante :

Initialisation : tous les sites du système démarrent depuis un état quelconque.

Élection : l'algorithme finit toujours par atteindre un état final dans lequel un site est dans l'état *LEADER*.

Si l'on compare maintenant les problèmes d'élection et d'exclusion mutuelle, on s'aperçoit vite que l'exclusion mutuelle n'est pas réductible à un problème d'élection. En effet, dans la spécification de l'élection, tous les sites du système peuvent accéder à l'état

LEADER. Rien ne permet de restreindre l'ensemble des candidats. On ne peut donc exclure le cas où l'algorithme désigne systématiquement comme *LEADER* un site ne désirant pas la section critique. Ainsi, on ne pourra jamais assurer la propriété de "progression" de l'exclusion mutuelle.

À l'inverse, on peut montrer que l'élection est un problème plus faible que celui de l'exclusion mutuelle. En effet il est facile d'obtenir le premier à partir du second :

1. Tous les sites font une demande d'entrer en section critique : ils passent tous dans l'état *REQ*.
2. Lorsqu'un site passe à l'état *CS* de l'exclusion mutuelle, il devient le leader.

Les propriétés de sûreté de l'exclusion mutuelle assurent l'unicité du leader tandis que celle de "progression" assure la terminaison de l'algorithme.

De ce résultat de réductibilité, on obtient une transitivité des contraintes de l'élection vers les contraintes de l'exclusion mutuelle. On peut alors reprendre pour l'exclusion mutuelle le théorème d'impossibilité que Danna Angluin a énoncé en 1980 dans [Ang80] : *"dans un système de plus d'un site il est impossible de résoudre l'élection si tous les sites du système sont indistinguables les uns des autres".*

Dans la suite de ce manuscrit on considérera donc uniquement des systèmes répartis vérifiant la propriété suivante :

Énumération : À tout instant, l'ensemble des sites du système diffère, deux à deux, d'au moins une variable appelée "identifiant". L'ensemble, composé de ces identifiants, est muni d'une relation d'ordre total ($\prec_{id}$).

2.3.2 Paradigme du consensus

Un autre problème classique d'unicité dans les systèmes répartis est celui du consensus. Il s'agit ici de mettre d'accord tous les participants du système sur l'une des valeurs proposées par ceux-ci. Il peut se définir formellement de la façon suivante :

Accord : la valeur décidée est la même pour tous les processus corrects

Intégrité : tout processus décide au plus une fois (une décision prise est définitive)

Validité : toute valeur décidée est l'une des valeurs proposées

Terminaison : si au moins un processus correct lance le consensus, tout processus correct décide au bout d'un temps fini.

Comparons maintenant notre problème à celui du consensus et pour commencer, intéressons nous à la réductibilité du consensus dans l'exclusion mutuelle. S'il parait simple d'obtenir la propriété d'"accord" à partir de la propriété de la sûreté de l'exclusion mutuelle, en conditionnant le choix de valeur par l'obtention de la section critique, il est moins évident d'obtenir la "terminaison" du consensus. En effet, rien dans les propriétés de l'exclusion mutuelle n'oblige à informer les membres du système de l'identité du détenteur de la section critique. Il apparait donc difficile de diffuser, avec l'unique exclusion mutuelle, la valeur décidée et donc d'obtenir la "terminaison". Le problème de l'exclusion mutuelle ne peut se réduire à celui du consensus.

Inversement, si l'on considère un consensus basé sur les identifiants, ses propriétés assurent que tous les sites finissent par se mettre d'accord sur l'identité de l'un d'entre

eux. Ce site peut alors exécuter la section critique tout en préservant la clause de sûreté. Une fois terminé il pourrait lancer à nouveau un consensus pour permettre à un autre site d'accéder à la section critique.

Il ne reste plus alors qu'à assurer que le consensus se fasse (ou finisse par se faire) autour d'un site désirant entrer en section critique. Comme pour l'élection, c'est ce dernier point qui va poser problème. En effet, le consensus est générique et n'impose pas de critère de choix. De plus, rien n'impose de tenir compte de toutes les valeurs proposées. Ce deuxième point pose un problème plus profond que celui du critère de choix de la valeur car il impliquerait finalement une diffusion de toute l'information présente dans le système.

Les paradigmes du consensus et de l'exclusion mutuelle apparaissent donc comme deux problèmes orthogonaux. Contrairement à l'élection, cette comparaison ne permet de déduire de nouvelles propriétés de l'exclusion mutuelle. Elle nous a tout de même permis de mettre en avant le problème de la diffusion des requêtes.

2.3.3 Spécificité de l'exclusion mutuelle

De ces études comparatives ressort la principale difficulté de l'exclusion mutuelle, à savoir : l'asymétrie. En effet même si l'unicité est une problématique commune à d'autres paradigmes de la programmation répartie, l'exclusion mutuelle se différencie d'eux par la nécessaire discrimination des sites attendant la section critique. Cette discrimination ne peut se faire qu'avec une asymétrie "originelle" du système, à partir de laquelle on pourra construire une asymétrie dynamique, évoluant en fonction des demandes des applications et diffusée dans le système.

Le lecteur pourra trouver une évocation de ce problème dans le livre de K. Mani Chandy and J. Misra [CM88]. Par ailleurs, un résultat d'impossibilité en l'absence d'asymétrie a été énoncé par James E. Burns dans [Bur81].

2.4 Premières solutions basées sur la mémoire partagée

Après avoir formalisé le problème puis caractérisé ses particularités, on peut maintenant s'intéresser aux solutions décrites dans la littérature.

La première est due au mathématicien hollandais Dekker en 1965. Celle-ci est limitée aux systèmes ne possédant que 2 sites et munis d'une mémoire partagée, les accès aux mots de cette mémoire partagée devant se faire de façon "atomique". La même année cette solution a été étendue à n sites par Dijkstra.

Cependant peut on vraiment dire que ces premières solutions résolvent le problème qui nous intéresse ? En effet toutes deux reposent entièrement sur un mécanisme d'atomicité des accès aux mots mémoire que l'on peut voir comme une primitive d'exclusion mutuelle bas niveau.

Il faut attendre 1974 et le tout premier article de Lamport [Lam74] pour avoir une véritable solution au problème de d'exclusion mutuelle. Elle repose toujours sur un système enrichi d'une mémoire partagée, mais son accès ne fait plus l'hypothèse de lecture et d'écriture atomique de mots mémoire.

L'évolution logique fut ensuite de s'abstraire de l'hypothèse d'existence d'une mémoire partagée.

2.5 Taxonomie des algorithmes d'exclusion mutuelle pour système réparti

C'est en 1977 avec l'algorithme de Gérard Le Lann [Lan77] et en 1978 avec celui de Lamport [Lam78] que l'on a commencé à considérer pour l'exclusion mutuelle des systèmes répartis sans mémoire partagée. Dans ces systèmes les processus s'exécutent sur des sites distants et ne peuvent communiquer entre eux que par envoi de messages. Depuis nombre de propositions ont été faites pour ces systèmes. Pour appréhender l'ensemble de cette littérature, il peut être intéressant de les classer suivant les concepts utilisés et les mécanismes mis en œuvre. Cette taxonomie existe déjà de manière éparse dans de multiples publications : [Ray91], [CM92], [Sin93] et [Vel93]. Nous ne ferons donc ici que regrouper les critères et noms de classe déjà décris dans celles-ci. Nous essaierons aussi le cas échéant d'indiquer quelle publication a introduit telle ou telle terminologie.

Les algorithmes d'exclusion mutuelle se classent usuellement en deux grandes familles : les algorithmes basés sur les permissions et ceux basés sur la circulation d'un jeton. Cette classification a été introduite en 1991 par Michel Raynal dans [Ray91].

2.5.1 Les algorithmes à permissions

Les algorithmes à permissions reposent sur une idée simple et naturelle : chaque site demande aux autres, s'il le désire, l'autorisation d'entrer en section critique. Il ne pourra alors entrer en section critique qu'à la seule condition d'avoir bien reçu toutes les permissions nécessaires.

Dans ce type d'algorithme, la propriété de sûreté de l'exclusion mutuelle est assurée par l'invariant suivant : "À tout moment au plus un site possède toutes les permissions nécessaires".

Pour garantir cet invariant, chaque algorithme à permissions définit les conditions nécessaires à l'envoi d'une permission à un autre site du système. On peut alors les classer suivant le type de ces conditions en trois sous-classes : les algorithmes à permissions individuelles, les algorithmes à permissions d'arbitres et les algorithmes à permissions mixtes.

2.5.1.1 Permissions individuelles

Dans un algorithme à permissions individuelles, chaque site gère les requêtes d'entrée en section critique qu'il reçoit en fonction de ses propres besoins. Autrement dit, un site ne désirant pas exécuter de section critique enverra systématiquement son autorisation tandis qu'un site désirant entrer en section critique réglera individuellement (localement) le conflit en suivant un système de priorité. Ainsi dans ce deuxième cas, le site bloquera les requêtes moins prioritaires que la sienne et enverra une permission aux autres. À charge

pour le système de priorité d'assurer la propriété de vivacité (progression et/ou équité faible).

Pour la diffusion des requêtes deux stratégies ont été proposées :

Broadcast : Les requêtes sont alors envoyées à l'ensemble du système par un mécanisme de diffusion. Le principal algorithme utilisant cette stratégie a été proposé en 1981 par Glenn Ricart et Ashok K. Agrawala dans [RA81]. Pour résoudre localement les conflits, il utilise les horloges logiques introduites par Lamport en 1978 dans [Lam78]. Chaque site augmente son estampille temporelle lorsqu'il souhaite entrer en section critique et un ordre total est construit grâce à celui des identifiants.

Multicast dynamique : L'idée est ici de minimiser l'ensemble de diffusions des requêtes. Les algorithmes de cette classe font suite à l'algorithme de Ricart et Agrawala [RA81] et se basent sur l'assertion suivante : *"Si un site a reçu la permission d'un autre, il peut considérer qu'elle reste valable tant que ce dernier ne lui a pas envoyé de nouvelle requête"*. Dans ces algorithmes, chaque site maintient une liste de permissions, une permission restant acquise jusqu'à réception d'une requête. L'entrée en section critique est donc subordonnée à la réception des permissions manquantes ce qui limite l'ensemble de diffusion à celles-ci. Plusieurs algorithmes utilisent cette stratégie, ils ne se distinguent que par les règles de priorité utilisées pour assurer la vivacité. Parmi ceux-ci on compte : [CR83] publié en 1983 par Osvaldo S. F. Carvalho et Gérard Roucairol qui comme [RA81] utilise les horloges de Lamport et [CM84] publié en 1984 K. Mani Chandy et Jayadev Misra qui lui utilise un graphe de priorité acyclique.

2.5.1.2 Permissions d'arbitres

Ici, contrairement aux algorithmes à permissions individuelles où un site finissait par recevoir l'accord explicite ou tacite (multicast dynamique) de tous les sites pour chaque entrée en section critique, un site i n'attend plus qu'un ensemble réduit de permissions. Cet ensemble de sites R_i est propre à chaque site i.

La deuxième différence des permissions d'arbitres par rapport aux permissions individuelles, est qu'un site n'envoie qu'une permission à la fois. Cet envoi n'est plus seulement conditionné par l'état de l'application vis-à-vis de la section critique, mais aussi des permissions déjà émises par le site.

Pour assurer la propriété de sûreté il faut et il suffit que tous ces ensembles de diffusion soient en intersection deux à deux, autrement dit :

$$\forall (i,j), R_i \cap R_j \neq \emptyset$$

Les algorithmes à permissions d'arbitres se distinguent par leurs ensembles de diffusion R_i. Le principal est celui décrit en 1985 par Mamoru Maekawa [Mae85], où ces ensembles sont construits à partir des plans projectifs finis. Dans cet article, Mamoru Maekawa montre aussi que ces ensembles, s'ils existent, sont les plus petits ensembles vérifiant la propriété d'intersection. La complexité de l'algorithme est alors en $\mathcal{O}(\sqrt{N})$. Le problème de cette approche est la complexité de construction de ces ensembles qui rend difficile sa mise en œuvre sur système composé d'un grand nombre de sites.

Une autre limite de ces algorithmes est l'ajout obligatoire d'un mécanisme de reprise de la permission. En effet, pour réduire l'ensemble de diffusion tout en conservant la propriété de sûreté, ils restreignent les sites à n'envoyer qu'une permission à la fois. Cette nouvelle contrainte peut aboutir à un inter-blocage et ainsi compromettre la propriété de vivacité ("progression" comme "équité faible"). Un site doit donc pouvoir récupérer sa permission pour la donner à un autre plus prioritaire. Le surcoût dû à ce mécanisme est proportionnel au nombre de requêtes concurrentes.

2.5.1.3 Permissions mixtes

Cette dernière sous-classe d'algorithme utilise, comme pour les permissions d'arbitres, des sous-ensembles R_i pour diffuser ses requêtes diminuant ainsi la complexité en nombre de messages. Mais, à l'instar des algorithmes à permissions individuelles, les processus peuvent ici émettre plusieurs permissions simultanément, ce qui écarte tout risque d'inter-blocage. La vivacité étant ainsi naturellement assurée, ils peuvent s'affranchir du coûteux mécanisme de reprise d'une permission. Le prix de cette économie est une contrainte plus forte sur les ensembles R_i pour garantir la propriété de sûreté :

$$\forall(i,j), i \in R_j \vee \ j \in R_i$$

Ce type d'algorithme a été introduit par Mukesh Singhal en 1991 dans [Sin91]. Il a noté que certaines taxonomies le classe comme un algorithme à permissions d'arbitres "sans inter-blocage". C'est d'ailleurs le titre de l'article. Le terme de *permissions mixtes*, utilisé ici, a été introduit par Michel Raynal dans son livre [CM92].

2.5.2 Les algorithmes à jeton

Étudions maintenant la deuxième grande famille que constituent les algorithmes à jeton. En considérant un historique global des accès à la section critique, le problème exclusion mutuelle apparaît comme celui de la séquentialisation de ces accès. Comme dans une course de relais, la section critique passe de site en site. L'idée des algorithmes de cette famille est alors de considérer un témoin appelé "*jeton*" (*token*). Il peut être vu comme une super permission dont la seule possession permet d'exécuter une section critique. Cette abstraction de l'exclusion mutuelle permet d'obtenir facilement la propriété de sûreté. Il suffit pour cela de maintenir l'invariant suivant "*À tout moment il n'y a dans le système qu'au plus un jeton*". Reste à assurer la vivacité, c'est-à-dire la circulation du jeton ainsi que celle des requêtes vers ce dernier.

Pour retrouver le jeton on distingue deux stratégies : la première utilise des mécanismes de diffusion, tandis que la deuxième repose sur l'utilisation des propriétés de topologie particulière.

2.5.2.1 Les algorithmes à jeton non structurés

À bien des égards, ces algorithmes ressemblent aux algorithmes à permissions, comme eux ils vont diffuser la requête dans un ensemble R_i. Cet ensemble doit avoir la propriété suivante : "*À tout moment, le site possédant le jeton appartient au* R_i". Comme dans les

algorithmes à diffusion, on observe deux stratégies pour maintenir cet invariant, tout en minimisant le nombre de messages émis :

Diffusion des requêtes dans des ensembles statiques : La façon la plus simple d'obtenir l'invariant est de diffuser les requêtes à l'ensemble du système. C'est la méthode retenue par Ichiro Suzuki et Tadao Kasami en 1985 dans leur article [SK85]. Cet algorithme repose sur un graphe logique complet. Ce graphe peut n'être qu'une abstraction d'un graphe physique fortement connexe. La même requête peut alors être retransmise plusieurs fois par les mêmes nœuds intermédiaires.

Partant de cette constatation, Jean-Michel Hélary, Noël Plouzeau et Michel Raynal ont proposé une optimisation de [HPR88] pour les réseaux : la requête est propagée de proche en proche par une "*propagation par vague*" pour trouver l'actuel possesseur du jeton. L'algorithme profite de l'information collectée au cours de la propagation pour optimiser l'envoi du jeton. Ainsi ce dernier remonte le long de l'arbre construit par la requête. Ce parcours inverse est le plus court chemin (tout du moins l'était au moment de l'émission de la requête).

Diffusion des requêtes dans des ensembles dynamiques : Comme pour les algorithmes à permissions, certains ont cherché à minimiser la complexité en messages en réduisant les R_i. Or ici tous les sites sont susceptibles de posséder le jeton. Cette réduction passe donc forcement par une dynamisation des R_i. Chaque site va alors maintenir, en fonction des informations qui lui sont parvenues, un ensemble R_i des possibles possesseurs du jeton.

Mukesh Singhal a ainsi publié une série d'articles ([Sin89], [CSL91],...) sur ce principe. L'heuristique commune à tous ces algorithmes est la suivante : "*Du point de vue d'un site S_i, le jeton est susceptible d'être possédé par l'ensemble des sites qui attendaient encore le jeton lorsque S_i a terminé sa dernière section critique ou bien par l'un des sites lui ayant envoyé une requête depuis*".

Dans son article [Sin89], Mukesh Singhal utilise cette heuristique en initialisant le système avec des ensembles imbriqués (pour chaque site S_i, $R_i = \{S_1, ..., S_{i-1}\}$). Puis dans [CSL91] il généralise son algorithme à l'ensemble des initialisations vérifiant la propriété suivante : $\forall (i,j), i \in R_j \vee \ j \in R_i$. Cette condition est identique à celle utilisée dans les algorithmes à *permissions mixtes*.

2.5.2.2 Les algorithmes structurés

L'idée de cette dernière classe d'algorithmes est d'utiliser les propriétés d'une structure logique pour retrouver et/ou acheminer le jeton. Ces algorithmes évitent ainsi l'utilisation de primitives de diffusion. Il est à noter, que le choix de la structure logique peut être imposé par l'architecture physique du système. Si tel n'est pas le cas, la structure peut être amenée à évoluer au cours du temps. On distingue ainsi les algorithmes à structure fixe des algorithmes à structure dynamique. Dans cette classification, on ne tiendra pas compte des changements d'orientation des liens de communication. Un algorithme à structure dynamique se définit donc comme un algorithme dont les voisinages des nœuds dans le

graphe *non orienté correspondant* à la structure logique peuvent être amenés à varier dans le temps.

Structure fixe : Gérard Le Lann fut le premier à introduire ce type de structure. Dans son algorithme [Lan78], il utilise un **anneau** logique pour acheminer le jeton de site en site. Alain J. Martin a ensuite enrichi cet algorithme de requête ([Mar85]). Dans cette deuxième version, l'anneau est utilisé pour acheminer en sens contraire les requêtes jusqu'au possesseur du jeton.

L'autre grand type de structure logique utilisé comme support l'exclusion mutuelle est celle de l'**arbre**. L'idée commune à tous ces algorithmes est d'utiliser l'acyclisme des arborescences pour organiser un routage des requêtes vers le jeton. Parmi ceux-ci, on peut citer l'algorithme de Kerry Raymond [Ray89] ou celui de Mitchell L. Neilsen et Masaaki Mizuno [NM91]. Une approche similaire est aussi proposée par Jan L. A. van de Snepscheut ([vdS87]), mais la structure utilisée n'est pas un arbre mais un graphe orienté acyclique.

Structure dynamique : Ces algorithmes vont faire évoluer la structure logique en fonctions des requêtes. L'idée est d'utiliser ces messages pour optimiser l'acheminement des futures requêtes vers le jeton. Mohamed Naimi et Michel Tréhel ont ainsi proposé un algorithme utilisant un arbre dynamique ([NT87b, NT96]). Le père d'un nœud est mis à jour à chaque réception de requête (son fonctionnement sera présenté plus en détail dans la section 4.1). Ce mécanisme permet non seulement d'obtenir une complexité en $\mathcal{O}(log(N))$ mais aussi de favoriser les sites qui demandent fréquemment l'accès à la section critique. Notons enfin qu'un algorithme similaire a été proposé la même année par José M. Bernabéu-Aubán and Mustaque Ahamad ([BAA89]).

Structure hybride : Jean-Michel Hélary et Achour Mostefaoui, ont caractérisé le comportement des nœuds les algorithmes de Raymond et de Naimi-Tréhel. Ils parlent ainsi d'un comportement de *mandataire* pour le premier de *messager* pour le deuxième. Ils ont alors proposé dans l'article [HM94] un algorithme générique permettant d'émuler ces deux algorithmes suivant que l'ensemble des nœuds du système adoptent l'un ou l'autre de ces comportements.

À partir de cet algorithme générique et en spécifiant des classes de nœuds pour chacun de ces comportements, ils proposent un algorithme hybride basé sur une structure en "*open-cube*". Cet algorithme permet de s'adapter dynamiquement aux requêtes, tout en maintenant les propriétés de symétries de cette structure arborescente particulière. L'idée est ensuite d'utiliser ces propriétés structurelles pour faciliter la mise en œuvre de la tolérance aux fautes.

2.6 Synthèse et conclusion

Cette taxonomie, résumée dans l'arbre de la figure 2.2, nous a permis de mettre en évidence la diversité des algorithmes. Les différentes stratégies mises en œuvre par ces algorithmes présentent chacune des avantages et des inconvénients. Il n'existe donc pas

une classe d'algorithme meilleure que les autres mais plutôt des classes adaptées à un contexte et des objectifs de performances précis.

Les algorithmes à permissions sont naturellement plus robustes vis-à-vis des pannes. Un site ayant reçu les permissions nécessaires peut accéder à la section critique sans pour autant attendre la réponse de tous les sites du système. La panne d'un site ne conduit donc pas forcément à l'arrêt du système. Les algorithmes à permissions présentent par contre trois inconvénients. Tout d'abord, ils sont couteux en messages. Si l'envoi des requêtes peut parfois se faire en *multicast* (ce n'est pas possible sur un réseau *Wide Access Network*), les permissions sont, elles, envoyées par des communications point à point. La deuxième limite de ces algorithmes intervient lorsque l'on tente de réduire la complexité en messages par l'usage de permissions d'arbitres. Si Mamoru Maekawa a montré que l'on pouvait se ramener à une complexité en $\mathcal{O}(\sqrt{N})$ ([Mae85]), il est très difficile de construire les ensembles permettant d'atteindre cette limite. Enfin, ajouter des sites aux systèmes tout en conservant de bonnes performances de fonctionnement demande l'ajout de mécanismes complexes et coûteux en messages.

À l'inverse, qu'ils soient basés sur l'utilisation de structures simples ou bien de mécanismes de diffusion, les algorithmes à jeton présentent l'avantage d'être beaucoup plus facile à mettre en œuvre, ce qui permet d'envisager leur application dans des systèmes de grande taille. Ils sont aussi beaucoup moins couteux en termes de messages : les sites n'attendant que la réception du jeton pour entrer en section critique, on évite le coût des envois point à point des permissions. L'envoi des requêtes peut, quant à lui, s'adapter au type d'environnement. Les algorithmes non structurés seront à privilégier si le réseau offre un service de diffusion, dans le cas contraire, les algorithmes structurés prendront tout leur sens. Notons que pour ces derniers, il peut y avoir une augmentation du délais moyen d'obtention de la section critique en raison des retransmissions des messages. L'utilisation de structures dynamiques, comme dans l'algorithme de Naimi-Tréhel, permet alors de limiter ce surcoût.

Un autre avantage des algorithmes à jeton structurés sur les algorithmes à permissions, réside dans la facilité d'ajout de nouveaux participants. Le mécanisme d'insertion ne modifie que localement la structure : ajout de feuille dans le cas de Naimi-Tréhel, insertion dans l'anneau pour les dérivés de Le Lann ou bien encore ajout d'un nœud dans le graphe pour l'algorithme de Raymond.

Mais si l'ajout est simple, la gestion de la panne d'un nœud est rendu beaucoup plus difficile. En effet, dans les algorithmes à jeton structurés, la vivacité est entièrement basée sur les propriétés des structures utilisées. Ces propriétés peuvent être entièrement remises en cause par la panne d'un simple site et leur recouvrement nécessite des mécanismes complexes.

C'est pour leurs bonnes propriétés de passage à l'échelle que nous avons choisi dans cette thèse de nous intéresser aux algorithmes d'exclusion mutuelle à jeton. Leur défaut majeur étant leur faible résistance aux pannes, nous allons dans le chapitre suivant traiter de cette problématique.

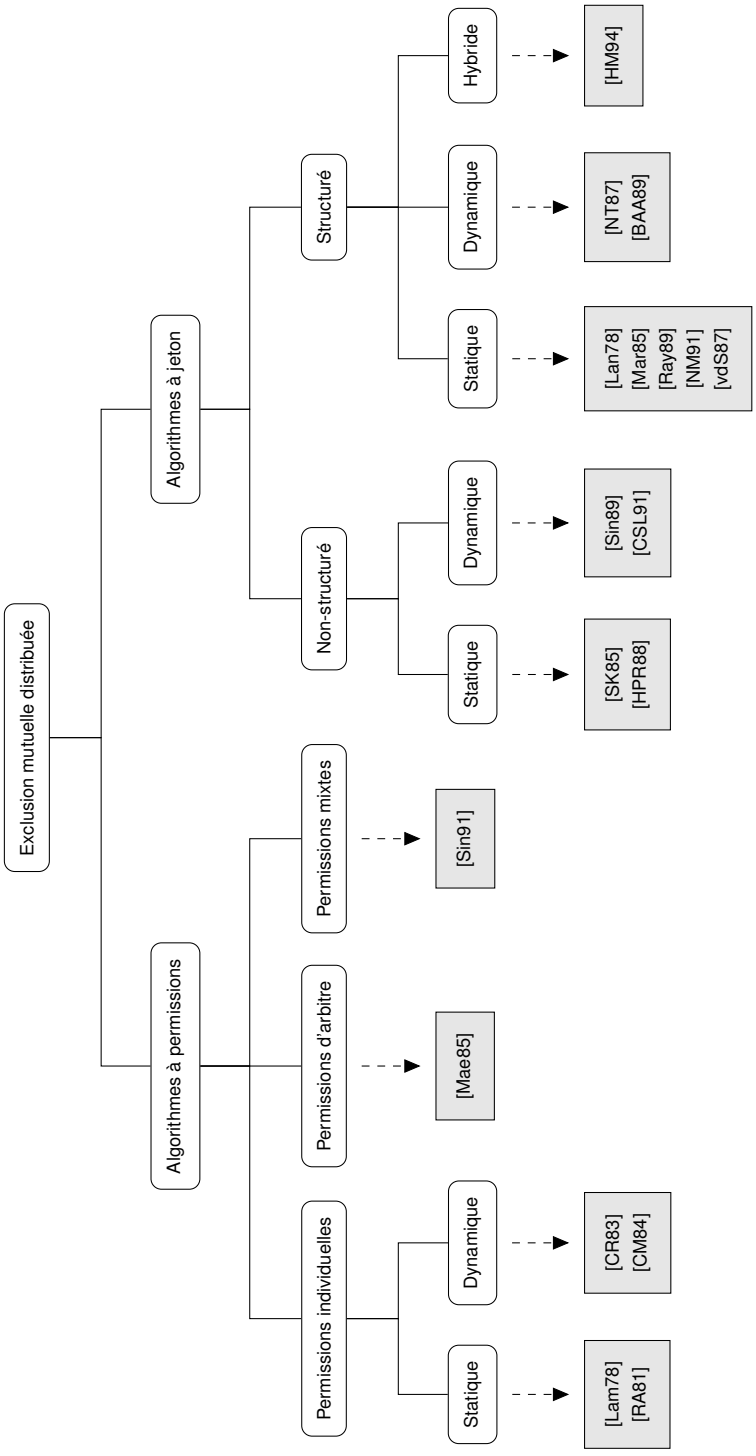


FIGURE 2.2 – Taxonomie des algorithmes distribués d'exclusion mutuelle

Première partie

Algorithme d'exclusion mutuelle tolérant aux défaillances

Chapitre 3

État de l'art des algorithmes à jeton tolérants aux défaillances

Sommaire

Comme nous l'avons vu dans l'introduction, l'agrégation des ressources existantes au sein de grands systèmes répartis permet de répondre à moindre coût aux besoins toujours croissants des applications. Or le fait de regrouper un nombre important de machines rend la probabilité d'une panne très élevée. La mise en œuvre d'algorithmes d'exclusion mutuelle répartis dans ces grands systèmes est donc conditionnée par la gestion des défaillances.

Parmi, l'ensemble des algorithmes d'exclusion mutuelle présentés au chapitre 2, nous nous intéressons ici aux algorithmes reposant sur l'utilisation d'un jeton unique, dit *algorithme à jeton* (voir section 2.5.2). Ces derniers offrent l'avantage d'être relativement peu coûteux en termes de messages, mais en contrepartie, ils sont particulièrement sensibles aux défaillances.

Dans ce chapitre, nous commençons par présenter les différentes hypothèses qui caractérisent les systèmes répartis. Puis nous feront un état de l'art des algorithmes à jeton tolérants les défaillances et proposerons une synthèse des différents systèmes considérés par ces algorithmes. Ensuite, nous présenterons les principales raisons qui font de

l'algorithme de Naimi-Tréhel un algorithme particulièrement bien adapté aux systèmes répartis à grande échelle. Enfin, nous introduirons les différentes versions tolérantes aux défaillances de cet algorithme.

3.1 Modèle

Pour pouvoir comparer correctement deux algorithmes tolérant aux défaillances quel que soit le problème adressé, il faut d'abord commencer par comparer le type de défaillance qu'ils considèrent. Il faut aussi prendre en considération les hypothèses posées sur le système réparti sous-jacent. En effet, les mécanismes de détection et de recouvrement des défaillances reposent entièrement sur les propriétés des systèmes répartis visés.

Nous allons donc dans un premier temps, lister les différentes hypothèses présentes dans la littérature. Certaines d'entre elles sont communes à l'ensemble de l'algorithmique répartie, d'autres au contraire, sont propres au paradigme de l'exclusion mutuelle.

À partir de cette liste, nous dresserons un tableau comparatif des algorithmes d'exclusion mutuelle à jeton tolérants aux défaillances en fonction des systèmes considérés.

3.1.1 Répartition

Un système réparti est un ensemble de *processus* s'exécutant sur des machines appelées *nœuds* ou *sites*. Dans la suite de cette thèse nous supposerons qu'il n'y a qu'un seul processus par machine. Nous considèrerons comme interchangeable les termes de : *processus*, *nœuds* ou *sites*.

3.1.1.1 Identification des nœuds

Un système réparti est composé d'un ensemble de nœuds que l'on a ou non distingués au départ. Ainsi, on peut avoir un système :

anonyme : Tous les nœuds au système partagent exactement le même état initial.

partiellement distinguable : Certains nœuds du système sont distinguables par rapport à d'autres. On peut alors construire des classes de nœuds, permettant d'établir une relation d'ordre partiel sur les nœuds.

identifiable : Tous les nœuds du système sont munis d'un identifiant propre. On peut alors construire un ordre total sur ces identifiants.

3.1.1.2 Dynamicité du système

La propriété de dynamicité d'un système réparti caractérise l'évolution de l'ensemble de ses nœuds. Un système peut être :

statique : Un ensemble de nœuds, prédéfini, participe au système dès son lancement. Il représente l'univers du système. Ces systèmes ne peuvent donc tolérer qu'un nombre fini de pannes "franches".

pseudo dynamique : Pour lever cette restriction sur le nombre de pannes "franches" (inférieur au nombre de nœuds du départ), on peut ajouter un état dégradé au système pendant lequel l'ajout de nœud est possible (avant de repasser à l'état normal de fonctionnement). Ce système "pseudo dynamique" peut être vu comme une itération de système statique.

dynamique : À tout moment un nouveau nœud peut être ajouté au système. Le nombre de pannes "franches" possible n'est alors plus borné. D'autre part, la gestion des arrivées et des départs peut poser des problèmes sur la vision que les sites ont du système.

3.1.1.3 Connaissance globale ou vision locale

Dans un système réparti chaque nœud connaît un certain nombre des autres membres du système. On parle alors de vue. Cette notion n'est pas forcément liée à celle de la topologie, un nœud n'étant pas forcément directement connecté à un élément de sa vue. Un système peut avoir :

vision globale : À tout moment, chaque nœud connaît l'ensemble des autres nœuds et à fortiori, le nombre de ces nœuds.

vision locale : Les nœuds n'ont qu'une vision partielle des nœuds.

3.1.2 Communication

3.1.2.1 Type de communication

Dans un système reparti, il existe deux grands types de communication :

envoi de messages : Des liens directionnels, appelés canaux de communication, relient entre eux les nœuds du système. L'information y circule sous forme de messages. Ces canaux peuvent être **bi-directionnels** ou **uni-directionels**

mémoire partagée : Le système est muni d'une mémoire partagée par l'ensemble des nœuds. Chacun d'eux peut lire les informations écrites par les autres. Cette mémoire peut être matérielle ou une émulation logicielle. Il est donc possible d'adapter l'ensemble des algorithmes à mémoire partagée aux systèmes communicants par messages.

3.1.2.2 Connexité du graphe et partitionnement

Quel que soit le mode de communication, un système réparti peut être vu comme un graphe où chaque arrête représente une connexion logique entre deux nœuds. Ce graphe peut être :

Non-orienté complet : Tous les nœuds du système peuvent communiquer deux à deux sans intermédiaire. C'est naturellement le cas des systèmes à mémoire partagée, mais cela peut être aussi le cas pour les systèmes communiquant par envoi de messages les canaux sont alors bi-directionels.

Non orienté connexe : Les canaux sont bi-directionels, mais deux sites quelconques du système ne sont pas forcément reliés directement. En revanche, ils peuvent toujours communiquer en utilisant d'autres sites du système comme intermédiaire. Ce type de système peut, en présence de défaillances, donner lieu à un partitionnement.

Orienté fortement connexe : Comme précédemment, il existe toujours un chemin permettant à un site de communiquer ce qui pose là encore des problèmes de partitionnement. Mais ici, les canaux sont unidirectionnels. Le chemin pour la communication inverse peut donc différer.

3.1.2.3 Ordonnancement des messages

Dans un système de communication par messages, une propriété importante concerne l'ordre de délivrance par rapport à l'ordre d'émission. Un canal est dit :

FIFO (First In First Out) : Si les messages reçus par un site et provenant d'un même émetteur, sont délivrés dans leur ordre d'émission.

3.1.2.4 Service de diffusion

Les algorithmes répartis utilisant une communication par envoi de messages, reposent sur une couche plus basse offrant ces services de communication. Parmi ces services on trouve naturellement l'envoi et la réception de messages, mais aussi d'autres services servant à la communication de groupe. Ainsi, on distinguera les systèmes :

avec service de diffusion (*broadcast*) : ce service permet diffuser une information à l'ensemble des membres du système. Pour les calculs de complexité, on considère alors que la diffusion correspond à l'envoi d'un seul message. Exemple : *réseau local Ethernet*.

sans service de diffusion : en absence de service, la diffusion d'information est toujours possible, mais elle devient coûteuse. Diffuser une information à n sites correspond à émettre n messages distincts. C'est le cas par exemple dans les réseaux WAN (*Wide Area Network*).

3.1.3 Défaillance

Avant d'aborder les types de défaillance qui peuvent affecter un système, nous devons commencer par introduire les notions de défaillance, faute et erreur.

Un élément est dit **défaillant** si son comportement n'est pas conforme à sa spécification [Lap85]. La défaillance fait partie d'un cycle dont le premier élément est la faute. La **faute** (*fault*) est la *cause* qui provoque une erreur, elle peut être entre autres une condition, un événement ou une action. Une **erreur** (*error*) est la conséquence d'une faute. L'erreur peut être latente, c'est à dire que la partie du système où elle réside n'est pas encore utilisée ; elle n'est donc pas déclarée. Une fois activée l'erreur peut conduire à une défaillance.

La défaillance d'un élément du système pouvant constituer une faute pour un autre élément du système, on aboutit au cycle présenté sur la figure 3.1.

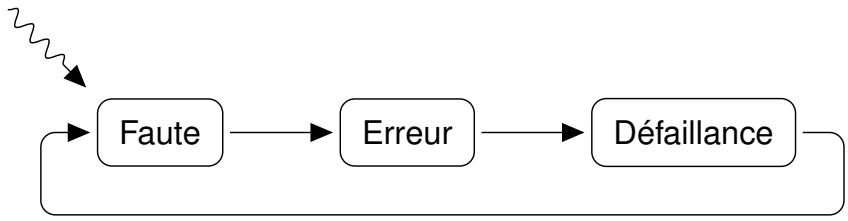


FIGURE 3.1 – Cycle des défaillances dans un système

3.1.3.1 Types de défaillance

On distingue plusieurs types de défaillance pouvant affecter le fonctionnement d'un système. Notons que chacun de ces types se décline suivant l'objet qu'il affecte : site ou canal de communication. D'autre part, on peut établir une hiérarchie dans les défaillances : un type de défaillance est plus faible qu'un autre s'il en est un sous ensemble. Ainsi, les défaillances introduites ici sont classées de la plus faible à la plus forte :

Défaillance franche ou par arrêt ou crash (*fail-silent*) **:** cette défaillance entraîne l'arrêt définitif du composant. Avant cette défaillance le processus a un comportement normal et à partir de celle-ci le processus cesse définitivement toute activité.

Un canal de communication qui subit une telle défaillance est coupé définitivement.

Défaillance transitoire ou omission (*omission failure*) **:** cette défaillance entraîne une cessation momentanée de l'activité du composant. Dans le cas d'un processus certains messages peuvent, par exemple, ne pas être envoyés. Le processus reprend ensuite le cours normal de son activité et devient silencieux.

Pour un canal de communication, les défaillances par omission correspondent à des pertes ponctuelles de messages.

Crash et recouvrement (*fail-recovery*) **:** cette défaillance ne s'applique qu'au processus et elle entraîne elle aussi l'arrêt du processus mais dans ce cas le processus peut parvenir à reprendre son activité. Cette reprise est souvent conditionnée par une phase de recouvrement afin d'obtenir l'accord d'autres processus du système. La différence avec les défaillances précédentes réside dans la perte de l'état local du processus.

Il n'y a pas d'équivalent pour un canal de communication car les canaux n'ont pas d'état.

Défaillance temporelle (*timing failure*) **:** cette défaillance suppose que les hypothèses temporelles réalisées sur la durée d'une tâche ou sur le délai de transmission d'un message ne sont pas respectées. Cette défaillance peut être due au ralentissement ou à l'accélération de la durée d'une tâche ou de la transmission d'un message.

Défaillance arbitraire ou malicieuse (*byzantine failure*) **:** affecté d'une telle défaillance, le processus adopte un comportement imprévisible. Cette défaillance qui regroupe l'intégralité des comportements possibles permet de modéliser l'intrusion d'une personne mal intentionnée dans le système.

Pour un canal de communication cela correspond à la perte, à la corruption de messages ou à la génération de "faux" messages.

Par opposition aux composants affectés par une défaillance, dit **incorrects**, un processus ou un canal de communication est qualifié de **correct** s'il ne subira aucune défaillance pendant une exécution donnée (ni dans son passé, ni dans son futur). Un processus ou un canal est dit **vivant** tant qu'il n'est pas tombé en panne.

3.1.3.2 Modèles de canaux

Afin de raffiner le modèle de défaillances, il est courant de séparer la description du comportement des sites de celui des canaux de communication. Il existe plusieurs modélisations pour les canaux de communication : fiable, quasi-fiable, équitable. Certaines caractéristiques sont identiques dans ces trois modélisations :

- *Pas de création* : Si un processus q reçoit un message m d'un processus p, alors le processus p a envoyé le message m au processus q.
- *Pas de duplication* : Si un processus p envoie un message m à un processus q, alors le processus q recevra le message m du processus p au plus une fois.

En revanche, la caractérisation des canaux de communication en termes de *perte de messages* diffère entre ces trois modèles [CASD85, HT93] :

canaux non fiables (*lossy channels*) **:** Tout message envoyé peut être perdu.

canaux équitables (*fair-lossy channels*) **:** Si un processus correct p envoie un message m une infinité de fois à un processus correct q alors le processus q reçoit une infinité de fois le message m.

canaux quasi-fiables (*quasi-reliable channels*) **:** Si un processus correct p envoie un message m à un processus correct q alors q reçoit finalement le message m.

canaux fiables (*reliable channels*) **:** Si un processus p envoie un message m à un processus correct q alors q reçoit finalement le message m.

Remarquons enfin, qu'il est relativement simple d'implémenter un canal quasi-fiable à partir d'un canal équitable. La mise en place d'un mécanisme de numérotation des messages permet d'en détecter les pertes et donc de les retransmettre. La mise en place de tels mécanismes ne permet pas d'obtenir un canal fiable car le processus émetteur doit être correct pour assurer la retransmission.

3.1.3.3 Nombre et simultanéité des défaillances

Limiter le nombre de défaillances tolérées peut être une façon de lever certains problèmes. On trouve deux types d'hypothèse sur la cardinalité des défaillances :

Nombre de défaillances totales : Le nombre total de défaillances pouvant être tolérées sur toute la durée du système.

Nombre de défaillances simultanées : Le nombre maximum de défaillance tolérable par l'algorithme à un instant donné du système.

3.1.3.4 Équité devant les défaillances

On peut, dans certains cas, restreindre le problème en ne considérant pas que tous les sites sont équitables devant les défaillances. Cette inéquité peut être statique (i.e., certains sites du système, par exemple des serveurs, ne peuvent tomber en panne) mais aussi dynamique. Dans ce dernier cas, on limite l'apparition des défaillances à certains états, les autres étant considérés comme sûrs. Pour le paradigme de l'exclusion mutuelle, on pourra considérer différemment les trois états : NO_REQ, REQ et CS. L'hypothèse généralement utilisée est que le site possédant le jeton ne peut être victime de défaillance (qu'il exécute ou non une section critique).

3.1.3.5 Nœud avec mémoire stable

Enfin, avec les défaillances se pose le problème de la reprise d'erreur et donc des données disponibles lors de cette reprise. On peut alors faire deux types d'hypothèse sur la mémoire stockant ces données :

Mémoire stable : Ce type de mémoire suppose que les données qui y sont enregistrées sont toujours disponibles après la défaillance. Un nœud peut alors toujours revenir à un état qu'il a enregistré. Si l'enregistrement dans cette mémoire est ponctuel, il peut se poser des problèmes de cohérence entre les sauvegardes (effet domino ...).

Mémoire volatile : Lors d'une faute toutes les données sont perdues. Pour sa reprise d'erreur, un site ne pourra alors compter que sur l'information qu'il collectera auprès des autres sites corrects.

3.1.4 Synchronisme

Le synchronisme est la propriété essentielle des systèmes distribués, elle exprime le rapport des composants du système par rapport au temps : il faut entendre ici la vitesse relative d'exécution des processus [1] et les délais de transmission des messages [2].

Parmi les modèles de synchronisme, on distingue trois grandes classes :

modèles asynchrones : Dans ce type de système il n'existe aucune borne, que ce soit pour la vitesse relative entre les différents processus ou pour les délais de transmission. Il est intéressant car tous les algorithmes qui sont réalisés pour un tel modèle pourront être implémentés sur n'importe quel système réel. Mais en présence possible de défaillances, certains problèmes ne peuvent y trouver de solution déterministe. Michael Fischer, Nancy Lynch et Mike Paterson, ont par exemple montré dans [FLP85] que le consensus n'est pas soluble sous ces hypothèses dès lors qu'il peut y avoir au moins une panne franche [3].

modèles synchrones : Dans ce type de système les délais d'acheminement des messages sont bornés par la borne δ et la vitesse relative entre les différents processus est

1. le ratio de temps entre le plus rapide et le plus lent des processus du système pour réaliser une tâche
2. le temps entre l'émission d'un message et sa réception
3. dans ce contexte, les défaillances temporelles n'ont pas de sens car il n'y a pas d'hypothèses temporelles

bornée par ϕ. Les bornes δ et ϕ sont connues. Ce modèle peut être difficilement implémentable dans un système réel, notamment lorsque les sites sont éloignés. En revanche, la plupart des problèmes y sont aisément solubles.

modèles partiellement synchrones : Ce modèle peut être vu comme un niveau intermédiaire des deux précédents. Comme dans le modèle synchrone, les bornes δ et ϕ existent, mais elles ne sont pas connues. Contrairement au modèle asynchrone, le consensus peut y être résolu en présence possible de panne(s) franche(s).

3.2 Etat de l'art des systèmes considérés

Nous venons donc de définir les différentes hypothèses utilisées par les algorithmes répartis tolérants aux défaillances. Nous allons maintenant donner une vision générale de leur utilisation dans le cadre de l'exclusion mutuelle.

Cette étude des modèles rassemble un large spectre d'algorithmes d'exclusion mutuelle tolérants aux défaillances. Néanmoins, ayant choisi de travailler sur les algorithmes d'exclusion mutuelle à jeton (voir section 2.6), nous nous sommes limités à l'étude de cette famille. Pour la même raison, ne sont présentés ici que des algorithmes distribués communiquant par envoi de messages et adressant la propriété de vivacité dite d'*équité faible* (voir section 2.2.3). Sans prétendre être exhaustive, cette étude regroupe une grande partie des travaux de la littérature répondant à ces critères.

La figure 3.2, reprend la taxonomie présentée à la section 2.5 pour classifier les algorithmes étudiés. Pratiquement toutes les classes d'algorithmes à jeton sont représentées ; seuls les algorithmes utilisant une *diffusion dans un ensemble statique* (algorithme de Suzuki et Kasami [SK85]) sont absents. Les travaux sur la tolérance aux défaillances des *algorithmes non-structurés* ([NLM90],[MS94] et [MS96]) sont eux basés sur l'algorithme de Mukesh Singhal [Sin89]. L'article original de Singhal proposait d'ailleurs un mécanisme de gestion des défaillances.

Les trois classes d'algorithmes structurés (statique, dynamique et hybride) sont elles toutes présentes dans notre étude. Ainsi, l'on y trouve bien des algorithmes à *structure statique*, qu'il s'agisse d'anneau, d'arbre ou de graphe. Les algorithmes utilisant un anneau sont basés sur l'algorithme de Gérard Le Lann [Lan78] qui comprenait déjà un mécanisme de tolérance aux défaillances. Les autres algorithmes présentés ici sont en fait des algorithmes d'élection qui viennent optimiser l'algorithme de Le Lann. On retrouve aussi ici l'algorithme à arbre de Kerry Raymond [Ray89], lequel contenait déjà une extension permettant le recouvrement de défaillance. Ye-In Chang, Mukesh Singhal et Ming T. Liu ont étendu cet algorithme aux graphes orientés acycliques ([CSL90b]), l'exploitation de chemins alternatifs leur permettant alors de tolérer un certain nombre de pannes franches. Cette version de l'algorithme de Raymond a été modifiée par Sandeep Kulkarni et Dhananjay Madhav Dhamdhere [KD94] afin d'éviter la formation de cycle dans le graphe.

Pour les algorithmes à *structure dynamique*, on retrouve l'algorithme de Naimi-Tréhel [NT87b] dans une version tolérante aux pannes franches présentée dans [NT87a]. On trouve aussi un algorithme complètement original proposé par Divyakant Agrawal et Amr El Abbadi. Comme celui de Naimi-Tréhel, il repose sur un arbre dynamique, mais celui-ci n'est mis-à-jour qu'à l'envoi du jeton de manière à toujours avoir le jeton pour racine. Son

mécanisme de tolérance aux défaillances repose, lui, sur l'utilisation des horloges logiques de Lamport [Lam78]. Le troisième algorithme de cette catégorie [Mue01], a été proposé par Frank Mueller. Il n'est pas à proprement parler un algorithme d'exclusion mutuelle, mais plutôt une sur-couche générique de tolérance aux défaillances pour l'exclusion mutuelle. L'auteur l'utilise pour donner une version fiable de l'algorithme de Naimi-Tréhel. C'est cet algorithme qui est cité ici.

On retrouve enfin, l'algorithme hybride de Jean-Michel Hélary and Achour Mostefaoui ([HM94]) déjà présenté dans la taxonomie (section 2.5.2.2). Cet algorithme exploite les propriétés de symétrie de l'open-cube pour tolérer les pannes franches.

Pour terminer, nous avons inclus à ce comparatif l'algorithme proposé par Carole Delporte-Gallet, Hugues Fauconnier, Rachid Guerraoui et Petr Kouznetsov dans l'article [DGFGK05]. Cet algorithme est en fait donné dans l'article, comme preuve de suffisance d'un mécanisme de surveillance ad hoc, appelé : détecteur $\mathcal{T}$ (nous reviendrons sur cette notion dans l'étude du tableau). Il n'utilise d'ailleurs pas la notion de jeton et pourrait être classé dans les algorithmes d'exclusion mutuelle à permissions. Néanmoins, il nous est apparu important, de l'inclure à notre étude, afin d'avoir une vision complète des modèles temporels permettant de résoudre le paradigme de l'exclusion mutuelle en présence de défaillances.

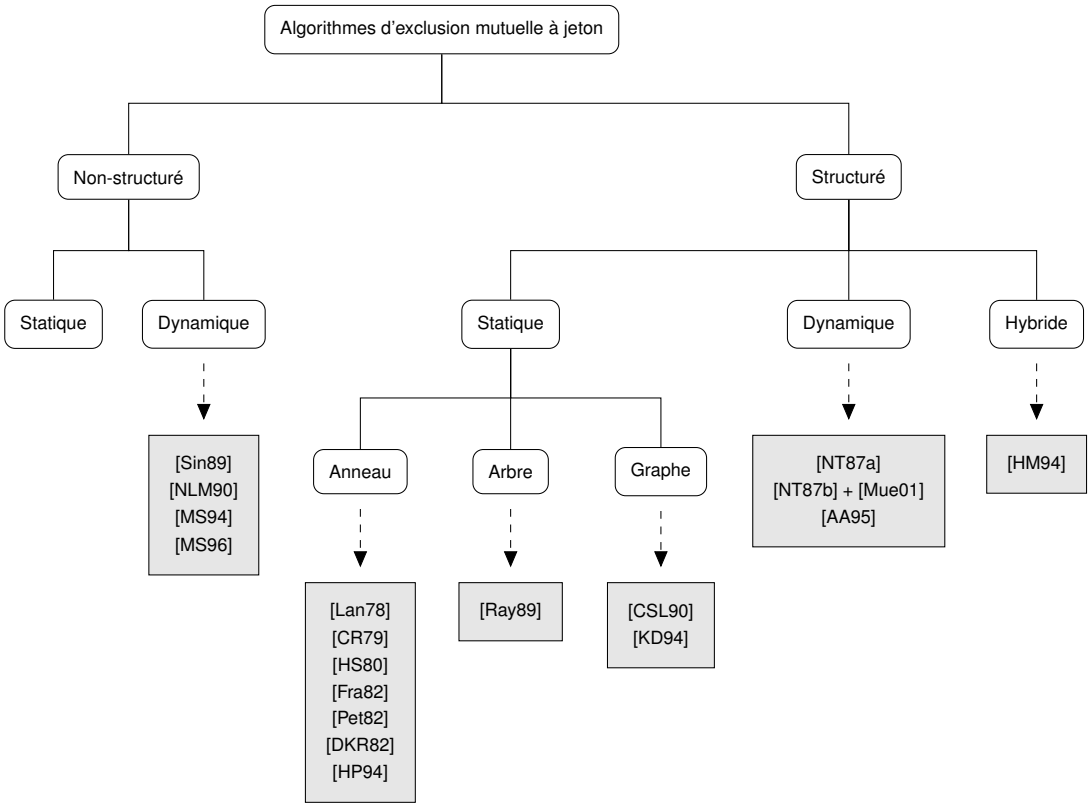


FIGURE 3.2 – Taxonomie des algorithmes distribués d'exclusion mutuelle tolérants aux défaillances

Le tableau de la figure 3.1 rassemble pour chacun de ces algorithmes, le détail du

modèle qu'il considère. Les différentes hypothèses sont regroupées en trois classes suivant qu'elles caractérisent :

Le type d'exclusion mutuelle :
- suivant que la propriété de sûreté (unicité du jeton) est assurée à tout moment (Strict) ou à terme lorsque le système se stabilise (Final.).

Les canaux de communication :
- topologie : Anneau unidirectionnel (Uni.), Anneau bidirectionnel (Uni.), Graphe non orienté complet (Comp.) et Graphe non orienté connexe mais non complet (Conn.).
- fiabilité des transmissions
- canaux FIFO.
- Utilisation d'un service de diffusion.

Les défaillances :
- type de défaillance : panne franche (Stp.), panne franche avec reprise (Rec.) et panne franche avec ou sans reprise (Stp./Rec.).
- équité devant les défaillances.
- nombre défaillances totales sur la durée du système.
- nombre défaillances simultanées reprise.
- le modèle temporel : synchrone (Sync.), Partiellement synchrone (Part.) et Asynchrone enrichi du détecteur $\mathcal{T}$ ($\mathcal{T}$).
- nœuds du système munis d'une mémoire stable.

Notons pour terminer la description du tableau, que les algorithmes noté *Modif.* sont en fait des modifications proposées dans les articles originaux afin de pouvoir s'adapter à d'autres modèles que ceux initialement considérés par les auteurs.

Ce tableau permet de mettre en évidence plusieurs points intéressants dans le choix des hypothèses. Tout d'abord, si l'on compare globalement les algorithmes, on s'aperçoit que très peu d'entre eux partagent les mêmes hypothèses. Choisir un algorithme d'exclusion mutuelle tolérant aux défaillances, c'est donc d'abord bien définir le système considéré. Le choix se restreint alors à un petit nombre d'algorithmes.

3.2.1 Hypothèses pour garantir la vivacité

Pour garantir la propriété de vivacité, les algorithmes doivent assurer la circulation des requêtes et du jeton. Ceci n'est possible qu'en absence de partitionnement du système. Notons que cette contrainte s'applique pour la vivacité de type *progression* comme pour la vivacité de type *équité faible*.

Les algorithmes reposant sur l'utilisation d'un graphe logique non orienté, *connexe* mais pas *complet*, doivent donc ajouter une contrainte supplémentaire sur le nombre de défaillances simultanées. Pour un graphe $k+1$-*connexe*, le nombre de fautes sera alors borné par la constante k.

	Exclusion mutuelle	Topologie logique	Fiabilité des transmissions	Canaux FIFO	Utilisation de la diffusion	Type de défaillance	Équité devant les défaillances	Nombre défaillances totales	Nombre défaillances simultanées	Le modèle temporel	Nœud avec mémoire stable
Le Lann [Lan78]	Strict.	Uni.	Non	Oui	Non	Msg	Oui	-	-	Sync.	Non
Chang-Roberts [CR79]	Strict.	Uni .	Non	Non	Non	Msg	Oui	-	-	Sync.	Non
Hirschberg-Sinclair [HS80]	Strict.	Bi.	Non	Non	Non	Msg	Oui	-	-	Sync.	Non
Franklin [Fra82]	Strict.	Bi.	Non	Oui	Non	Msg	Oui	-	-	Sync.	Non
Peterson [Pet82]	Strict.	Uni .	Non	Oui	Non	Msg	Oui	-	-	Sync.	Non
Dolev-Klawe-Rodeh [DKR82]	Strict.	Uni.	Non	Oui	Non	Msg	Oui	-	-	Sync.	Non
Higham-Przytycka [HP94]	Strict.	Uni.	Non	Oui	Non	Msg	Oui	-	-	Sync.	Non
Mueller [Mue01]	Strict.	Comp.	Oui	Non	Non	Stp	Oui	1	1	Part.	Non
Mueller (Modif.) [Mue01]	Strict.	Comp.	Oui	Non	Non	Stp	Oui	1	k	Part.	Non
Singhal [Sin89]	Strict.	Comp.	Oui	Non	Non	Stp/Rec	Oui	$\infty/n-1$	$n-1$	Sync.	Oui
Nishio-Li-Manning [NLM90]	Strict.	Comp.	Non	Non	Oui	Rec	Oui	∞	$n-1$	Part.	Oui
Nishio-Li-Manning (Modif.) [NLM90]	Strict.	Comp.	Non	Non	Oui	Rec	Oui	∞	$n-1$	Part.	Non
Manivannan-Singhal [MS96]	Strict.	Comp.	Non	Non	Oui	Stp/Rec	Non	$\infty/n-1$	$n-1$	Part.	Oui
Raymond [Ray89]	Strict.	Conn.	Oui	Non	Non	Rec	Oui	∞	k	Sync.	Non
Raymond (Modif.) [Ray89]	Strict.	Conn.	Oui	Non	Non	Rec	Oui	∞	k	Part.	Oui
Chang-Singhal-Liu [CSL90b]	Strict.	Conn.	Non	Non	Non	Stp/Rec	Oui	$\infty/n-1$	k	?	Oui
Kulkarni-Dhamdhere [KD94]	Strict.	Conn.	Non	Non	Non	Stp/Rec	Non	$\infty/n-1$	k	Part.	Non
Naimi-Tréhel [NT87a]	Strict.	Comp.	Oui	Non	Oui	Stp	Oui	$n-1$	$n-1$	Sync.	Non
Naimi-Tréhel (Modif.) [NT87a]	Final.	Comp.	Oui	Non	Oui	Stp	Oui	$n-1$	$n-1$	Part.	Non
Agrawal-Abbadi [AA95]	Strict.	Comp.	Non	Oui	Non	Stp/Rec	Non	$\infty/n-1$	$n-1$	Part.	Oui
Helary-Mostfaoui [HM94]	Final.	Comp.	Oui	Non	Non	Stp/Rec	Oui	$\infty/n-1$	$n-1$	Part.	Oui
Delporte-Fauconnier-*al* [DGFGK05]	Strict.	Comp.	Oui	Non	Non	Stp	Oui	$n-1$	$n-1$	$\mathcal{T}$	Non

TABLE 3.1 – Tableau des sytèmes considérés

3.2.2 Hypothèses pour garantir la sûreté

La propriété de sûreté pose le problème de la justesse de la détection de la perte du jeton. L'algorithme ne pourra régénérer le jeton qu'à la condition d'être sûr que ce dernier est bel et bien perdu. Or, cette certitude ne peut être obtenue sans poser des hypothèses temporelles sur le système et/ou les défaillances. On peut dégager différentes stratégies pour assurer la propriété de sûreté :

Système synchrone : La première solution est l'utilisation d'une borne temporelle connue. Sous cette hypothèse les algorithmes peuvent détecter de manière certaine les pannes franches des sites et donc les pertes du jeton. Le nombre de pannes franches n'est alors limité par le nombre de participants. Notons enfin, que les canaux de communication doivent être fiables afin de pouvoir exploiter cette hypothèse de synchronisme.

Pas de perte de jeton : Puisque le problème découle de la disparition possible du jeton, une solution est de considérer qu'il n'y a pas perte de jeton. L'algorithme suppose alors comme sûr le possesseur du jeton. Si les canaux utilisés ne sont pas fiables, cela suppose aussi qu'il n'y ait pas de défaillance sur l'émetteur et le récepteur lors de la transmission du message. Notons que ces propriétés de fiabilité sont dynamiques et donc difficilement implémentables. Le reste des hypothèses sur le système peuvent alors être relâchées : panne franche et système temporel *partiellement synchrone*.

Panne avec recouvrement : Plutôt que de supposer le possesseur du jeton comme sûr, certains algorithmes se limitent aux pannes avec recouvrement : les sites peuvent tomber en panne mais ils finiront tous par exécuter la procédure de recouvrement prévue par l'algorithme. Les hypothèses de synchronisme dépendent alors du type de recouvrement : système *partiellement* synchrone si le site est muni d'une mémoire stable ou système *synchrone* avec mémoire volatile. Dans ce dernier cas, un site qui s'interrompt en possédant le jeton devra le régénérer à partir des informations collectées auprès de ses voisins corrects.

Détecteur $\mathcal{T}$: Cette approche consiste à externaliser le problème de la détection de la perte du jeton dans un détecteur de défaillances adapté. Intuitivement, puisque l'on ne peut tolérer aucune erreur de détection et puisque l'on doit toujours finir par détecter une perte lorsqu'elle a lieu, un détecteur parfait (*Détecteur P*) semble adapté à notre problème. Cependant, Carole Delporte, Hugues Fauconnier, Rachid Guerraoui et Petr Kouznetsov ont montré dans l'article [DGFGK05] qu'il existait un détecteur plus faible appelé **détecteur $\mathcal{T}$** (*Trusting failure detector*) permettant de résoudre l'exclusion mutuelle. Ce détecteur est d'ailleurs le détecteur le plus faible pour résoudre ce paradigme en présence de défaillance. On peut le voir comme une optimisation de l'initialisation du détecteur parfait : une fois la confiance établie avec un site on ne peut se tromper sur son état, et s'il tombe en panne, on finira toujours par le détecter.

Relacher la propriété de sûreté : Enfin, plutôt que de restreindre le modèle considéré, reste la solution de limiter le problème. Or, du point de vue des hypothèses temporelles c'est la clause de sûreté qui pose problème. C'est donc cette propriété

qui doit être simplifiée. Ainsi, certains des algorithmes étudiés considèrent une unicité du jeton à terme et non permanente. Il est alors possible de régénérer le jeton sur une simple suspicion de perte, en s'assurant de bien finir par faire disparaître l'un des deux jetons présents en cas de fausse suspicion.

3.3 Choix du modèle

Comme on vient de le voir, il convient de bien définir les hypothèses posées sur le système et le type de défaillance considérée. Dans ce chapitre, nous avons choisi de travailler sur les hypothèses suivantes :

- Système réparti statique composé de n nœuds munis d'un identifiant unique.
- Une communication par envoi de messages sur des canaux bidirectionels non-FIFO fiables formant un graphe complet.
- Un système synchrone, où la vitesse relative d'exécution et les délais de transmission des messages sont bornés, ces bornes étant connues.
- Des pannes franches pouvant frapper n'importe lequel des nœuds du systèmes sans contrainte supplémentaire de cardinalité, autre que celle du nombre initial de sites ($\#\text{fautes} \leq n-1$).

Remarquons que dans ces hypothèses n'apparaît pas celle du type de mémoire (*stable* ou *volatile*). En effet, nous ne considérerons ici que des pannes franches. D'autre part, ne figure pas non plus dans nos hypothèses celle de la disponibilité d'un service de diffusion de messages. Même si l'un des objectifs sera de minimiser l'emploi de la diffusion pour pouvoir s'adapter aux systèmes exempts de ce mécanisme, nous considèrerons séparément les deux types de système (avec ou sans diffusions) lors des études de performance.

Enfin, notons que parmi les cinq hypothèses permettant d'assurer la propriété de sûreté (voir section 3.2.2), nous avons choisi de ne retenir que celle d'un système *synchrone*. Ce choix s'explique par les considérations suivantes :

- Dans un premier temps nous avons écarté l'hypothèse d'*absence de perte du jeton* qui autorise un relâchement des hypothèses de synchronisme. En effet cette hypothèse suppose une inéquité des sites devant les défaillances. Cette propriété de résistance aux défaillances devant être dynamique, elle nous est apparue comme difficilement implémentable dans un système à large échelle. Pour la même raison, nous avons aussi écarté l'hypothèse de pannes avec reprises, l'algorithme n'assurant pas le service d'exclusion mutuelle tant que l'ensemble des sites en défaillance dans le système n'ont pas redémarré.
- Nous avons ensuite choisi de ne considérer que le problème de l'exclusion mutuelle dans sa forme stricte : à tout moment il n'y a au plus qu'un seul jeton. Ce choix est un parti pris et relâcher la propriété de sûreté, peut être envisageable. Notons que pour être vraiment utile à l'application, un algorithme permettant la cohabitation temporaire de plusieurs jetons peut nécessiter un mécanisme supplémentaire permettant de faire de la réconciliation. Ce mécanisme dépend alors directement l'application.
- Enfin, nous n'avons pas choisi de considérer l'existence d'un détecteur de classe $\mathcal{T}$. L'existence d'un tel détecteur permet il est vrai de relâcher les hypothèses de syn-

chronisme sur le système. De plus si son implémentation demande des hypothèses temporelles fortes, celles-ci peuvent être fournies par un réseau parallèle dédié à cette surveillance. Mais le fait d'intégrer la surveillance directement dans l'algorithme, permet d'exploiter ces messages et de limiter la surveillance en fonction des informations disponibles dans l'algorithme.

3.4 Versions tolérantes aux défaillances de l'algorithme de Naimi-Tréhel

Parmi l'ensemble des classes d'algorithmes d'exclusion mutuelle à jeton, nous avons choisi de nous intéresser aux algorithmes utilisant une structure dynamique et plus particulièrement à ceux basés sur l'algorithme de Naimi-Tréhel [NT87b]. Ce choix s'est fait en continuité de celui des algorithmes à jeton, avec l'idée de privilégier la facilité de mise en œuvre dans des systèmes comprenant un grand nombre de participants. Ainsi, l'algorithme de Naimi-Tréhel présente :

- une faible complexité en nombre de messages ($\mathcal{O}(log(n))$).
- une absence d'utilisation de la diffusion, ce qui peut permettre sa mise en œuvre sur des réseaux de type WAN (Wide Acces Network).
- une adaptation dynamique à la charge permettant de favoriser les sites demandant souvent la section critique.

Si y on retrouve de façon exacerbée les qualités des algorithmes à jeton, l'algorithme de Naimi-Tréhel présente aussi la même vulnérabilité face aux défaillances. Or, comme on l'a vu à la section 3.2, plusieurs extensions ont été proposées pour remédier à cette limitation de l'algorithme original : [Mue01], [Era95] et [NT87a].

Celle de Frank Mueller [Mue01] repose sur l'ajout d'un mécanisme de surveillance de la présence du jeton, basé sur un anneau logique venant s'ajouter à l'arbre dynamique de l'algorithme de base. Cet anneau est aussi utilisé, à l'occasion, pour assurer l'unicité de la régénération du jeton. La limite de cet algorithme réside dans le nombre de défaillances considérées. En effet, après chaque défaillance l'anneau servant à la régénération du jeton doit être réparé, sous peine de voir disparaître la propriété de sûreté. Or dans l'algorithme de Naimi-Tréhel, certaines défaillances peuvent rester silencieuses. Dans ce cas il n'y a pas de réparation de l'anneau et donc pas d'autres défaillances possibles. Une solution à ce problème peut être la multiplication des anneaux, mais cela se fera aux dépens des performances.

Stéphane Eranian a lui proposé dans sa thèse [Era95], une version tolérante aux défaillances pour gérer la synchronisation des processus dans le micro-noyau *CHORUS* [MVF+92]. Si ces travaux permettent de confirmer le choix de l'algorithme de Naimi-Tréhel, ils restent très dépendants de la plate-forme. En effet, le recouvrement des défaillances utilise un ensemble d'agents construits autour du micro-noyau *CHORUS*.

La version tolérante aux défaillances, proposée par Mohamed Naimi et Michel Tréhel [NT87a] est elle basée sur un empilage de petits mécanismes reposant tous sur de la diffusion. Cette utilisation massive de la diffusion, tant pour la détection que pour le recouvrement, diminue un peu l'intérêt de l'algorithme original. D'autre part, comme on

le verra plus en détail à la section 4.4.4, le recouvrement de pannes dans cet algorithme engendre une perte des requêtes en cours. Ceci a non seulement pour effet d'augmenter le coût du recouvrement mais engendre aussi un défaut d'équité. Cependant, le modèle considéré, l'absence d'ajout de nouvelles structures et la conservation du dynamisme initial en font tout de même une solution intéressante.

C'est au vu de cet état de l'art qu'il nous est paru intéressant de proposer une nouvelle version de l'algorithme de Naimi-Tréhel. Dans les sections qui suivent nous allons donc étudier plus précisément l'algorithme original [NT87b] et sa version tolérante aux défaillances [NT87a]. Nous présenterons ensuite notre proposition avant d'en évaluer les performances en la comparant à la solution proposée par Mohamed Naimi et Michel Tréhel.

Chapitre 4

Algorithme équitable d'exclusion mutuelle tolérant aux défaillances

Sommaire

4.1 Algorithme de Naimi-Tréhel

L'algorithme à jeton de Naimi-Tréhel[NT96] (présenté figure **??**) repose sur le maintien de deux structures logiques :

L'arbre des *LAST* **:** un arbre logique dynamique qui sert à acheminer les demandes d'entrée en section critique.

La chaîne des *NEXT* **:** une file d'attente distribuée contenant l'ensemble des sites attendant l'accès à la section critique. Le site possédant le jeton constitue le "début" de cette file d'attente et en est, en absence de requête, son unique membre.

Notons que l'on trouve d'autres dénominations pour l'arbre des *LAST*. Ainsi suivant les articles, il a pu s'appeler *father*, *owner* ou encore *probable owner*. Si l'on a choisi ici de retenir le nom de *LAST*, c'est qu'il reflète bien le point de vue de chacun des sites sur cet arbre : sur chaque site, la variable *LAST* contient l'identité du site lui ayant émis la *dernière* requête d'entrée en section critique.

```
1  Initialisation ()
2      if S_i ≠ S_0 then
3          last ← S_0
4      else
5          last ← NIL
6      next ← NIL
7      requesting ← FALSE

8  RequestCS ()
9      requesting ← true
10     if last ≠ NIL then
11         Send ⟨REQ, S_i⟩ to last
12         last ← NIL
13         Wait for token
14     /* Enter critical section */

15 ReleaseCS ()
16     requesting ← false
17     if next ≠ NIL then
18         Send ⟨TOKEN⟩ to next
19         next ← NIL

20 ReceiveREQ (S_j)
21     if last = NIL then
22         if requesting = true then
23             next ← S_j
24         else
25             Send ⟨TOKEN⟩ to S_j
26     else
27         Send ⟨REQ, S_j⟩ to last
28     last ← S_j

29 ReceiveToken ()
30     /* End of the bloking call */
```

FIGURE 4.1 – Algorithme original de Naimi-Tréhel

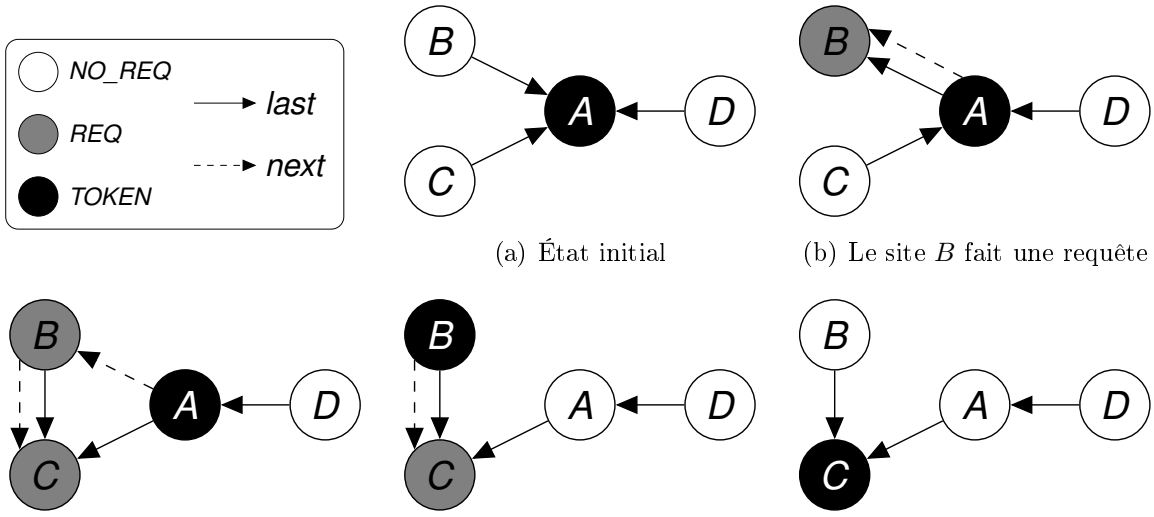
Initialement, tous les sites ont un *NEXT* vide et le propriétaire du jeton est la racine de l'arbre des *LAST* : il est le *LAST* de tous les autres sites (algorithme 4.1, ligne 1 à 7). Lorsqu'un site demande le jeton, sa requête est transmise de site en site, suivant l'arbre des *LAST* jusqu'à en atteindre la racine. À chaque réception de requête, les sites repositionnent leur *LAST* pour désigner le site demandeur (l'arbre est modifié dynamiquement). Lorsque

la requête arrive à la racine, celle-ci peut posséder le jeton sans pour autant être en section critique. Dans ce cas, elle l'envoie au site ayant émis la requête (algorithme 4.1, ligne 25). Dans le cas contraire, elle positionne sa valeur *NEXT* pour pointer vers le site demandeur (algorithme 4.1, ligne 23). Alors lorsqu'elle aura fini d'exécuter sa section critique, elle transmettra le jeton à son *NEXT*.

4.1.1 Exemple d'exécution de l'algorithme de Naimi-Tréhel

Pour illustrer cet algorithme, nous allons dérouler l'exemple présenté dans les figures 4.2. Initialement le site A possède le jeton (figure 4.2(a)) et l'ensemble des autres sites font pointer leur variable $LAST$ sur ce dernier. Puis sur la figure 4.2(b), le site B envoie une requête à son $LAST$ pour accéder à la section critique. À la réception de cette requête, le site A change son $NEXT$ et son $LAST$ pour pointer vers B (algorithme 4.1, lignes 23 et 28). Ainsi B devient la nouvelle racine. On suppose maintenant que le C désire à son tour entrer en section critique (figure 4.2(c)). Il envoie alors lui aussi une demande d'entrée en section critique au site A. Mais puisque A a déjà accepté la requête de B, il retransmet la requête de C à son $LAST$ (algorithme 4.1, ligne 27) qui prend B comme $NEXT$. Au passage, les sites A et B prennent le site C comme nouveau $LAST$: il est le dernier à leur connaissance à avoir émis une requête.

La chaîne des $NEXT$ contient donc maintenant les sites A, B et C. Ces derniers accéderont tour à tour à la section critique, chacun d'eux envoyant le jeton à leur $NEXT$ (algorithme 4.1, ligne 18) une fois leur code concurrent exécuté (figures 4.2(d) et 4.2(e)).



(a) État initial (b) Le site B fait une requête

(c) Le site C fait une requête (d) Le site A libère le jeton (e) Le site B libère le jeton

FIGURE 4.2 – Exemple d'exécution de l'algorithme de Naimi-Tréhel

Cet exemple permet de voir comment, l'algorithme de Naimi et Tréhel s'adapte dynamiquement à la fréquence des requêtes de chacun des sites. En effet, les sites inactifs du

point de vue de l'exclusion mutuelle, représentés ici par le site D, se retrouvent rapidement dans les couches basses de l'arbre des *LAST*. Ils sont alors naturellement mis à l'écart, diminuant d'autant la complexité moyenne par requêtes. À l'inverse, les sites demandant fréquemment la section critique restent assez proches de la racine. Ils bénéficient alors d'une complexité réduite pour leur accès à la section critique.

Les gains de complexité et de latence d'obtention sont donc obtenus grâce à la dynamicité des structures. Le prix à payer est celui du maintien de l'invariant :

Invariant 1. *À terme la racine de l'arbre des LAST est aussi l'extrémité de la chaîne des NEXT.*

Cet invariant est trivialement vérifié à l'état initial. La difficulté de sa préservation vient de la dynamicité des structures. En effet, il n'est réellement vérifié qu'en absence de toute requête pendante. Le reste du temps, ces messages en transit forment des connexions virtuelles tant pour l'arbre des *LAST* que pour la chaîne des *NEXT*. Ce point délicat est essentiel pour bien comprendre la suite de ce chapitre. Nous allons donc nous y arrêter plus avant.

4.1.2 Difficultés engendrées par le dynamisme

Supposons un arbre des *LAST* quelconque, où l'ensemble des sites ne sont pas intéressés par la section critique. Lorsqu'un site appelle la fonction *RequestCS()*, celle-ci vide sa variable *LAST* (algorithme 4.1, ligne 12). Le site et tous ses descendants dans l'arbre se retrouvent donc séparés de son ancien père pour former un nouvel arbre des *LAST*. À cet instant on observe donc une forêt, composée de deux arbres des *LAST*. Ces arbres ne sont reliés que par la requête émise à la ligne 11 de l'algorithme 4.1.

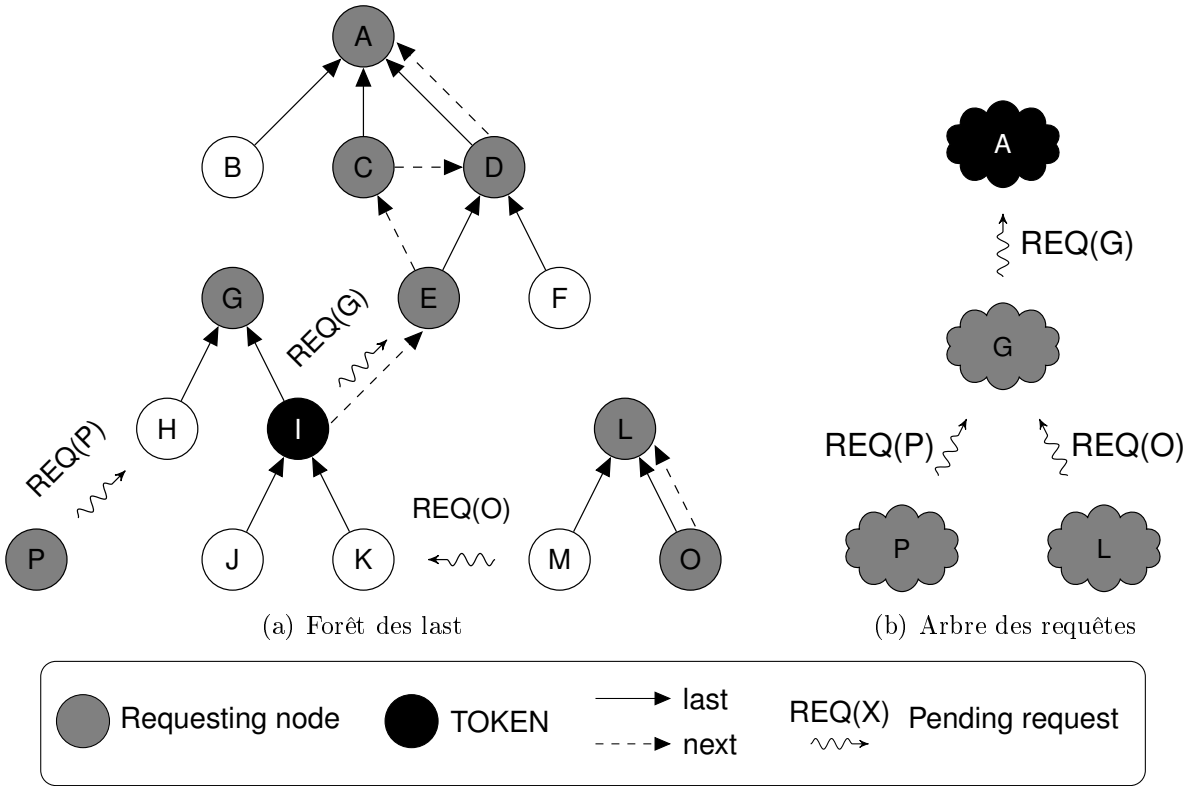
La réception de cette requête par l'ancien père du site demandeur n'implique pas forcément le retour à un unique arbre des *LAST*. En effet, si ce dernier n'est pas l'ancienne racine de l'arbre, il va se rattacher au nouvel arbre et transmettre la requête à son propre *LAST* (algorithme 4.1, lignes 23 et 28). On observe donc toujours la présence de deux arborescences reliées par un message en transit. Cette situation ne prend fin qu'à la réception de la requête par l'ancienne racine du système.

Si cet exemple d'exécution ne comprend qu'une seule requête, l'algorithme de Naimi-Tréhel peut très bien comporter de nombreuses requêtes concurrentes. Chacune d'elles génère temporairement un arbre de *LAST*. Ainsi, pour un système comprenant n requêtes en transit, la forêt des *LAST* sera composée de $n + 1$ arbres.

Cette dynamicité de l'algorithme complexifie aussi la structure de la chaîne des *NEXT*. En effet, tout site du système, attendant la section critique et n'ayant pas encore de *LAST* (algorithme 4.1, ligne 21) peut enregistrer une requête, *i.e.*, prendre son émetteur comme nouveau *NEXT*. Autrement dit, toutes les racines de la forêt des *LAST* sont susceptibles d'enregistrer une requête et donc de former une chaîne de *NEXT*. Il n'y a donc pas une unique chaîne des *NEXT* mais autant de morceaux de chaîne que d'arbres dans la forêt des *LAST*.

L'exemple présenté dans la figure 4.3, montre un état possible des structures de l'algorithme de Naimi-Tréhel. Pour des raisons de place, nous y avons limité le nombre de

sites n'attendant pas la section critique (en blanc). Cette figure comporte donc proportionnellement un nombre important de sites en attente de la section critique (colorés en gris).



(a) Forêt des last (b) Arbre des requêtes

FIGURE 4.3 – Exemple d'état global de l'algorithme

La *forêt des LAST* (figure 4.3(a)) correspondant à cet exemple est composée de quatre arbres. L'un d'eux est réduit à sa racine (le site P) ; il s'agit d'une ancienne feuille venant d'émettre une requête. Remarquons que leurs racines (A, G, P et L) correspondent bien aux extrémités des différentes files des *NEXT*. Le sommet P n'ayant reçu aucune requête, sa file des *NEXT* est réduite à lui même.

D'autre part, on voit sur cet exemple que les trois requêtes en transit ($REQ(G)$, $REQ(P)$ et $REQ(O)$) ne relient pas directement entre eux les morceaux de la chaîne des *NEXT*. En revanche, elles connectent entre eux les quatre arbres des *LAST* de manière à former un arbre. On peut voir cette arborescence sur la figure 4.3(b), où les 4 arbres des *LAST* sont formalisés par des nuages.

Notons pour terminer que la dynamicité de l'algorithme peut amener la chaîne des *NEXT* à *" serpenter"* entre les arbres des *LAST*.

On voit donc qu'une situation ponctuelle de l'algorithme de Naimi-Tréhel est beaucoup plus complexe que ne le laissait à penser l'invariant 1. Plutôt que de travailler sur cet invariant *"à terme"*, il peut être intéressant d'avoir des invariants *"instantanés"*. On définit alors les quatre invariants suivants :

Invariant 2. *si le système comporte n messages de requêtes en transit, l'algorithme est composé de $n+1$ arbres des LAST appellés* ***forêt des last****.*

Invariant 3. *Toutes les racines des arbres de la forêt des LAST sont des extrémités de chaîne de NEXT et réciproquement.*

Invariant 4. *L'ensemble des requêtes en transit forme un arbre, dont les nœuds sont composés par l'ensemble des arbres de la forêt des LAST, appellé* ***arbre des requêtes****.*

Invariant 5. *L'émetteur d'une requête en transit et l'extrémité de sa chaîne de NEXT se trouvent dans le sous arbre des requêtes ayant pour racine le dernier site à avoir émit ou ré-émit cette requête.*

Invariant 6. *La racine de l'arbre des requêtes est l'arbre des LAST dont la racine est l'extrémité de la chaîne des NEXT ayant comme origine le possesseur du jeton.*

Les figures 4.3(a) et 4.3(b) illustrent bien l'invariant 6. Le site A est la racine de la racine de l'arbre des requêtes. Ce site est aussi l'extrémité finale de la chaîne des *NEXT* issue du possesseur du jeton (I). Il important de remarquer que le site possédant le jeton (I) n'appartient pas forcément à l'arbre de racine A.

4.2 Impact des défaillances sur les structures logiques

Assurer la tolérance aux défaillances revient à maintenir les invariants définis précédemment. L'impact des pannes sur ces derniers varie suivant leurs emplacements dans les structures logiques de l'algorithme. On peut ainsi distinguer quatre types de panne suivant leurs conséquences pour l'algorithme. Cette taxonomie n'est pas exclusive et une défaillance peut avoir de multiples conséquences. La panne d'un site peut par exemple, provoquer la perte d'une requête et celle du jeton.

Le but de cette étude est de mettre en relief, non seulement ce qui ne marche plus dans l'algorithme, mais aussi ce qui continue de fonctionner en présence de panne(s) franche(s). Elle servira ensuite comme base pour étudier et comparer les mécanismes proposés pour rendre l'algorithme tolérant aux défaillances.

4.2.1 Pannes sans conséquence

Il s'agit des pannes survenant sur les feuilles des arbres des *LAST* qui n'attendent ou ne possèdent pas le jeton et qui ne sont pas les destinataires d'un message en transit. Par extension on peut aussi y associer les pannes des nœuds dont l'ensemble des descendants ne seront plus intéressés par la section critique. Les quatre invariants demeurent vérifiés quel que soit le nombre d'occurrences de ce type de défaillance. Ainsi, le nœud B de la figure 4.3(a) peut tomber en panne sans perturber le fonctionnement de l'algorithme pour le reste du système.

4.2.2 Rupture de l'arbre des *LAST* : Perte d'un message de requête

Des pertes de requêtes peuvent survenir lorsqu'elles sont (re)transmises à un nœud déjà en panne ou s'arrêtant pendant le traitement de ces dernières. La conséquence directe d'une telle perte est l'invalidation des invariants 2 et 4. La panne du site K de

la figure 4.3(a) provoque par exemple une cassure de l'*arbre des requêtes* (figure 4.3(b)) isolant l'arbre des *LAST* de racine L du reste des sites. Si aucun des sites de cet arbre ne peut accéder à la section critique, le système ne se bloquera pas pour autant. En effet, si l'on prive le système des sites L, M et O, les quatre invariants sont de nouveau respectés. De plus, les sites de l'arbre isolé peuvent émettre des requêtes. Ces requêtes seront enregistrées normalement dans la chaîne des *NEXT* de cet arbre : l'invariant 3 reste quant à lui valide.

4.2.3 Perte du jeton

Les pertes de jetons sont provoquées indifféremment par la panne du possesseur du jeton et par l'envoi du jeton à un site déjà arrêté. En tant que tel, ces disparitions n'entraînent que la perte de l'invariant 6. Même si elles ne peuvent plus être satisfaites, l'algorithme peut donc continuer à acheminer des requêtes vers l'extrémité de la chaîne des *NEXT*.

4.2.4 Rupture de la chaîne des *NEXT*

La chaîne des *NEXT* peut se retrouver coupée par la défaillance d'un de ses nœuds intermédiaires. Avec ce type de défaillance, le système ne s'arrête pas instantanément : des requêtes peuvent toujours être enregistrées et l'accès à la section critique est toujours possible. Cependant, elles conduisent toujours à la perte du jeton et donc à terme à l'arrêt du système.

4.3 Critères d'évaluation des algorithmes de Naimi-Tréhel tolérants aux défaillances

Comme nous venons de le voir, les conséquences des défaillances dans l'algorithme de Naimi-Tréhel sont multiples. L'ajout d'un mécanisme de tolérance aux pannes devra donc tenir compte de cette pluralité pour **leur détection comme pour leur recouvrement**. Tout en réparant les deux structures logiques, de façon à retrouver leurs invariants initiaux, ce mécanisme devra donc le cas échéant :

- ignorer les pannes *sans conséquence*.
- éviter de geler l'accès à la section critique s'il est toujours possible, c'est à dire s'il n'y a pas eu de *perte du jeton*.
- éviter autant que possible la perte du jeton et sa coûteuse régénération.
- permettre à l'ensemble des sites encore actifs d'émettre des requêtes dans les arbres des *LAST* exempts de pannes.

Ce faisant, une bonne version tolérante aux défaillances de l'algorithme de Naimi-Tréhel devrait aussi, **en absence comme en présence** de défaillances, conserver les qualités initiales de cet algorithme. À savoir :

- une faible complexité en nombre de messages .
- une limitation de l'utilisation de la diffusion.
- une dynamicité favorisant les sites les plus actifs.

– l'équité engendrée par le respect de l'ordre des requêtes enregistrées dans la file des *NEXT*.

4.4 Algorithme de Naimi-Tréhel tolérant les défaillances

Dans l'article [NT87b], Mohamed Naimi et Michel Tréhel proposent une version tolérante aux défaillances de leur algorithme [NT87a]. Cette nouvelle version conserve les mécanismes de l'algorithme sans les modifier. En absence de défaillance, seuls des messages nécessaires à la détection des défaillances s'ajoutent à ceux de l'algorithme original. Le recouvrement se fait par l'enchainement de mécanismes successifs, le nombre de ces mécanismes variant suivant le type de défaillance.

La figure 4.4 présente le pseudo code de cet algorithme [4]. Comme on peut le voir dans la fonction *Intialisation()* (ligne 1) les sites conservent les variables de l'algorithme initial, mais s'enrichissent aussi de trois nouvelles variables :

token : Dans l'algorithme original la présence du jeton inutilisé était déduite des variables suivantes : $requesting$ = FALSE et $last = NIL$. Autrement dit, une racine d'arbre des $LAST$ n'attendant pas le jeton le possède forcément. L'ajout d'une variable *token* positionnée à TRUE lorsque le site possède le jeton va permettre de détecter sa présence même si ce dernier est en cours d'utilisation.

state : Cette variable va servir à gérer les différentes étapes de la détection au recouvrement. Elle stocke l'état du site, lequel peut être l'un des trois état de l'automate classique de l'exclusion mutuelle (déjà présenté à la figure 2.1(a)) ou l'un des quatre états du recouvrement : $CONSULTING$, $QUERY$, $CANDIDATE$ et $OBSERVER$.

xc : Cette variable est un vecteur servant à gérer le problème d'un jetons en transit lors du recouvrement. Chaque site y stocke l'ensemble des sites qui cherchent ou cherchaient le jeton.

Enfin l'algorithme de Naimi-Tréhel utilise plusieurs temporisateurs :

TokenTimer : utilisé pour la détection de fautes, son dimensionnement est un compromis entre fausse détection et latence de recouvrement des pannes.

ConsultTimer, *FailureTimer* **et** *ElectionTimer* : utilisés lors des différentes étapes du recouvrement, le dimensionnement de ces temporisateurs est directement lié aux hypothèses de synchronisme. Ils correspondent à deux ou trois fois la borne maximum pour la transmission d'un message.

L'ajout de la gestion des temporisateurs est la seule différence notable dans les fonctions de base l'algorithme. Ainsi, un site voulant entrer en section critique continue d'émettre une requête à son $LAST$. Cependant, avant d'attendre l'arrivée du jeton, il arme un temporisateur *tokenTimer* (algorithme 4.4, ligne 18).

4. Pour des raisons de lisibilité nous avons uniformisé les pseudo-codes présentés dans cette thèse. Le pseudo code original de l'algorithme [NT87b], a donc été reformulé en s'attachant à ne pas en modifier le fonctionnement.

```
1  Initialisation ()
2      if S_i ≠ S_0 then
3          token ← FALSE
4          last ← S_0
5      else
6          token ← TRUE
7          last ← NIL
8      state ← NO_REQ
9      next ← NIL
10     requesting ← FALSE
11     xc ← ∅

12 RequestCS ()
13     requesting ← TRUE
14     state ← REQ
15     if last ≠ NIL then
16         Send ⟨REQ, S_i⟩ to last
17         last ← NIL
18         Alarm (TokenTimer)
19         Wait for token
20     /* Enter critical section */

21 ReleaseCS ()
22     requesting ← FALSE
23     state ← NO_REQ
24     if next ≠ NIL then
25         Send ⟨TOKEN⟩ to next
26         token ← FALSE
27         next ← NIL

28 ReceiveREQ (S_j)
29     if last = NIL then
30         if requesting = TRUE then
31             next ← S_j
32         else
33             Send ⟨TOKEN⟩ to S_j
34             token ← FALSE
35     else
36         Send ⟨REQ, S_j⟩ to last
37     last ← S_j

38 ReceiveToken ()
39     Reset alarm
40     token ← TRUE
41     if xc ≠ ∅ then
42         Send ⟨PRESENT, S_i⟩ to all sites in xc
43         xc ← ∅
44     /* End of the bloking call */

45 TimeoutToken ()
46     state ← CONSULTING
47     broadcast ⟨CONSULT, S_i⟩
48     Alarm (ConsultTimer)

49 ReceiveQuiet ()
50     if state = CONSULTING then
51         Reset alarm
52         state ← REQ
53         Alarm (TokenTimer)

54 ReceiveConsult (S_j)
55     if NEXT = S_j then
56         Send ⟨QUIET⟩ to S_j

57 TimeoutConsult ()
58     state ← QUERY
59     broadcast ⟨FAILURE, S_i⟩
60     Alarm (FailureTimer)

61 ReceiveFaillure (S_j)
62     switch state do
63         case NO_REQ or REQ or CS or CONSULTING
64             if token then
65                 Send ⟨PRESENT, S_i⟩ to S_j
66             else
67                 xc ← xc ∪ {S_j}
68         case OBSERVER
69             Alarm (ElectionTimer)

70 ReceivePresent (S_j)
71     if state = QUERY then
72         Reset alarm
73         last ← S_i
74         next ← NIL
75         RequestCS()

76 TimeoutFailure ()
77     state ← CANDIDATE
78     broadcast ⟨ELECTION, S_i⟩
79     Alarm (ElectionTimer)

80 ReceiveElection (S_j)
81     switch state do
82         case NO_REQ or REQ or CONSULTING or QUERY
83             state ← OBSERVER
84         case CANDIDATE
85             if S_j < S_i then
86                 state ← OBSERVER
87     if state = OBSERVER then
88         Alarm (ElectionTimer)

89 TimeoutElection ()
90     next ← NIL
91     last ← NIL
92     xc ← ∅
93     broadcast ⟨ELECTED⟩
94     if requesting then
95         RequestCS()
96     else
97         state ← NO_REQ

98 ReceiveElected (S_j)
99     Reset alarm
100    last ← NIL
101    next ← NIL
102    xc ← ∅
103    if requesting then
104        RequestCS()
105    else
106        state ← NO_REQ
```


FIGURE 4.4 – Version tolérante aux défaillances de l'algorithme Naimi-Tréhel

Si le site reçoit un message *TOKEN* avant la fin du temporisateur *TokenTimer*, il l'arrête (algorithme 4.4, ligne 39) et entre en section critique. Dans le cas contraire, la fin de ce temporisateur déclenche un appel à la fonction *TimeoutToken* (algorithme 4.4, ligne 45).

4.4.1 Traitement de la suspicion de défaillance

L'appel à la fonction *TimeoutToken* caractérise donc la suspicion d'une défaillance. Rappelons, que l'on ne fait aucune hypothèse sur la durée des sections critiques, il ne peut donc y avoir de borne sur le temps d'attente du jeton. Quel que soit le dimensionnement du temporisateur *TimeoutToken*, il ne peut s'agir d'une détection de défaillance mais uniquement d'une suspicion.

L'algorithme de Naimi-Tréhel étant basé sur l'utilisation d'une file d'attente répartie, l'idée est ici d'assurer localement la tolérance aux défaillances. Chaque site ne surveille que le fait d'être bien enregistré dans la file répartie, c'est à dire qu'il existe un autre site correct faisant pointer sa variable *NEXT* sur lui.

La fonction *TimeoutToken* va donc diffuser à l'ensemble des sites du système un message *CONSULT* (algorithme 4.4, ligne 47). Le site passe alors dans l'état *CONSULTING* et arme un nouveau temporisateur : *ConsultTimer*.

Parmi les autres sites du système, seul celui dont la variable *NEXT* pointe sur l'émeteur, répond au message *CONSULT* par un message *QUIET* (algorithme 4.4, ligne 56). Lorsqu'il reçoit cette réponse, le site peut retourner dans l'état *REQ*. Il arme alors un nouveau temporisateur *TokenTimer* et attend le jeton. À chaque fausse suspicion le coût est : une diffusion plus un message.

Supposons maintenant qu'aucun des sites du système n'envoie de réponse *QUIET*. Le temporisateur *ConsultTimer* va alors déclencher un appel à la fonction *TimeoutConsult()* (algorithme 4.4, ligne 57).

4.4.2 Traitement de la détection de la perte d'une requête

Après l'expiration d'un temporisateur *ConsultTimer*, le site considére que sa requête est perdue ou que son prédécesseur est défaillant. Il va donc chercher à ré-émettre sa requête. Avant, il doit vérifier que la panne ayant provoqué la perte de sa requête, n'a pas aussi provoqué *une perte du jeton*.

Le site diffuse alors à l'ensemble du système un message *FAILURE*, pour vérifier si le jeton est toujours présent (algorithme 4.4, ligne 59). Puis il passe de l'état *CONSULTING* à l'état *QUERY* et arme un temporisateur *FailureTimer*. Lorsqu'il reçoit ce message le possesseur du jeton, s'il existe, va lui répondre un message avec *PRESENT* (algorithme 4.4, ligne 65).

À la réception de ce message *PRESENT* (algorithme 4.4, ligne 70), le site choisit le possesseur du jeton comme nouveau *LAST* et lui renouvelle sa requête en lançant une nouvelle fois la fonction *RequestCS*(). De plus, pour éviter la création de boucle dans la chaîne des *NEXT*, la variable *NEXT* est ré-initialisée à NIL (algorithme 4.4, ligne 74). En effet, dans le cas contraire, on ne pouvait exclure le fait que cette nouvelle requête parvienne à l'extrémité de sa propre chaîne des *NEXT*.

Si cette solution a le mérite d'éviter la formation de boucle, elle reste très pénalisante pour le recouvrement puisque à l'issue de cette étape l'ancien *NEXT* (s'il existait) se retrouve à son tour sans prédécesseur. La procédure sera donc lancée récursivement jusqu'à disparition de l'ancienne chaîne des *NEXT*. La limite de ce **recouvrement individuel** est donc une addition des délais de recouvrement pour aboutir à un recouvrement global.

Revenons maintenant sur le temporisateur *FailureTimer* et sur son dimensionnement. Ici, contrairement à la détection des pertes de requêtes, l'algorithme ne peut tolérer une fausse détection de perte de jeton, l'unicité du jeton étant la base de la propriété de sûreté. Ce temporisateur est donc dimensionné à trois fois le temps maximum de transition d'un message et la variable xc est utilisée pour envoyer de façon différée un message *PRESENT*.

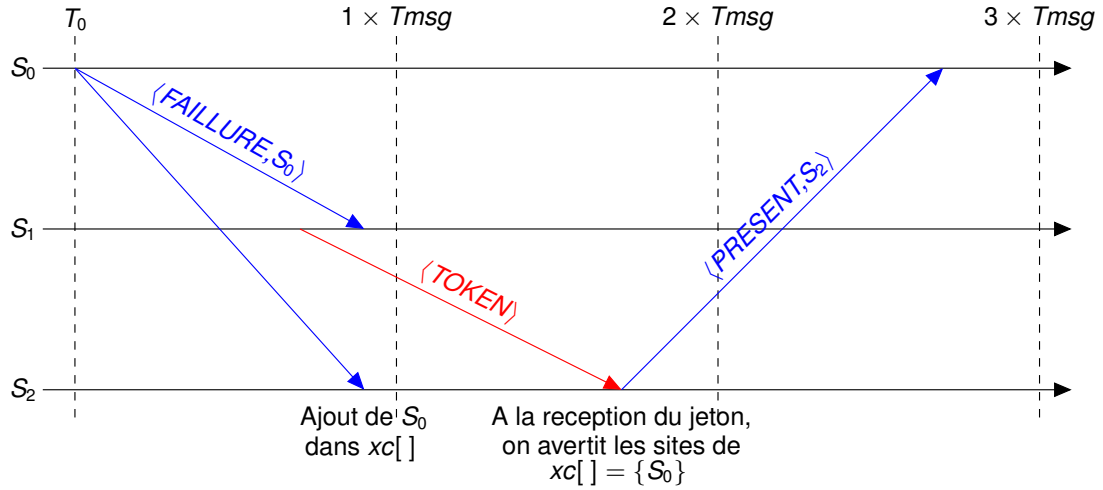


FIGURE 4.5 – Dimensionnement du timer *FailureTimer* à $3 \times T_{msg}$

Ainsi, lorsqu'un site qui ne possède pas le jeton reçoit un message *FAILURE*, il ajoute l'expéditeur à l'ensemble xc des sites qui cherchent le jeton (algorithme 4.4, ligne 67). Par la suite, et s'il est amené à recevoir le jeton, il enverra un message *PRESENT* à tous les sites qu'il aura enregistré dans son vecteur xc. L'exemple de la figure 4.5 illustre ce mécanisme et montre pourquoi le temporisateur *FailureTimer* se doit d'être dimensionné à trois fois le délais maximum de transmission.

4.4.3 Traitement de la détection de la perte du jeton

L'expiration du temporisateur *FailureTimer* correspond à une défaillance de type *"perte de jeton"* qui s'ajoute à la *"perte de requête"*. Son recouvrement passe par la regénération du jeton par l'un des sites du système et pour les autres par une ré-émission des requêtes perdues.

Tous les sites ayant détecté la *"perte du jeton"* diffusent à l'ensemble du système un message *ELECTION*, passent dans l'état *CANDIDATE* et arment un temporisateur *ElectionTimer* (algorithme 4.4, lignes 77 à 79).

À la réception d'un message *ELECTION*, les sites non candidats passent dans l'état *OBSERVER* quel que soit leur état présents(algorithme 4.4, ligne 83). Les sites candidats

passent eux aussi dans l'état $OBSERVER$ si l'identifiant de l'émetteur est plus petit que le leur (algorithme 4.4, ligne 86). Tous réarment alors leur temporisateur *ElectionTimer*. Une fois ce dernier écoulé il ne reste qu'un seul candidat. Celui-ci regénère le jeton et diffuse un message *ELECTED*. Tous les autres nœuds du système vont alors le prendre comme nouveau $LAST$. Dans le même temps, ils vident leur $NEXT$ et renouvellent leur requête s'ils étaient initialement en attente de la section critique (requesting = TRUE) (algorithme 4.4, lignes 98 à 98).

Si l'on considère les structures après la regénération, on a un arbre de hauteur 1 et plus aucune chaîne des $NEXT$. Tout se passe donc comme si l'on avait réinitialisé le système, l'ensemble des requêtes pendantes devant alors être ré-émises.

4.4.4 Limites de l'extension proposée par Naimi et Tréhel

Cette extension proposée par Mohamed Naimi et Michel Tréhel présente l'avantage d'être très peu intrusive dans l'algorithme original. Ceci lui permet de conserver la complexité initiale pour l'acheminement des requêtes et du jeton, ce qui était l'un des critères d'évaluation proposés à la section 4.3. D'autre part, en cas de perte du jeton, le mécanisme de ré-initialisation permet de retrouver facilement les invariants 2 à 6 nécessaires à la vivacité, ce qui simplifie grandement la preuve de l'algorithme. Enfin, cette extension reste bien **insensible aux défaillances n'ayant pas de conséquence** sur les structures de l'algorithme (voir section 4.2.1).

Elle souffre tout de même de quelques limitations en vue d'un passage à l'échelle. Ainsi, si l'on considère les autres critères proposés à la section 4.3, on s'aperçoit qu'ils sont peu ou prou satisfaits :

Dynamicité pendant le recouvrement : Si cette extension ne gèle pas l'algorithme original lors du recouvrement de défaillances, elle finit souvent par annuler les requêtes concurrentes au recouvrement. Si les sites peuvent continuer à émettre, à retransmettre et à accepter de nouvelles requêtes pendant les recouvrements, ceux-ci les conduisent à effacer leur $NEXT$ (algorithme 4.4, ligne 74) et donc à annuler les requêtes enregistrées.

Limitation des pertes de jeton : Dans cette extension la file des $NEXT$ est surveillée localement et unilatéralement, c'est-à-dire que chacun de ses membres veille à avoir un prédécesseur *correct* et se charge de ré-émettre une requête si tel n'est pas le cas. Ce comportement est particulièrement bien adapté aux défaillances entraînant une perte d'un message de requête. En revanche, il est inefficace face aux pannes survenant sur les sites de la chaîne des $NEXT$. En effet, dans ce dernier cas la ré-émission d'une requête ne règle pas le problème de la présence du site en panne au milieu de la chaîne des $NEXT$. Ce dernier se verra donc inévitablement envoyer le jeton : lequel sera alors perdu pour le système. Ce n'est qu'à la détection de cette perte, que la défaillance sera à proprement parlé corrigée par le mécanisme d'élection suivi d'une réinitialisation du système excluant les sites défaillants. Outre le fait qu'il engendre une perte de l'ensemble des requêtes émises et qu'il gèle le fonctionnement du système, ce mécanisme de recouvrement se montre extrêmement coûteux, tant en nombre de messages (4 séries de diffusion) qu'en

latence (trois expiration de temporisateur). Il aurait pu être avantageusement remplacé par un autre, permettant une réparation de la chaîne des *NEXT*, évitant ainsi la perte du jeton. Pour cela il aurait été nécessaire de discriminer la perte d'un message de requête de celle d'un site ayant enregistré une requête.

Conservation de l'équité de l'algorithme original : Comme on l'a vu, l'algorithme de Naimi-Tréhel utilise une file d'attente distribuée. Une fois enregistrée dans cette file, une requête a la garantie d'être satisfaite avant toutes celles qui pourraient lui succéder. Pour peu qu'on la préserve, la chaîne des *NEXT* offre donc à l'algorithme initial de bonnes propriétés d'équité. Une des limites de l'extension proposée par Mohamed Naimi et Michel Tréhel réside dans les multiples ré-émissions des requêtes. Qu'il s'agisse du recouvrement d'une perte de requête ou de celui de défaillances entraînant la perte du jeton, les morceaux de chaîne de *NEXT* encore intacts sont systématiquement détruits, pour être reconstruits à l'aide de ces ré-émissions. Un site n'ayant pas la garantie de retrouver sa place dans la chaîne des *NEXT*, il en résulte à chaque fois une perte de l'ordre des requêtes déjà émises. Plus que la disparition des bonnes propriétés de l'algorithme initial, cette perte de l'équité a pour conséquence une limitation dans les garanties de vivacité. Ainsi, dans le pire cas (tous les sites en attente), l'absence de famine individuelle (voir *équité faible* à la section 2.2.3) n'est garantie que si le système possède régulièrement des périodes $\mathcal{T}_{fault\_free}$ exemptes de défaillance d'une durée permettant à l'ensemble des sites d'accéder à la section critique : $\mathcal{T}_{fault\_free} = n(T_{msg} + T_{CS})$. Cette durée de stabilité dépend directement du nombre de participants, ce qui peut poser des problèmes de passage à l'échelle. En pratique, cette limitation pourrait être levée en considérant, non pas l'ensemble des sites, mais uniquement la taille de la file d'attente. En revanche, $\mathcal{T}_{fault\_free}$ pose le problème d'être proportionnelle à la durée d'accès à la section critique (T_{CS}), durée qui peut être longue et qui n'est pas forcément connue a priori.

Limitation de l'usage de la diffusion : L'un des intérêts majeurs de l'algorithme de Naimi-Tréhel est son indépendance vis-à-vis de l'existence d'un service de diffusion. C'est d'ailleurs l'une des raisons qui avait motivé notre choix. Or, dans l'extension que nous venons d'étudier, chacune des étapes du recouvrement des défaillances utilise ce type de service. Plus gênant encore, elle utilise un mécanisme de surveillance lui aussi basé sur la diffusion. Comme on le verra dans l'étude de performance, ce dernier point pose des problèmes pour le dimensionnement des temporisateurs de détection. Un des gros défauts de l'extension proposée par Mohamed Naimi et Michel Tréhel est donc la perte de cette qualité initiale de l'algorithme en présence comme en absence de défaillance.

4.5 Algorithme équitable tolérant aux défaillances

Au regard de cette étude, l'extension proposée par Mohamed Naimi et Michel Tréhel pose des problèmes rendant difficile sa mise en œuvre dans des systèmes à grande échelle. Nous allons donc présenter dans cette section une nouvelle extension qui, tout en conservant les mêmes hypothèses de travail, devra permettre un déploiement sur un grand nombre de sites tout en conservant l'équité de l'algorithme original.

4.5.1 Description de l'algorithme

Le principe guide de notre algorithme est de reconstruire la file des requêtes (chaîne des *NEXT*) partitionnée par les défaillances, en ré-assemblant dans l'ordre les morceaux encore intacts. C'est uniquement lorsque cela est impossible, que des requêtes d'accès à la section critique sont ré-émises. L'arbre des *LAST* est alors reconstruit en conséquence.

Contrairement à l'extension présentée précédemment, nous avons choisi ici de modifier légèrement le comportement originel de l'algorithme de Naimi-Tréhel. Ainsi, dans sa version initiale, les sites ignorent tout de leurs prédécesseurs dans la file d'attente répartie. Cette connaissance unilatérale de la chaîne des *NEXT* est à l'origine de multiples diffusions dans l'extension proposée par Naimi-Tréhel. Aussi pour informer chaque site de ses prédécesseurs, nous avons ajouté à l'algorithme initial un mécanisme d'acquittement des requêtes pour permettre de véhiculer l'identité de leurs prédécesseurs.

Afin de gérer les informations contenues dans ces acquittements, nous avons pourvu notre algorithme, dont le pseudo code est présenté à la figure 4.6, de deux variables supplémentaires. Ainsi, tout en continuant à maintenir les variables *LAST* et *NEXT*, chacun des sites du système se verra doter des variables suivantes :

pred $[0..k-1]$ **:** Cette variable est un vecteur servant à enregistrer l'identité et les positions [5] respectives des k prédécesseurs d'un site dans la file des *NEXT*, k étant un paramètre de l'algorithme.

pos **:** Cette variable est un entier positif qui sert à maintenir, pour chaque site de la chaîne des *NEXT*, sa position courante dans cette file d'attente répartie. Gérée de façon incrémentale, le nœud détenant le jeton possède la position la plus petite et le dernier site inséré dans la chaîne possède la plus grande. Un site n'attendant pas la section critique et ne possédant pas le jeton voit sa position fixée à -1.

Comme pour l'extension de Naimi-Tréhel, nous introduisons plusieurs types de temporisateur. On distinguera plus particulièrement deux catégories :

temporisateurs de détection (*CommitTimer* et *TokenTimer*) : Contrairement à l'extension de Naimi-Tréhel, nous utiliserons deux temporisateurs différents pour détecter respectivement la perte d'une requête ou la perte du jeton. Leur dimensionnement est indépendant et résulte d'un compromis entre fausses détections et délais de recouvrement des pannes.

temporisateurs de recouvrement (*ReconnectionTimer*) : commun à tous nos mécanismes de recouvrement, son dimensionnement correspond à deux fois la borne maximum de transmission d'un message de façon à garantir l'absence de fausse détection.

4.5.2 Gestion des requêtes et acquittement

L'acheminement des requêtes demeure identique à l'algorithme original. Ainsi, celles-ci continuent à être émises et re-transmises le long de l'arbre des *LAST* ; arbre qui reste mis à jour à chaque réception (algorithme 4.6, lignes 34 et 35). Ce n'est qu'une fois la racine de cet arbre atteinte que notre mécanisme d'acquittement se met en route. Comme dans

5. Pour simplifier l'écriture du pseudo code, nous accéderons à la position du x^{me} prédécesseur par $pred[x].pos$

```
1  Initialisation ()
2    if S_i ≠ S_0 then
3      last ← S_0
4      pos ← -1 ;
5    else
6      last ← NIL
7      pos ← 0
8    next ← NIL
9    requesting ← FALSE

10 RequestCS ()
11   requesting ← true
12   if last ≠ NIL then
13     Send ⟨REQ, S_i⟩ to last
14     last ← NIL
15     Alarm (CommitTimer)
16     Wait for ⟨TOKEN⟩
17   /* Enter critical section */

18 ReleaseCS ()
19   requesting ← FALSE
20   if next ≠ NIL then
21     Send ⟨TOKEN, S_i, pred[ ], pos⟩ to next
22     next ← NIL
23     pos ← -1

24 ReceiveRequest (S_j)
25   if last = NIL then
26     if requesting = TRUE then
27       next ← S_j
28       if pos ≠ -1 then
29         Send ⟨COMMIT, S_i, pred[ ], pos⟩ to S_j
30     else
31       Send ⟨TOKEN, S_i, pred[ ], pos⟩ to S_j
32       pos ← -1
33   else
34     Send ⟨REQ, S_j⟩ to last
35   last ← S_j

36 ReceiveToken (S_j, pred_j[ ], pos_j)
37   ReceiveCommit(S_j, pred_j[ ], pos_j)
38   Reset alarm

39 ReceiveCommit (S_j, pred_j[ ], pos_j)
40   pred[..] ← [S_j, pred_j[1], .., pred_j[k − 1]]
41   if pos = -1 then
42     pos ← pos_j + 1
43     if NEXT ≠ NIL then
44       Send ⟨COMMIT, S_i, pred[ ], pos⟩ to NEXT
45   Alarm (TokenTimer)

46 TimeoutCommit ()
47   newPred.id ← NIL
48   newPred.next ← NIL
49   newPred.pos ← -1
50   last ← NIL
51   next ← NIL
52   broadcast ⟨SEARCH_QUEUE, S_i⟩
53   Alarm (ReconnectionTimer)

54 TimeoutToken ()
55   check the liveness of pred[1]
56   if pred[1] failed then
57     if ∃ a minimal x, pred[x] is alive then
58       Send ⟨CONNECTION, S_i, pred[x].pos⟩ to pred[x]
59       Alarm (TokenTimer)
60     else
61       newPred.id ← NIL
62       newPred.pos ← -1
63       broadcast ⟨SEARCH_POSITION,pos,pred[]⟩
64       Alarm (ReconnectionTimer)
65   else
66     Alarm (TokenTimer)

67 ReceiveSEARCH_POSITION (S_j, pos_j, faultyPred[])
68   if pos ≠-1 ∧ pos < pos_j then
69     Send ⟨POSITION, S_i, pos, next⟩ to S_j
70   if ( last ∈ faultyPred[] ∧ not requesting) then
71     last ← S_j

72 ReceivePOSITION (S_j, pos_j, next_j)
73   if newPred.pos < pos_j then
74     newPred.id ← S_j
75     newPred.pos ← pos_j
76     newPred.next ← next_j

77 TimeoutReconnection ()
78   if newPred.id = NIL then
79     if pos = -1 then
80       Pred[ ] ← ∅
81       pos ← 0
82     Regenerate new Token
83   else
84     if newPred.next ≠ NIL then
85       Send ⟨CONNECTION, S_i, newPred.pos⟩ to
         newPred.id
86     else
87       Send ⟨REQ, S_i⟩ to newPred.id
88     Alarm (CommitTimer)

89 ReceiveCONNECTION (S_j,pos_old)
90   if pos = pos_old then
91     next ← S_j
92     Send ⟨COMMIT, S_i , pred[ ], pos⟩ to S_j
93   else
94     Send ⟨TOKEN, ∅, 0⟩ to S_j

95 ReceiveSEARCH_QUEUE (S_j)
96   if S_j is a best candidate for the current election then
97     if pos ≠-1 then
98       Send ⟨POSITION, S_i, pos, next⟩ to S_j
99     if requesting ∧ pos = -1 then
100      next ← NIL
101      last ← NIL
102      Send ⟨REQ, S_i⟩ to S_j
103      Alarm (CommitTimer)
104    if last ≠ NIL then
105      last ← S_j
```

FIGURE 4.6 – Notre version tolérante aux défaillances de l'algorithme Naimi-Tréhel

l'algorithme initial, on distingue alors deux cas suivant que la racine attend (ou exécute) la section ou qu'elle possède le jeton sans pour autant l'utiliser.

Dans le premier cas, avant d'enregistrer l'émetteur de la requête comme nouveau *NEXT*, la racine lui envoie un message *COMMIT* pour l'avertir de la bonne réception de sa requête (algorithme 4.6, ligne 29). Si ce message d'acquittement permet d'avoir une file d'attente doublement chainée, nous l'enrichissons de champs supplémentaires afin d'étendre ce chaînage sur plusieurs niveaux. La racine joint donc au message *COMMIT* :

- Son vecteur *Pred*[ ] contenant les identifiants de ses prédécesseurs dans la chaîne de *NEXT*. À la réception du message, l'émetteur de la requête met à jour son propre vecteur *Pred*[ ] en enregistrant l'émetteur de l'acquittement (son prédécesseur direct) suivi de ses $k-1$ prédécesseurs [6] (algorithme 4.6, ligne 40).
- La valeur de sa variable *pos*. Pour connaître sa propre position dans la chaîne des *NEXT*, le site ayant émis la requête incrémentera cette valeur de 1 (algorithme 4.6, ligne 42).

Dans le deuxième cas la racine n'utilise pas le jeton, celle-ci va joindre au message *TOKEN* sa position et la liste de ses prédécesseurs (algorithme 4.6, ligne 31). La réception du jeton tient alors lieu d'acquittement et la partie du message contenant la position et les prédécesseurs est traitée comme telle, avec un appel à la fonction *ReceiveCommit*() (algorithme 4.6, ligne 37). Notons que dans ce dernier cas, il n'y a plus à proprement parler de prédécesseur dans la file d'attente puisque le site possède le jeton. La liste reçue correspond aux sites ayant exécuté précédemment la section critique. Si cette information est inutile pour réparer la chaîne des *NEXT*, elle trouvera tout son sens en cas de perte du jeton, simplifiant alors grandement la régénération de celui-ci (voir section 4.5.3).

Le mécanisme d'acquittement présenté ici est incrémental. Chaque site de la chaîne des *NEXT* utilise les informations reçues dans le message *COMMIT* pour envoyer à son tour un acquittement. Le coût de ce mécanisme n'est alors que d'un message supplémentaire par requête, quel que soit la valeur du paramètre k choisie. La complexité de l'algorithme reste donc en $\mathcal{O}(log(n))$. Le dimensionnement du nombre de prédécesseurs k sera étudié expérimentalement à la section 5.2, puis théoriquement à la section 5.3.

Notons enfin, que pour gérer le croisement du jeton et d'un acquittement dans un canal non FIFO, il faut ajouter sur chaque site un compteur de requête (*cmptReq*). La valeur de ce compteur, initialisée à zéro et incrémentée à chaque nouvelle demande d'entrée en section critique, sera jointe dans les messages de requête, dans les acquittements et dans le jeton. Un message contenant un numéro de requête trop vieux sera ignoré. Pour des raisons de lisibilité, nous avons choisi de ne pas représenter ce compteur et les tests associés dans le pseudo code de l'algorithme 4.6.

L'utilisation des temporisateurs de détection procède alors comme suit. Lorsqu'un site émet une requête d'entrée en section critique, il arme un temporisateur *CommitTimer* (algorithme 4.6, ligne 15) et attend l'arrivée d'un message *COMMIT*. Dès qu'il reçoit ce message, il désarme le temporisateur *CommitTimer*, pour lancer un temporisateur *TokenTimer*[7] (algorithme 4.6, ligne 45). Ce dernier temporisateur ne sera désarmé qu'à

6. Notons ici que l'envoi du dernier prédécesseur dans l'acquittement est inutile, mais cela permet de simplifier le pseudo-code.

7. Dans le pseudo code, l'appel à la fonction *alarm*() annule si besoin le temporisateur précédemment

la réception du jeton (algorithme 4.6, ligne 38). Ces deux temporisateurs permettent de différencier la perte du message de requête (absence d'acquittement) d'une panne touchant la chaîne des *NEXT* (non réception du jeton). Les mécanismes de recouvrement mis en œuvre diffèrent donc suivant le type de temporisateur expiré.

Dans un premier temps, nous allons étudier l'expiration d'un temporisateur *TokenTimer*. On considérera alors que le site demandant la section critique a bien reçu un message *COMMIT*. Il a donc non seulement reçu sa position dans la file d'attente mais aussi l'identité d'un certain nombre de ses prédécesseurs.

4.5.3 Traitement de la défaillance d'un prédécesseur

L'expiration d'un temporisateur *TokenTimer*, et donc l'appel à la fonction *TimeoutToken()* (algorithme 4.6, ligne 54), correspond à un retard dans l'obtention du jeton et entraîne un mécanisme de recouvrement que nous appelons *M1-SearchPred*. Le site suspecte alors une défaillance dans la chaîne des *NEXT*. Il ne va pas pour autant contrôler l'ensemble de cette file, mais va se contenter de vérifier l'état de son prédécesseur direct. Chaque site de la chaîne exécutant ce mécanisme, il s'établit un ensemble de surveillances locales. C'est la somme de ces surveillances qui réalise une surveillance globale de la file d'attente répartie.

La fonction *TimeoutToken()* interroge donc le prédécesseur du site. Si ce dernier répond, le site réarme le temporisateur *TokenTimer* et se remet en attente du jeton (algorithme 4.6, ligne 66). Le surcoût dû à une fausse suspicion de défaillance dans la chaîne des *NEXT* se limite donc à l'émission de deux messages.

À l'inverse, si le prédécesseur ne répond pas dans le temps imparti fixé par les hypothèses de synchronisme, le site passe de la simple suspicion à la détection d'une défaillance. Il lance alors un mécanisme de recouvrement. Tout comme la phase de surveillance, ce mécanisme est local. Chaque site se contente de se re-connecter à un prédécesseur. L'intégrité globale de la file résulte de l'ensemble de ces réparations locales.

La re-connexion d'un site S_i passe donc par la recherche du premier prédécesseur $Pred[x]$ *correct* : c'est-à-dire du plus proche (x minimal) prédécesseur non défaillant (algorithme 4.6, ligne 57).

Pour trouver de tels prédécesseurs, on peut envisager plusieurs stratégies. La première consiste à interroger successivement chacun des prédécesseurs connus, jusqu'à obtenir une réponse. Cette solution présente l'avantage d'être très peu coûteuse en messages : $f + 1$ messages pour f défaillances consécutives dans la chaîne des *NEXT*. La contre partie est un long temps de recouvrement : chaque étape engendre une latence égale à deux fois la borne de synchronisme T_{msg} soit $2fT_{msg}$. Pour minimiser ce temps de recouvrement, on peut mettre en place une stratégie qui vise à paralléliser la vérification des prédécesseurs. Le site émet alors un message à l'ensemble de ses prédécesseurs connus et se re-connecte au site le plus proche de lui dans la file d'attente, parmi l'ensemble des réponses reçues. Le temps de recouvrement est alors réduit à $2T_{msg}$ mais le nombre de messages dépend directement du nombre k (nombre de prédécesseurs connus). Une approche intermédiaire est alors possible. Elle consiste à interroger successivement un nombre toujours croissant

armé.

de prédécesseurs. On peut par exemple paralléliser la vérification d'un, puis de deux, puis de quatre prédécesseurs, et ainsi de suite jusqu'à trouver un prédécesseur *correct*. Le temps de recouvrement, dans le pire cas, est alors réduit à $2log(k)T_{msg}$. Tandis que le nombre de messages émis reste restreint en présence d'un petit nombre de fautes consécutives.

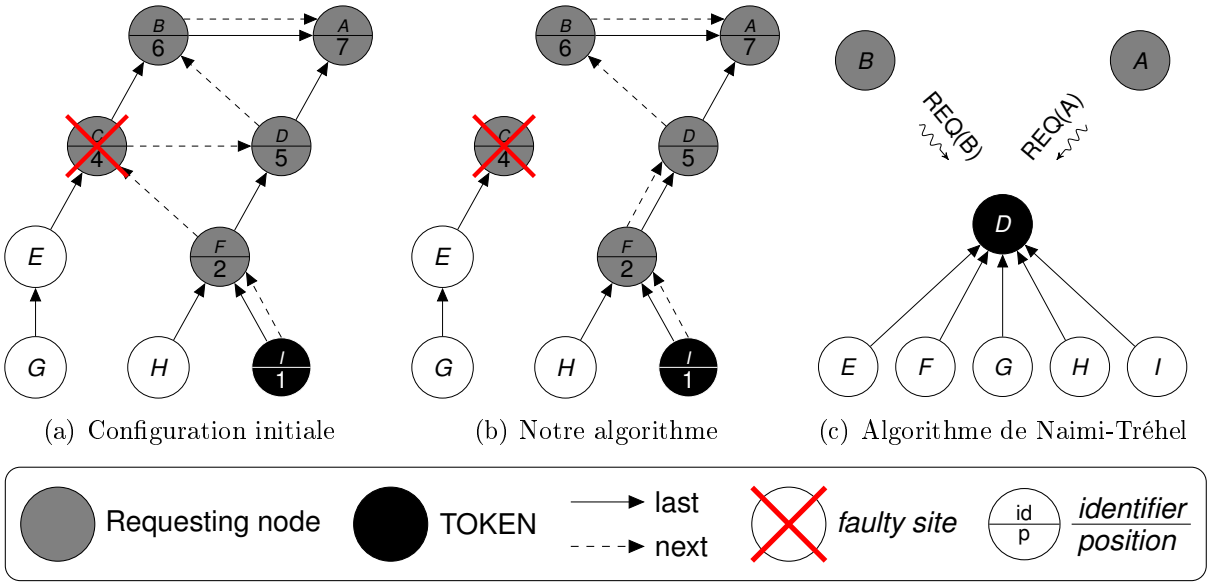
Une fois trouvée l'identité de son plus proche prédécesseur $Pred[x]$ *correct* (nous étudierons dans la section 4.5.4 le cas de la défaillance de l'ensemble des prédécesseur connus), le site S_i va pouvoir lancer la re-connexion. Celle-ci passe par l'envoi d'un message *CONNECTION* puis par le ré-armement du temporisateur *TokenTimer* utilisé pour détecter une nouvelle défaillance (algorithme 4.6, lignes 58 et 59). Si tel venait à être le cas, l'expiration du temporisateur relancerait la recherche d'un nouveau prédécesseur. Dans le cas contraire, le prédécesseur finit par recevoir le message *CONNECTION*. Il prend alors l'émetteur du message comme nouveau *NEXT* et lui envoie un message d'acquittement (algorithme 4.6, lignes 91 et 92). Le site S_i est alors re-connecté à la chaîne des *NEXT*. Notons pour terminer qu'un site ayant déjà obtenu une position, ne la mettra pas à jour lorsqu'il recevra ce message d' acquittement (algorithme 4.6, ligne 41). Ceci est un point important de notre algorithme : tous les mécanismes mis en œuvre assurent qu'un site ayant obtenu une position pour une requête, la conserve jusqu'à ce qu'il obtienne puis libère le jeton.

Cette propriété va d'ailleurs nous permettre de traiter le cas particulier où $Pred[x]$ a déjà accédé à la section critique. Dans ce cas, il aura envoyé le jeton à son *NEXT* qui n'est autre que l'un des prédécesseurs défaillants. Pour détecter cette perte du jeton, il suffit alors de joindre au message *CONNECTION* la position $Pred[x].pos$ reçue dans l'acquittement. À la réception de ce message, elle sera comparée à la position courante de $Pred[x]$. Toute différence entre ces deux valeurs caractérisera un accès à la section critique. En effet, ces valeurs ne peuvent différer que si le site a reçu puis libéré le jeton entre temps. Si tel est le cas, le site re-génère un nouveau jeton (algorithme 4.6, ligne 94).

Pour terminer l'étude de ce premier mécanisme de re-connexion, il nous faut remarquer que celui-ci **conserve intact l'ordre de la chaîne** des *NEXT*. En effet, tout se passe comme si l'on avait extrait les sites défaillants de la file d'attente. Cette propriété permet, d'une part, de conserver l'équité de l'algorithme initial et ,d'autre part, d'assurer la conservation des invariants liant la structure de l'arbre des *LAST* à la chaîne des *NEXT*. Le maintien de ces invariants permet d'éviter d'avoir à reconstruire l'arbre en fonction de la nouvelle chaîne, simplifiant d'autant la preuve de l'algorithme.

Exemple du mécanisme *M1-SearchPred* : La figure 4.7 montre un exemple de recouvrement utilisant le mécanisme *M1-SearchPred* et le recouvrement de la même défaillance par l'extension de Naimi-Tréhel. Dans cet exemple nous considérons la défaillance d'un site C appartenant à la chaîne des *NEXT* (figure 4.7(a)). La file des *NEXT* est donc brisée en deux morceaux : I,F et D,B,A. On suppose aussi que tous les sites présents dans la file ont déjà obtenu une position et attendent le jeton qui est détenu pour le moment par le site I. Enfin nous considérons ici que le paramètre fixant le nombre de prédécesseurs connus est supérieur à 1 ($k > 1$).

Compte tenu de la configuration initiale, le mécanisme *M1-SearchPred* finira par être lancé sur le site D à l'expiration du temporisateur *TokenTimer*. Ce dernier va alors dé-



(a) Configuration initiale (b) Notre algorithme (c) Algorithme de Naimi-Tréhel

FIGURE 4.7 – Exemple de recouvrement de défaillances : *M1-SearchPred*

tecter la défaillance de son prédécesseur direct (C) et par conséquent interroger ses autres prédécesseurs connus. Une fois reçue la réponse du site F, D va lui envoyer un message *CONNECTION*. D deviendra alors son nouveau *NEXT* achevant ainsi la re-connexion de la chaîne. La figure 4.7(b), représentant l'état final de ce recouvrement, met en évidence l'exclusion du site défaillant (C) de la file d'attente. Elle permet aussi de voir que ce site n'est pas pour autant complètement sorti de l'algorithme. En effet, ce mécanisme n'utilise pas de diffusion. Il n'est donc pas possible d'extraire complètement C de l'arbre des *LAST*. Ici, les sites E et G demeurent des descendants de C. Nous verrons à la section 4.5.5 comment ces derniers pourront se re-connecter globalement à l'arbre des *LAST*.

Comparons maintenant ce résultat à celui obtenu par l'algorithme tolérant aux défaillances de Naimi-Tréhel (figure 4.7(c)). Tout comme dans notre algorithme, le site D va finir par détecter l'absence de prédécesseur. Il va alors chercher le jeton puis renouveler sa requête, sans oublier d'effacer son *NEXT*, ce qui lancera récursivement la même opération chez A et B. S'ils permettent de détecter la défaillance, l'ensemble de ces mécanismes ne permet pas d'exclure le site C de la chaîne des *NEXT*. Le jeton finira alors par s'y perdre après que les sites I et F aient fini d'exécuter leur section critique (voir la section 4.4.4). Lorsque D détecte cette perte, il lance une élection puis re-génère un nouveau jeton. Tous les sites font alors pointer leur *LAST* vers D et positionnent leur *NEXT* à $VIDE$. Les sites n'ayant pas encore obtenu le jeton (A et B) vont pouvoir renouveler leur requête d'entrer en section critique auprès de D.

La comparaison de ces deux résultats de recouvrement met en évidence deux différences notables. Tout d'abord, notre algorithme peut permettre d'éviter la perte du jeton alors qu'il est systématiquement perdu avec l'autre approche (nous reviendrons sur l'énorme impact de cette propriété à la section 4.5.7). D'autre part, notre approche permet de conserver l'ordre des requêtes après le recouvrement. Le site B est donc assuré d'accéder à la section critique avant le site A. A contrario, l'extension de Naimi-Tréhel perd cette

notion d'ordre. On ne peut en effet exclure le fait que la nouvelle requête du site B soit reçue par le site D après celle du site A. Notre approche présente donc de bien meilleure qualité en termes d'équité.

4.5.4 Traitement de la défaillance de l'ensemble des prédécesseurs connus

Nous allons maintenant étudier le cas d'une défaillance généralisée des prédécesseurs connus. La seule information disponible pour le recouvrement est alors la position reçue avec l'acquittement de la requête. Le mécanisme présenté dans cette section, noté par la suite *M2-SearchPos*, va donc exploiter cette information pour re-connecter le site à la chaîne des *NEXT*. Comme pour le mécanisme précédent (*M1-SearchPred*), l'utilisation de la position va limiter toute reconstruction de l'arbre des $LAST$ en assurant la conservation de l'ordre de la file d'attente.

Afin de maintenir cet ordre, le site ayant détecté la défaillance recherche parmi les autres sites corrects, celui qui possède la plus grande position inférieure à la sienne (inférieur maximum). Ne pouvant se baser sur la connaissance de prédécesseurs, cette recherche va devoir utiliser un mécanisme de diffusion. Le site émet donc à l'ensemble du système un message *SEARCH_POSITION* (algorithme 4.6, ligne 63) contenant sa propre position ainsi que l'ensemble des prédécesseurs détectés comme défaillants par le mécanisme *M1-SearchPred*. Une fois ce message émis, il arme un temporisateur *ReconnectionTimer* et attend les messages de réponse (algorithme 4.6, ligne 64). Le dimensionnement de ce temporisateur est de deux fois la borne de synchronisme ($2T_{msg}$), ce qui assure au site de recevoir toutes les réponses qui lui auront été envoyées.

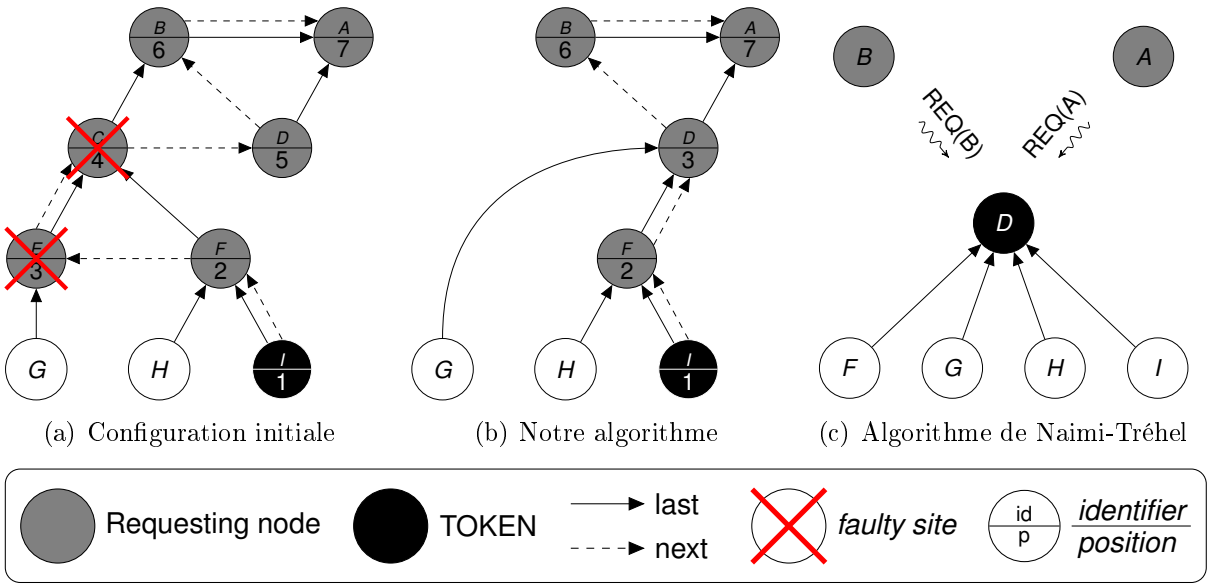
À la réception du message *SEARCH_POSITION*, chaque site compare sa propre position (s'il en possède une) à celle contenue dans le message. Seuls les sites possédant une position inférieure répondent alors par un message *POSITION* contenant leur position (algorithme 4.6, lignes 68 et 69). Le site ayant détecté la défaillance, enregistra ces réponses durant toute la durée du temporisateur *ReconnectionTimer* (algorithme 4.6, lignes 72 à 76). À l'expiration de celui-ci, il choisira parmi ces réponses le site qui a la plus grande position et lui enverra un message *CONNECTION*[8] (algorithme 4.6, ligne 85). Notons que ce message, ainsi que son traitement, sont identiques à ceux présentés dans la section précédente (4.5.3) pour le mécanisme *M1-SearchPred*.

Nous venons de voir comment le site recherche, puis se re-connecte à son nouveau prédécesseur. Cependant, cette recherche peut s'avérer vaine. Le site n'a alors reçu aucune réponse à son message *SEARCH_POSITION*. Dans ce cas, le dimensionnement du temporisateur *ReconnectionTimer* et les hypothèses de synchronisme lui assure d'être, parmi l'ensemble des sites corrects, celui qui possède la plus petite position. Or, comme nous le verrons dans la preuve (section 4.6), l'algorithme assure que le possesseur du jeton est aussi le site qui possède la plus petite position. On peut donc déduire de cette absence de réponse, que le jeton n'est plus présent dans le système et puisque le site a maintenant la plus petite position, il va pouvoir re-générer le jeton (algorithme 4.6, ligne 82).

8. Dans le cas du mécanisme *M2-SearchPos*, il est a noté que le prédécesseur trouvé a toujours un *NEXT*. Le test de la ligne 84, sert pour le troisième type de recouvrement.

Qu'il engendre une re-connexion à un prédécesseur ou une re-génération du jeton, le mécanisme *M2-SearchPos* présenté ici ne modifie en rien l'ordre des sites corrects présents dans la file des *NEXT* avant la défaillance. Tout comme pour le mécanisme *M1-SearchPred*, il n'est alors pas nécessaire de reconstruire l'arbre des *LAST* autour de la nouvelle chaîne. Cependant, nous avons choisi de profiter de la diffusion du message *SEARCH_POSITION*, pour exclure les prédécesseurs défaillants de l'arbre des *LAST*. À la réception de ce message, les sites qui n'attendent pas la section critique vérifient si leur *LAST* appartient à la liste des prédécesseurs défaillants contenue dans ce dernier. Si tel est le cas, ils prennent l'émetteur du message comme nouveau *LAST* (algorithme 4.6, ligne 71). Comme nous le verrons dans la preuve, cette nouvelle configuration respecte les invariants 2 à 6 définis à la section 4.1.2. Intuitivement, on peut remarquer que ce nouveau *LAST* est un successeur de l'ancien *LAST* dans la chaîne des *NEXT*. Tout se passe comme si le site avait vu passé la requête de son nouveau *LAST* après celle de son ancien.

Exemple du mécanisme *M2-SearchPos* : Tout comme dans la section précédente nous comparons sur une même configuration le résultat de ce mécanisme à celui obtenu par l'extension de Naimi-Tréhel. Nous considérons initialement la défaillance de deux sites consécutifs de la chaîne des *NEXT* : C et E (figure 4.8(a)). La file est donc brisée en deux morceaux (I,F et D,B,A). On suppose aussi que tous les sites présents dans la file ont déjà obtenu une position et que le site I est le propriétaire courant du jeton. Enfin, puisque nous voulons étudier le mécanisme *M2-SearchPos*, nous fixons le paramètre k à 2 prédécesseurs connus.



(a) Configuration initiale (b) Notre algorithme (c) Algorithme de Naimi-Tréhel

FIGURE 4.8 – Exemple de recouvrement de défaillances avec $k = 2$: *M2-SearchPos*

À l'expiration de son temporisateur *TokenTimer* et n'ayant plus de prédécesseurs connus non défaillants, le site D finit par diffuser un message *SEARCH_POSITION* incluant la valeur de sa propre position dans la file (5) et les identifiants des sites détectés

comme défaillants (E et C). Seuls les sites F et I dont les positions sont inférieures à 5 répondent à D. Profitant de ce message, les sites G et F dont le $LAST$ pointe vers l'un des sites fautifs, vont le corriger en le pointant directement vers D. À l'expiration de *ReconnectionTimer*, D choisit F comme prédécesseur, ce dernier étant parmi les émetteurs de réponses le site ayant la plus grande position. D lui envoie donc un message *CONNECTION*. F positionne alors son *NEXT* à D et lui envoie l'identifiant de ses deux nouveaux prédécesseurs (F et I). Sur la figure 4.8(b) on peut observer le résultat final de ce recouvrement. On remarque alors deux choses : d'une part les sites défaillants ont bien été extraits de la chaîne des *NEXT* et d'autre part l'utilisation de la diffusion a permis ici de les exclure complètement de l'arbre des $LAST$. Tout se passe alors comme si les sites défaillants n'avaient jamais appartenu au système.

Si l'on considère maintenant l'état final obtenu par le mécanisme de recouvrement proposé par Naimi et Tréhel (figure 4.8(c)), on retrouve quasiment celui de la figure 4.7(c) : il ne manque que le site E. En fait, le nombre de défaillances présentes dans la chaîne des *NEXT* n'influe en rien sur cet algorithme : quelle que soit la configuration, le jeton se perd dans le premier site défaillant, ce qui conduit à une ré-initialisation globale du système.

La comparaison des deux situations finales aboutit aux mêmes conclusions que l'on avait faites à la section 4.5.3 pour le mécanisme *M1-SearchPred*, à savoir, une limitation de la perte du jeton ainsi qu'une plus grande équité pour notre extension. Notons tout de même qu'ici le nombre de défaillances n'est plus limité et que ces dernières peuvent toucher plus de k sites consécutifs dans la chaîne des *NEXT*. Notre algorithme assure donc à tout site muni d'une position d'accéder à la section critique en un temps borné sans condition sur le nombre de défaillances. Nous démontrerons cette assertion lors de la preuve à la section 4.6.

Il peut être intéressant de comparer le résultat des recouvrements du mécanisme *M1-SearchPred* (figure 4.7(b)) et celui du mécanisme *M2-SearchPos* (figure 4.8(b)). On remarque alors une différence notable : dans le premier, le site C défaillant est toujours présent dans l'arbre des $LAST$ alors que dans le second, l'ensemble des sites défaillants (C et E) ont été exclus non seulement de la file mais aussi de l'arbre. Cette différence s'explique par l'utilisation (ou non) de la diffusion de message et suggère une optimisation pour les systèmes répartis pourvus d'un service de diffusion. En effet, à la vue de ces résultats, on peut penser à généraliser dans ce type d'environnement l'utilisation du mécanisme *M2-SearchPos* pour l'ensemble des sites qui possèdent leur position.

4.5.5 Traitement de la perte d'une requête

Nous considérons maintenant le cas où un site S_i ne reçoit pas de message *COMMIT* en réponse à sa requête. À l'expiration du temporisateur *CommitTimer*, ce site ne connaît donc ni sa position, ni ses prédécesseurs dans la file des *NEXT*.

La difficulté pour re-connecter ce site, en absence d'information, est de prévenir la formation de cycles dans la chaîne des *NEXT*. Ainsi, il faudra non seulement éviter de re-connecter le site à l'autre extrémité de sa propre chaîne, mais aussi éviter des re-connections croisées lors de recouvrements concurrents. De tels cas peuvent survenir, lorsqu'un site S_i choisit de se re-connecter à l'extrémité d'une chaîne issue d'un site S_j

alors que ce même site S_j se re-connecte à l'extrémité de la chaîne issue de S_i.

Afin de prévenir ces deux écueils, le mécanisme présenté ici, noté par la suite *M3-SearchQueue*, va exploiter la présence de positions dans le système pour choisir le site de re-connexion. L'idée est de chercher le site qui possède *la plus grande position*. Ce choix présente deux avantages. D'une part, ce site ne peut être l'extrémité du morceau de chaîne issue du site S_i. En effet, S_i ne possédant pas de position, aucun de ses successeurs (a fortiori l'extrémité de sa chaîne) ne peut en avoir une. D'autre part, les prédécesseurs de ce site possèdent tous une position. Ils ne pourront donc pas lancer ce mécanisme de manière concurrente : seuls les sites n'ayant pas reçu d'acquittement peuvent lancer le mécanisme *M3-SearchQueue*. On évite ainsi les cas de re-connexions croisées. Le choix de la plus grande position comme point de re-connexion exclut donc toute formation de cycle dans la file d'attente.

La recherche de cette position passe par la diffusion d'un message *SEARCH_QUEUE* et le déclenchement d'un temporisateur *ReconnectionTimer* par le site S_i ayant détecté la défaillance (algorithme 4.6, lignes 52 et 53). Tous les sites ayant une position dans la file répondent alors au site S_i en lui envoyant un message *POSITION*. Comme pour le mécanisme *M2-SearchPos*, l'expiration du temporisateur lance la fonction *TimeoutReconnection()* qui est chargée de la re-connexion au site ayant la plus grande position. Notons qu'ici, la re-connexion ne passe pas forcément par l'envoi d'un message *CONNECTION*. En effet, il nous faut distinguer deux cas, suivant que le site trouvé a ou non enregistré un *NEXT* avant de recevoir le message *SEARCH_QUEUE* (algorithme 4.6, lignes 84 à 87). Dans le premier cas, ce *NEXT* devrait avoir la position maximum. N'ayant pas répondu, il est alors considéré comme défaillant (le cas d'un message *COMMIT* en transit sera étudié plus bas). Dans ce cas, l'utilisation d'un message *CONNECTION* permet au site S_i de prendre la place de l'ancien *NEXT* (algorithme 4.6, ligne 85). À l'inverse si le site trouvé ne possédait pas de *NEXT*, S_i se re-connecte en lui envoyant une simple requête (algorithme 4.6, ligne 87).

Comme pour le mécanisme *M2-SearchPos*, l'absence de réponse, et donc de site correct possédant une position, implique une perte du jeton. Pour simplifier nous supposerons que le site S_i est le seul site à avoir lancer le mécanisme *M3-SearchQueue*, nous reviendrons plus tard sur les recouvrements concurrents. Dans ces conditions, le site S_i peut s'attribuer une position égale à 0 et re-générer le jeton (algorithme 4.6, lignes 81 et 82).

Le mécanisme *M3-SearchQueue* permet donc de re-connecter un site qui n'avait pas obtenu de position. Cette re-connexion pose cependant un problème vis-à-vis de l'arbre des *LAST*. En effet, en absence d'information, le site se re-connecte systématiquement à la position la plus élevée ce qui ne permet pas de préserver les invariants liant la chaîne des *NEXT* à l'arbre des *LAST*. Contrairement aux mécanismes *M1-SearchPred* et *M2-SearchPos*, on est ici dans l'obligation de reconstruire un arbre compatible avec cette nouvelle chaîne.

Afin de ne pas augmenter la complexité de l'algorithme, cette reconstruction va se faire dynamiquement à la réception du message *SEARCH_QUEUE*. Il n'y a donc pas d'ajout de messages supplémentaires. Le principe de mise à jour de la variable *LAST* est le suivant :

- les sites attendant le jeton sans avoir encore de position, mettent leur *NEXT* et leur *LAST* à NIL et ré-émettent leur requête à l'émetteur du message *SEARCH_QUEUE*

ce qui incrémente leur compteur de requêtes *cmptReq* (algorithme 4.6, lignes 100 à 102).
- parmi les sites ayant un *LAST* (*i.e.*, qui ne sont pas racine), ceux qui ont reçu une position ou qui n'attendent pas la section critique, re-positionnent leur *LAST* pour désigner l'émetteur du message *SEARCH_ QUEUE* en considérant que celui-ci est le dernier à demander la SC (algorithme 4.6, ligne 105).

Recouvrements concurrents : Lors d'appels concurrents au mécanisme *M3-SearchQueue*, il y a un risque de re-génération de plusieurs jetons. Pour lever ce problème, on se ramène au cas simple d'un recouvrement "solitaire", en incluant un mécanisme d'élection. Là encore nous profitons de la diffusion déjà présente pour ne pas ajouter de surcoût supplémentaire. Cette élection est basée sur l'utilisation d'estampille.

Ainsi, chaque site maintient un couple *(cmptElec,id)*, initialisé à *(0,0)*, où : *cmptElec* est un compteur d'élection et *id* correspond à l'identifiant du meilleur candidat. Lorsqu'un site lance d'un recouvrement *M3-SearchQueue* (expiration du temporisateur *CommitTimer*), il incrémente son compteur *cmptElec* et enregistre son propre identifiant. Le couple passe donc à (*cmptElec* +1, identifiant du site). Les messages *SEARCH_ QUEUE* sont alors estampillés avec ce couple.

Lorsqu'un site reçoit un message *SEARCH_ QUEUE* portant une estampille plus grande que la sienne (au sens de la relation d'ordre total définie par Lamport [9]), il la met à jour en affectant la valeur reçue et considère l'émetteur comme nouveau meilleur candidat. De plus, s'il était lui même candidat (*i.e.*, sans position avec un temporisateur *ReconnectionTimer* en cours), il abandonne son recouvrement en réarmant un temporisateur *CommitTimer* (algorithme 4.6, ligne 103).

Pour minimiser le nombre de recouvrements concurrents, on borne la date des expiration d'alarmes *CommitTimer* par la date de la fin de la dernière élection (date de la réception du dernier *SEARCH_ QUEUE* + $2T_{msg}$). On évite ainsi de relancer une élection dans une autre. Notons que le recouvrement se fait globalement. Tous les sites bénéficient donc de l'élection courante et de la reconstruction de l'arbre qu'elle induit.

Requêtes en transit : L'ajout de ces estampilles va aussi permettre de traiter le problème des requêtes en transit pendant le recouvrement. En effet, puisque les messages *SEARCH_ QUEUE* sont utilisés pour reconstruire l'arbre, on peut voir leurs estampilles comme un numéro d'incarnation de l'arbre. L'idée est alors de ne pas traiter les messages provenant d'une ancienne version de l'arbre.

Ainsi, les requêtes sont estampillées par leur émetteur d'origine avec la valeur du couple *(cmptElec,id)*. Cette estampille ne sera pas modifiée lors des retransmissions dans l'arbre des *LAST*. Les sites ne traitent alors pas les messages portant une estampille inférieure à la leur [10]. Si l'estampille est supérieure, on simule l'arrivée d'un message *SEARCH_ QUEUE*. Notons, que le problème des acquittements et du jeton en transit est, lui, résolu par le test sur le compteur de requête *cmptReq*, déjà présenté à la section 4.5.2.

9. $(a_1, b_1) < (a_1, b_1) \Longleftrightarrow (a_1 < a_2) \vee (a_1 = a_2 \wedge b_1 < b_2)$
10. pour simplifier la lecture du pseudo-code, ces tests n'apparaissent pas dans l'algorithme 4.6

Exemple du mécanisme *M3-SearchQueue* **:** Comme précédemment, nous comparons le recouvrement de la perte d'une requête par chacune des deux approches. La figure 4.9(a) présente une forêt des $LAST$ composée de trois arbres. Ces arbres sont reliés par deux requêtes en transit provenant des sites F et I. Toutes deux ont pour destination le site C. La défaillance de ce dernier va donc isoler les arbres de racine E et H.

La figure 4.9(d) montre le résultat obtenu par notre algorithme après recouvrement. On suppose ici que les sites F et I ont tous deux détecté la perte de leur requête respective (expiration du temporisateur *TokenTimer*) et que le site I remporte l'élection initiée par les diffusions de messages *SEARCH_ QUEUE*. Le temporisateur *ReconnectionTimer* finit donc par expirer sur le site I. À cet instant, il aura reçu les messages *POSITION* des sites A et B. Comme A possède la plus grande position, le site I se re-connecte à lui, en lui envoyant une nouvelle requête. Par ailleurs, les autres sites (E, F, H et J) en attente d'acquittement (sans position) renouvellent leur requête au site I pour se re-connecter à la chaîne des $NEXT$ (figure 4.9(d)).

Parallèlement à cette re-construction de la chaîne $NEXT$, le mécanisme *M3-SearchQueue* génère une nouvelle forêt des $LAST$ compatible avec la nouvelle chaîne. Les sites D et G qui n'attendent pas la section critique prennent le site I comme nouveau $LAST$. Il en est de même pour le site B qui possède une position. Les autres sites forment des arbres d'un seul nœud (figure 4.9(d)).

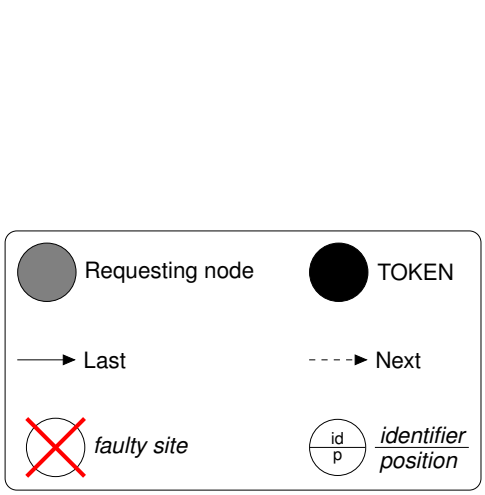
Une fois l'ensemble des requêtes traitées, le système sera alors composé d'un arbre des $LAST$ unique présenté sur la figure 4.9(e). Cette figure permet entre autres de voir comment les positions se sont propagées une fois les requêtes acquittées.

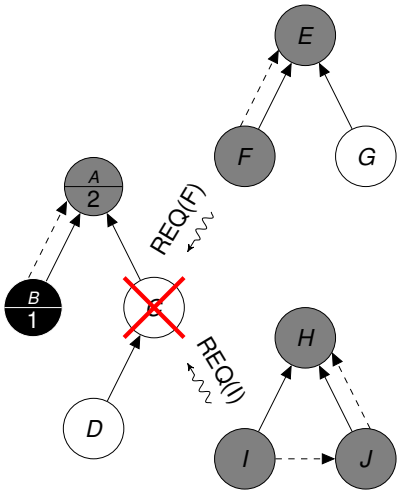
Considérons maintenant le recouvrement de la même défaillance par l'extension proposée par Naimi et Tréhel (figure 4.9(b)). Pour commencer, remarquons qu'ici la défaillance ne touche pas la chaîne des $NEXT$. Contrairement aux deux exemples précédents (figures 4.8(c) et 4.9(b)), le jeton ne va pas se perdre. Il n'y aura donc pas de ré-initialisation du système. Les sites F et I, qui ont détecté la perte de leur requête, finissent par la renouveler auprès du possesseur du jeton (B). Au passage, ils annulent leur $NEXT$, coupant ainsi les sites E et J de la chaîne. Ces derniers devront donc lancer un autre recouvrement individuel.

Comparons maintenant sur les figures 4.9(e) et 4.9(c) les résultats de ces deux recouvrements après le traitement des requêtes ré-émises. Notre approche présente alors le réel avantage de retrouver un système stable. À l'inverse, dans la solution proposée par Naimi-Tréhel, le résultat des **recouvrements individuels** n'est pas complètement stable. D'une part, la chaîne des $NEXT$ est toujours endommagée. Il faudra attendre d'autres recouvrements individuels, ce qui peut conduire à une inutilisation temporaire du jeton. De plus, le site défaillant C est toujours présent dans l'arbre, ce qui engendrera de nouvelles pertes de requêtes. Cette comparaison montre bien l'un des points forts de notre algorithme : lorsque l'utilisation de la diffusion devient nécessaire, on profite de celle-ci pour réaliser un **recouvrement global sans ré-initialisation**.

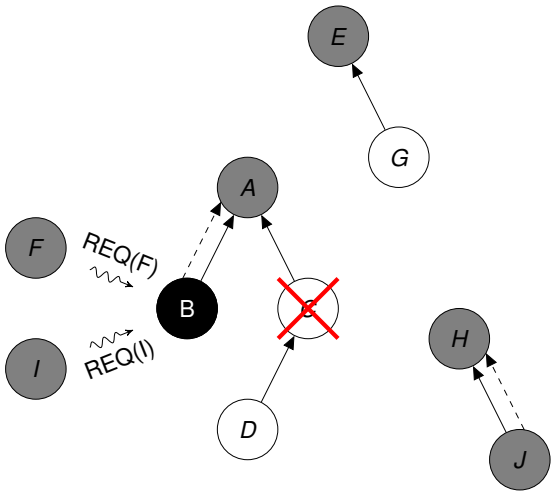
Pour terminer, nous profitons de cet exemple (figure 4.9(d)) pour montrer que la reconstruction de l'arbre des $LAST$ respecte bien ici les **invariants** (2 à 6) énoncés à la section 4.1.2 :

- invariant 2 : Pour les cinq requêtes en transit, on a bien une forêt des $LAST$ composée de six arbres.

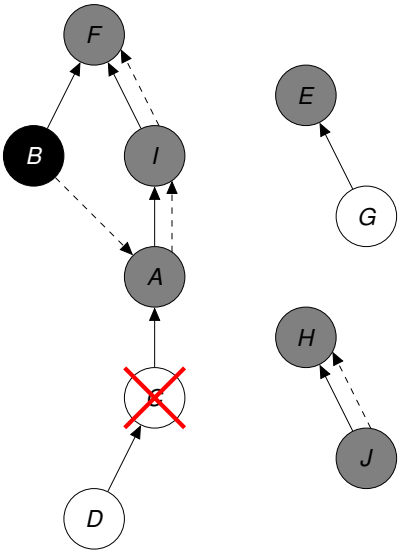




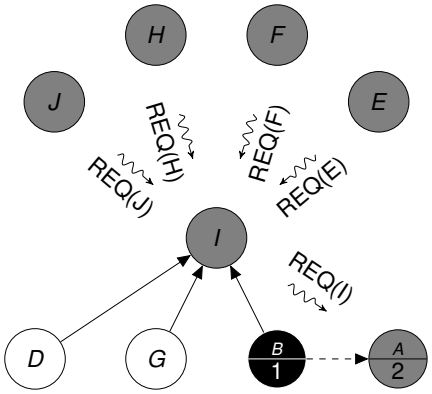
(a) Configuration initiale



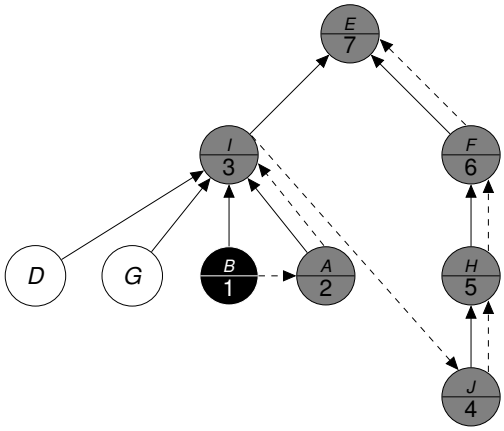
(b) Algorithme de Naimi-Tréhel : reconnexion de I et F



(c) Algorithme de Naimi-Tréhel : état final



(d) Notre algorithme : reconnexion de I et F



(e) Notre algorithme : état final

FIGURE 4.9 – Exemple de recouvrement de défaillances : *M3-SearchQueue*

- invariant 3 : Les morceaux de chaînes des *NEXT* ont bien pour extrémité les racines des arbres des *LAST* et vice-versa.
- invariant 4 : Les requêtes en transit forment bien avec les arbres des *LAST* un *arbre des requêtes*.
- invariant 6 : Le site B, qui possède le jeton, est bien l'origine de la chaîne des *NEXT* aboutissant à la racine de la racine de l'arbre des requêtes (A).

4.5.6 Ajout de pré-acquittements

La base de notre extension réside dans l'ajout d'un mécanisme d'acquittement des requêtes. Ces acquittements sont construits récursivement : chaque site de la chaîne des *NEXT* utilise les informations reçues dans le message *COMMIT* de son prédécesseur pour envoyer à son tour un acquittement à son *NEXT*. On ne peut donc exclure qu'un site reçoive une requête alors qu'il n'a pas encore reçu d'acquittement pour la sienne. Nous avions jusqu'alors choisi de résoudre ce problème en attendant la réception de l'acquittement manquant pour envoyer un message *COMMIT* complet. Cependant, une deuxième approche plus dynamique est envisageable. Celle-ci consiste à permettre l'émission d'acquittement partiel appelé **pré-acquittement**. Ce message sera ensuite complété par un acquittement classique dès que l'ensemble des informations nécessaires sera disponible.

Cette deuxième approche présente l'inconvénient d'ajouter un message supplémentaire dans le traitement des requêtes. Elle peut aussi conduire à complexifier la gestion du recouvrement des défaillances puisqu'un site pourra être amené à connaître un certain nombre de ses prédécesseurs sans pour autant avoir reçu de position (variable $pos = -1$).

Cependant, en consolidant au plus vite les morceaux de chaîne des *NEXT*, l'ajout de pré-acquittements permet d'améliorer la stabilité de notre algorithme. Les conséquences de la perte de la requête d'un site S_i sont alors limitées à ce site. En effet, si cette perte empêche ce site et donc ses successeurs d'obtenir une position, elle n'empêchera pas les successeurs de S_i d'être informés du bon enregistrement de leur requête. Ils pourront alors surveiller l'état de leurs prédécesseurs respectifs limitant ainsi l'usage du mécanisme *M3-SearchQueue* (détection de la perte d'une requête). Le coût de cette surveillance étant très faible, comparativement au mécanisme *M3-SearchQueue*, il devient possible d'utiliser des temporisateurs de détection plus agressifs (temps de détection plus court).

Considérons maintenant les modifications nécessaires à l'ajout des pré-acquittements dans notre extension. Seuls les mécanismes *M1-SearchPred* et *M3-SearchQueue*, sont ici concernés. En effet, le mécanisme *M2-SearchPos* est basé sur la recherche d'une position inférieure à la sienne. Elle nécessite donc la possession d'une position. En revanche, les deux autres mécanismes vont devoir être modifiés.

Modification du mécanisme *M1-SearchPred* : Il ne s'agit ici que d'une adaptation, le fonctionnement restant identique. Les deux modifications concernent la gestion de l'absence de position. Ainsi, en cas de défaillance du prédécesseur direct du site S_i, il n'est plus possible d'utiliser la position du premier prédécesseur correct trouvé ($Pred[x]$) pour détecter la perte du jeton. Nous utilisons alors la valeur du compteur de requêtes *cmptReq* (déjà utilisé pour la gestion des acquittements pendants à la section 4.5.2). Le numéro de la requête courante est alors ajouté au vecteur des prédécesseurs dans les acquittements

et pré-acquittements. Le site S_i pourra donc le joindre au message *CONNECTION* en lieu et place de la position ($Pred[x].pos$ dans la ligne 58 de l'algorithme 4.6). Comme pour la position un changement du compteur *cmptReq* caractérise alors une perte du jeton [11].

La deuxième modification concerne la gestion de l'échec de la recherche de prédécesseur direct. Si l'algorithme original lançait alors systématiquement le mécanisme *M2-SearchPos*, ce n'est ici pas toujours possible. Si le site ne possède pas de position, mais uniquement une liste des prédécesseurs reçue dans un pré-acquittement, il n'y a d'autre choix que de lancer le mécanisme *M3-SearchQueue*. Il faut donc ajouter un test sur l'existence de la position, dans la fonction *TimeoutToken()*, pour choisir le traitement approprié à une défaillance généralisée des prédécesseurs connus.

Modification du mécanisme *M3-SearchQueue* : Si le fonctionnement original ne conserve que la chaîne munie de positions, l'ajout des pré-acquittements va permettre de conserver les morceaux de chaîne des *NEXT* sans position. La recherche de la plus grande position, l'élection et la re-connexion du site élu restent identiques. Il faut cependant adapter la reconstruction de l'arbre. Ainsi, à la réception du message *SEARCH_ QUEUE*, seuls les candidats battus ré-émettent leur requête. Il faut alors tenir compte de ces morceaux de chaîne sans position lors de la reconstruction de l'arbre. La mise à jour de la variable *LAST* devient la suivante :

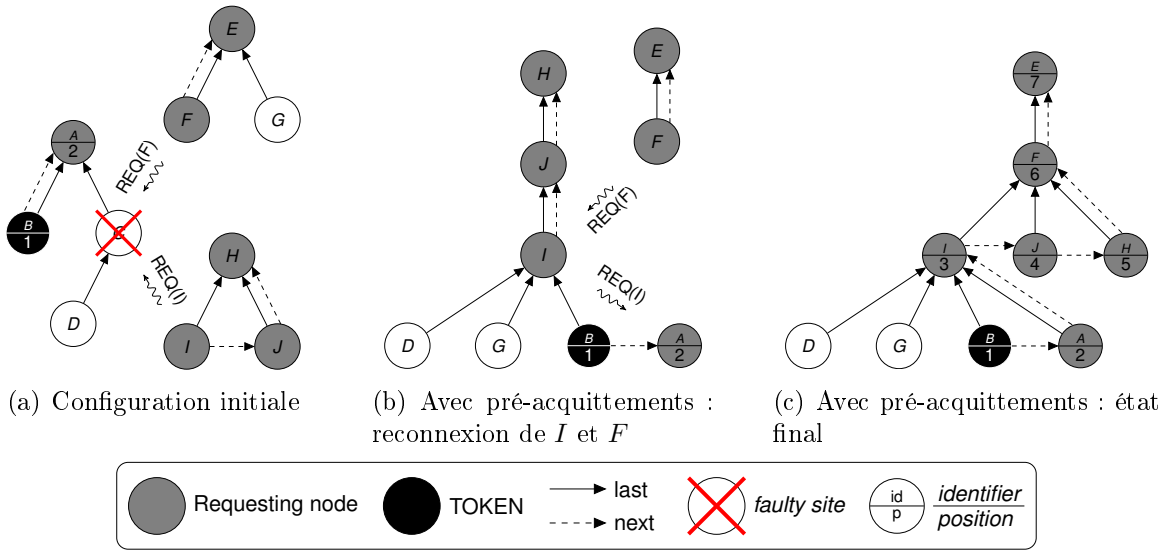
- Les sites ayant une position dans la file ou n'étant pas demandeurs de SC, repositionnent leur *LAST* s'ils en avaient un sur l'émetteur du message *SEARCH_ QUEUE* en considérant que celui-ci est le dernier à demander la SC.
- Les sites attendant le jeton *sans avoir encore de position* mettent leur *LAST* à la même valeur que leur *NEXT*. Ils sont sûrs que le *NEXT* a demandé le jeton après eux. Ceci qui permet d'éviter des cycles dans l'arbre des *LAST*.

Exemple avec pré-acquittements : Pour mieux comprendre cette reconstruction, nous allons reprendre l'exemple précédent en considérant l'ajout des pré-acquittements. Ainsi, la figure 4.9(d) montre le résultat du recouvrement de la défaillance du site C ayant provoqué la perte des requêtes provenant des sites F et I.

La figure 4.10(a) présente l'état initial du système après la défaillance du site C. Dans cette figure, on observe la formation d'un morceau de chaîne des *NEXT* dans l'arbre de racine H : la requête du site J (resp. H) est parvenue jusqu'au site I (resp. J) alors que la requête du site I est toujours en transit. N'ayant pas reçu de position ces sites ne peuvent acquitter les requêtes.

C'est dans ce cas qu'interviennent les "pré-acquittements". À la réception de sa requête, le site I avertit le site J qu'il est son prédécesseur dans la chaîne des *NEXT*. La construction de la liste des prédécesseurs étant incrémentale, chaque "pré-acquittement" peut bénéficier des "pré-acquittements" précédents. Ainsi, si la requête de H parvient à J alors qu'il a déjà reçu le pré-acquittement de I, le site J pourra "pré-acquitter" H en l'informant de ces 2 prédécesseurs : J et I. Le morceau de chaîne, formé par les sites I, J et

11. Notons, que le compteur *cmptReq* n'est plus ici modifié par le mécanisme *M3-SearchQueue*. Seul un accès à la section critique permet d'incrémenter ce compteur. De plus pour simplifier le test sur la perte du jeton, on peut incrémenter le compteur à la libération du jeton.



(a) Configuration initiale

(b) Avec pré-acquittements : reconnexion de I et F

(c) Avec pré-acquittements : état final

FIGURE 4.10 – Exemple de recouvrement de défaillances : *avec pré-acquittements*

K, est alors "consolidé" : les sites J et H ne surveillent plus l'arrivée de leur acquittement (*M3-SearchQueue*) mais uniquement l'état de leur prédécesseur direct (*M1-SearchPred*). Il en est de même pour l'autre morceau de chaîne sans position (F et E), où E surveille l'état de F.

Comme précédemment, on suppose maintenant que les sites F et I ont tous deux détecté la perte de leur requête respective (expiration du temporisateur *TokenTimer*) et que le site I remporte l'élection. Sur la figure 4.10(b) montrant le résultat de ce recouvrement, on peut voir que le site envoie toujours une requête à la plus grande position A. De même le site F (qui a perdu l'élection) continue d'envoyer une au site I. Par contre ici, les sites J, H et E (qui ont reçu un pré-acquittement) ne renouvellent pas leur requête et conservent leur *NEXT*.

Sur cette figure 4.10(b), on retrouve aussi une partie de la reconstruction de l'arbre des *LAST* vue précédemment (figure 4.9(d)). Les sites D et G, qui n'attendent pas la section critique, prennent le site I comme nouveau *LAST*. Le site B, qui possède une position et n'est pas racine, continue de prendre le site I comme nouveau *LAST*. En revanche, les sites attendant la section critique sans avoir reçu de position (I, J, H, F et E) font pointer leur *LAST* sur le *NEXT*. Ces morceaux de chaîne sans position forment alors des arbres "dégénérés". Une fois les deux requêtes traitées, le système sera alors composé d'un arbre des *LAST* unique présenté sur la figure 4.9(e).

Si l'on compare les résultats obtenus avec (figure 4.10(b)) et sans pré-acquittement (figure 4.9(d)), on remarque deux choses. D'une part les pré-acquittements ont permis de conserver l'ensemble des morceaux de chaîne des *NEXT*, ce qui permet dans une certaine mesure d'améliorer l'équité du recouvrement. D'autre part, les ré-émissions de requête se limitent aux extrémités des morceaux de chaîne sans position. Si le recouvrement reste global, l'utilisation des pré-aquittements permet d'en réduire le coût.

Pour terminer, nous profitons de cet exemple pour montrer que cette nouvelle recons-

truction de l'arbre des *LAST* respecte bien les **invariants** énoncés à la section 4.1.2 :

- invariant 2 : Pour les deux requêtes en transit, on a bien une forêt des *LAST* composée de trois arbres.
- invariant 3 : Les trois morceaux de chaînes des *NEXT* ont pour extrémité les racines des arbres des *LAST*, et inversement.
- invariant 4 : Les requêtes en transit forment bien avec les arbres des *LAST* un *arbre des requêtes*.
- invariant 6 : Le site B, qui possède le jeton, est bien l'origine de la chaîne des *NEXT* aboutissant à la racine de la racine de l'arbre des requêtes (A).

4.5.7 Propriétés de l'algorithme

Après avoir présenté notre extension ajoutant la tolérance aux défaillances à l'algorithme de Naimi-Tréhel, nous allons étudier ses propriétés. Notre algorithme présente les propriétés suivantes :

Conservation de la complexité de l'algorithme original : Si l'ajout d'acquittements augmente le nombre de messages nécessaires à chaque accès à la section critique, ce surcoût reste indépendant du nombre de sites présents dans le système. Notre extension conserve donc une complexité en $\mathcal{O}(log(n))$. Néanmoins en absence de défaillance, notre approche se montrera toujours un peu plus coûteuse en nombre de messages envoyés, que l'extension de Naimi-Tréhel. Notons que ce cette remarque ne s'applique qu'au système muni d'un service de diffusion. Dans le cas contraire la complexité de notre algorithme peut se montrer la plus faible (voir section 5.2.3.2). De plus, il faut relativiser le surcoût des acquittements par le coût de la surveillance et les gains qu'ils apportent dans ce domaine. Ce point sera vérifié dans les mesures de performance de la section 5.2.4.1.

Insensibilité aux défaillances n'ayant pas de conséquence : Tout comme l'extension proposée par Naimi et Tréhel, notre algorithme ignore les défaillances qui n'affectent pas le fonctionnement de l'algorithme initial (voir section 4.2).

Dynamicité pendant le recouvrement : Comme nous l'avons vu à la section 4.2, un petit nombre de pannes paralyse rarement l'ensemble de l'algorithme : hors cas exceptionnels (panne de la racine à l'initialisation), il existe toujours une grande partie de la *forêt des LAST* pouvant acheminer les requêtes vers des sites corrects. Notre algorithme permet d'exploiter ces structures valides pendant le recouvrement, ce qui permet de conserver l'aspect dynamique de l'algorithme initial. Ceci marque une réelle différence, sur la solution de Naimi-Tréhel. En effet, dans cette dernière la plupart des requêtes émises pendant le recouvrement seront annulées et donc vouées à être ré-émises (voir section 4.4.4).

Limitation des pertes de jeton : Notre algorithme se démarque aussi, en permettant de prévenir la perte du jeton, ce qui permet de masquer les délais de recouvrement en continuant à accéder à la section critique pendant le recouvrement. Ainsi, alors que l'extension de Naimi-Tréhel traite la défaillance d'un site de la chaîne des *NEXT* par une

ré-initialisation survenant uniquement après la *perte du jeton*, notre extension cherche à exclure le site défaillant avant que le jeton ne lui soit envoyé.

Si la taille de la chaîne est restreinte, alors la probabilité de perdre le jeton est faible pour les deux extensions. En effet, puisque l'on minimise le nombre de sites présents dans la chaîne, la probabilité qu'elle soit touchée par une défaillance est réduite.

À l'inverse, lorsque la taille de la file d'attente augmente, les conséquences sont différentes pour les deux approches. Dans l'extension de Naimi-Tréhel, l'accroissement du nombre de défaillances touchant la chaîne des *NEXT* entraîne directement une augmentation du nombre de pertes du jeton. Notre algorithme, lui, va permettre de limiter ces pertes. En effet, plus la chaîne sera grande et plus le mécanisme de re-connexion aura de temps pour réparer la chaîne, évitant ainsi la perte du jeton. Cet effet d'**isolation des défaillances** sera mis en évidence lors de l'étude de performance (section 5.2.3.3).

Limitation de l'usage de la diffusion : Si l'un des intérêts majeurs de l'algorithme de Mohamed Naimi et Michel Tréhel est son indépendance vis-à-vis de l'existence d'un service de diffusion, ce n'est pas le cas de leur extension pour la tolérance aux défaillances. Au contraire, notre approche tente de limiter au maximum l'emploi de la diffusion, qu'il s'agisse du recouvrement ou de la détection des défaillances. Comme nous le vérifierons dans l'étude de performances (section 5.2), son déploiement dans un système dépourvu d'un tel service est dès lors possible. Il permet d'ailleurs l'utilisation de temporisateurs de détection relativement agressifs, ce qui permet de diminuer sensiblement les temps de recouvrement.

Conservation de l'équité de l'algorithme original : Un autre point fort de notre algorithme réside dans la préservation de l'équité. Lors de l'étude de l'extension de Naimi-Tréhel, nous avions vu comment les ré-initialisations et les multiples ré-émissions qu'elle engendre remettent en cause l'ordre des requêtes. À l'inverse, en cherchant à maintenir dès que possible l'ordre des morceaux de la chaîne des *NEXT* encore intacts, notre algorithme préserve au mieux l'équité de l'algorithme initial. Ceci permet entre autres de garantir à tout site ayant reçu sa position d'accéder en un temps fini à la section critique quel que soit le nombre et la fréquence des défaillances.

4.6 Preuve

Cette section présente la preuve de notre algorithme. Nous avons choisi ici de ne considérer que la partie principale de notre algorithme, c'est à dire sans l'ajout du mécanisme de *pré-acquittement*. Cette démonstration est composée de deux parties. Dans un premier temps nous démontrerons que notre algorithme vérifie bien la propriété de sûreté de l'exclusion mutuelle. Puis, nous ferons de même pour la propriété de vivacité de type *équité faible* (voir section 2.2.3).

4.6.1 Sûreté

Notre algorithme est un algorithme à jeton : un site ne peut entrer en section critique (algorithme 4.6, lignes 16) qu'à l'unique condition de détenir le jeton. Prouver l'unicité d'accès à la section critique, revient donc à prouver l'unicité du jeton. Cette section s'attachera donc à prouver qu'à tout moment, il n'y ait au plus qu'un jeton dans le système.

Pour commencer, rappelons que chaque site maintien un compteur de requêtes *cmptReq*. Même s'il n'est pas présent dans le pseudo-code de l'algorithme 4.6, ce compteur *cmptReq* est transporté dans les messages *REQUEST*, *COMMIT* et *TOKEN*. Associé à l'identifiant du demandeur, il permet d'identifier de manière unique chaque requête, ainsi que les acquittements associés.

Pour simplifier l'énoncé de cette preuve, nous introduisons plusieurs définitions :

prédécesseurs directs : un site S_j est un prédécesseur direct du site S_i si S_i est le *NEXT* de S_j et si le *cmptReq* enregistré par S_j correspond au *cmptReq* actuel de S_i.

message *COMMIT* significatif : un message *COMMIT* adressé au site S_i est dit significatif si S_i n'a pas encore reçu de position et que ce message contient la valeur actuelle du compteur *cmptReq* du site S_i.

candidat : un site est dit candidat s'il n'a pas de position et qu'il a armé un temporisateur *ReconnectionTimer*.

T_{msg} **:** représente la borne sur le temps de propagation des messages offerte par les hypothèses de synchronisme.

Si cette partie de la preuve repose sur l'unicité du jeton, elle est aussi liée à la structure de la chaîne, aux positions et aux prédécesseurs connus. En effet, dans nos mécanismes de recouvrement, les maintiens de ces structures sont étroitement liés aux possibles régénérations du jeton. Nous introduisons donc pour chacune de ces structures une propriété de validité :

chaîne valide : la chaîne des *NEXT* est valide si tout site *correct* possède au plus un prédécesseur direct correct.

positions valides : l'ensemble des positions est valide, si quel que soit le site S_i possédant une position celle-ci est différente de celles des autres sites corrects, plus grande que celles de ces prédécesseurs directs et si tous les sites corrects possédant une position supérieure sont des successeurs de S_i dans la chaîne des *NEXT*.

prédécesseurs connus valides : l'ensemble des prédécesseurs connus est correct si quel que soit le site correct S_i ayant une position il n'y a pas d'autre site correct qui possède une position plus petite que celle de S_i et plus grande que la plus petite des positions des prédécesseurs connus par S_i. De plus les prédécesseurs connus corrects et ayant une position inférieure à celle de S_i (pas d'accès intermédiaire) sont ordonnés suivant leur position.

jeton valide : un site correct possédant le jeton, possède aussi la plus petite des positions des sites corrects.

De ces définitions on peut déduire plusieurs corollaires (qui se démontrent trivialement par l'absurde) :

Corollaire 1. *Lorsque l'ensemble des positions est valide, la position d'un site correct, qui n'a pas de NEXT ou dont le NEXT n'a pas de position, est la plus grande des positions des sites corrects.*

Corollaire 2. *Lorsque l'ensemble des postions est valide, si un site correct S_i a une position et a pour NEXT sur un site correct S_j qui a une position, alors il n'existe pas de site correct ayant une position plus grande que celle de S_i et plus petite que S_j.*

Pour clarifier cette démonstration, nous commençons par lister les actions ou émissions de messages qui entraînent une modification de ces variables :

- La notion *prédécesseur direct* est liée à la gestion de la variable *NEXT*. L'enregistrement d'un *NEXT* (algorithme 4.6, lignes 27 et 91) correspond à la réception d'un message *REQUEST* (par un site n'ayant pas encore de *NEXT*) ou d'un message *CONNECTION*. Notons qu'un site recevant une requête ne la traite que si son estampille est supérieure ou égale à la sienne.
- Les positions sont calculées pour chaque requête à la première réception d'un message *COMMIT* (algorithme 4.6, ligne 42) portant le bon numéro de requête (*cmptReq*). Ces messages peuvent faire suite à la réception de message *REQUEST*, *CONNECTION*.
- Les prédécesseurs connus (*Pred*[ ]) sont eux mis à jour à toutes les réceptions d'un message *COMMIT* (algorithme 4.6, ligne 40) portant le bon numéro de requête (*cmptReq*).
- Le jeton peut être acquis à la réception d'un message *TOKEN* portant le bon numéro de requête (*cmptReq*) ; soit être re-généré par les mécanismes *M2-SearchPos* et *M3-SearchQueue* à l'expiration d'un temporisateur *ReconnectionTimer* (algorithme 4.6, ligne 82).

Plus précisément, nous pouvons aussi tirer de l'algorithme un lemme sur la persistance de la valeur des positions :

Lemme 1. *Un site qui possède une position la conserve tant qu'il ne libère pas le jeton.*

Démonstration. Dans le pseudo-code de l'algorithme 4.6, en dehors de la fonction d'initialisation, il n'existe que quatre lignes qui modifient la variable *position* :

- Les deux premières, respectivement lignes 42 et 81, sont précédées d'un test, respectivement lignes 41 et 79, vérifiant que le site n'a pas de valeur *position* enregistrée. Elles ne peuvent donc pas modifier la position d'un site qui en possède une.
- Les deux suivantes, respectivement lignes 23 et 32, sont précédées par l'envoi d'un message *TOKEN*, respectivement lignes 21 et 31.

La perte d'une position équivaut donc à la libération du jeton. □

Lemme 2. *À tout moment le système possède une chaîne valide, des positions valides et des prédécesseurs connus valides.*

Démonstration. Pour prouver le lemme 2 nous allons procéder par récurrence en discrétisant le temps suivant les demandes d'entrée et de sortie de section critique et suivant les appels aux trois mécanismes de recouvrement. On suppose alors qu'à l'instant t, le système possède une *chaîne valide*, des *positions valides*, des *prédécesseurs connus valides*

et un *jeton valide*. On suppose aussi qu'en absence de toute nouvelle expiration de temporisateur et de nouvelle demande d'entrée et de sortie de section critique, les traitements des messages en transit et des messages engendrés par ces traitements (retransmissions des requêtes) ne remettent jamais en cause les quatre propriétés de *validité*.

Le passage de t à $t+1$ correspond au lancement de l'un des cinq mécanismes suivants : *RequestCS()*, *ReleaseCS()*, *M1-SearchPred*, *M2-SearchPos* ou *M3-SearchQueue*. On considère alors séparément leur déclenchement par le site S_i :

RequestCS() **:** L'appel à cette fonction engendre l'émission d'un message *REQUEST*. Or, en incrémentant le compteur *cmptReq* de S_i, il garantit qu'il n'y a plus de message *COMMITsignificatif* à destination de S_i et que le site S_i n'a plus de prédécesseur direct correct. De plus, cette requête ne sera jamais dupliquée : elle sera retransmise (algorithme 4.6, ligne 34) ou disparaitra avec l'enregistrement de S_i comme *NEXT* (algorithme 4.6, ligne 27). Le traitement de ce message ne peut donc conduire S_i à avoir plusieurs prédécesseurs directs corrects. Enfin les hypothèses de récurrences et le corollaire 1 assurent que le nouveau prédécesseur direct possède bien la plus grande des positions lorsqu'il émet le message d'acquittement correspondant à cette requête. Les différents traitements de ce message *REQUEST* ne remettent donc pas en cause la validité des structures.

ReleaseCS() **:** L'appel à cette fonction peut conduire le site S_i à relâcher le jeton (algorithme 4.6, lignes 21) et donc à perdre sa position (algorithme 4.6, lignes 23). Le jeton est alors envoyé à son *NEXT* (S_j) et on distingue trois cas :

1. Le site S_j n'a pas de position et n'a pas ré-émis de requête : Ce cas correspond à un message *COMMIT* encore en transit. Or par hypothèse de récurrence, le jeton est *valide*. S_i possédait la plus petite des positions des sites corrects. De plus, suivant le corollaire 1, S_i possédait aussi la plus grande des positions des sites corrects. Il n'y a donc plus de site correct avec une position. D'autre part, l'hypothèse de récurrence sur les messages en transit garantit qu'il ne peut y avoir d'autre message *COMMITsignificatif* en transit que celui provenant de S_i à destination de S_j. En effet le traitement de tels messages remettrait en cause la validité de la chaîne des *NEXT*. La réception du message en *TOKEN* par le site S_j ne remet donc pas en cause la validité des structures quel que soit son ordre de traitement.

2. Le site S_j n'a pas de position et a réémis une autre requête : Puisque S_j à réémis sa requête (suite à la réception d'un message *SEARCH_QUEUE*,) son compteur *cmptReq* ne correspond plus à la valeur contenue dans le message *TOKEN* envoyé par S_i. Ce message sera ignoré et ne peut donc remettre en cause la validité des structures.

3. Le site S_j a une position et n'a pas réémis de requêtes : Dans ce cas par hypothèse de récurrence et suivant le corollaire 2, on peut dire que S_i possédait la plus petite position des sites corrects et qu'il n'existait pas de site correct avec une postion plus grande que celle de S_i et plus petite que S_j. S_j possède donc maintenant la plus petite des positions. Il peut donc recevoir le message *TOKEN* sans remettre en question la validité des structures.

M1-SearchPred : Ce mécanisme résulte de l'expiration du temporisateur *timerToken*. Il implique donc la réception d'un acquittement et n'agit réellement que lorsque le $Pred[1]$ est défaillant et qu'il existe encore un prédécesseur connu correct. Dans le cas contraire il se contente, respectivement, de réarmer le temporisateur ou lancer le mécanisme *M2-SearchPos*. On considère donc le plus petit rang x tel que $Pred[x]$ soit correct. Le site S_i émet donc un message *CONNECTION* au site $Pred[x]$. On distingue alors deux cas :

1. La position actuelle de $Pred[x]$ correspond à celle connue par S_i ($Pred[x].pos$) : Dans ce cas, par hypothèse de récurrence, il n'y a pas de site correct ayant une position comprise entre $Pred[x].pos$ et $S_i.pos$. Or, puisque les positions sont *valides*, le prédécesseur direct de S_i est défaillant ou alors est le site $Pred[x]$ (cas d'un acquittement de $Pred[x]$ encore en transit), et réciproquement pour le *NEXT* de $Pred[x]$ qui est soit défaillant soit le site S_i. Le traitement du message *CONNECTION* et la mise à jour du *NEXT* de $Pred[x]$ (algorithme 4.6, ligne 91) ne remettent donc pas en cause la validité de la chaîne et des positions. De même, puisque les prédécesseurs connus de $Pred[x]$ sont valides, le message d'acquittement *COMMIT* faisant réponse au message *CONNECTION* ne peut remettre en cause la validité des prédécesseurs connus de S_i quel que soit l'ordre de traitement de ce message [12].
2. $Pred[x]$ n'a plus de position ou a une position différente de celle connue par S_i ($Pred[x].pos$) : Le changement ou la perte de position impliquant la libération du jeton, le site $Pred[x]$ a donc possédé le jeton. Par hypothèse de récurrence $Pred[x].pos$ était donc la plus petite position des sites corrects. De plus les hypothèses sur les prédécesseurs connus assurent qu'il n'y a pas de site correct avec une position comprise entre $Pred[x].pos$ et celle de S_i. S_i possède donc la plus petite des positions des sites corrects. Aucun autre site correct ne peut donc posséder le jeton, ni être le destinataire d'un jeton portant un compteur valide. L'envoi d'un message *TOKEN* par le site $Pred[x]$ au site S_i ne remet donc pas en cause la validité du jeton quel que soit son ordre de traitement.

M2-SearchPos : Le lancement de ce mécanisme fait suite à l'échec du mécanisme *M1-SearchPred* (les k prédécesseurs connus sont défaillants). Le site S_i diffuse un message *SEARCH_POSITION* et enregistre les réponses *POSITION* jusqu'à écoulement du temporisateur *ReconnectionTimer* de $2T_{msg}$. On distingue alors deux cas :

1. S_i a reçu au moins une réponse : Les hypothèses de synchronisme lui assurent que la plus grande des positions reçues correspond au site correct S_j ayant la plus grande position inférieure à la sienne parmi toutes celles des autres sites corrects. Puisqu'il n'y a pas d'autre site correct ayant une position comprise entre $S_j.pos$ et $S_i.pos$, les hypothèses de récurrence assurent que le prédécesseur direct du site S_i est le site S_j ou est défaillant. De même, on peut dire que le *NEXT* du site S_j est le site S_i ou est défaillant. Comme pour le mécanisme *M1-SearchPred*, S_i l'envoi d'un message *CONNECTION* et de son acquittement ne remettent pas en cause la validité des quatre structures.

12. La concurrence des messages *CONNECTION* sera étudiée dans la partie (b) du mécanisme *M3-SearchQueue*

2. S_i n'a pas reçu de réponse : Les hypothèses de synchronisme lui assurent qu'il n'existe pas de site correct avec une position plus petite. Or, puisque le jeton est valide, aucun autre site correct ne peut donc posséder le jeton, ni être le destinataire d'un jeton portant un compteur valide. La génération d'un jeton (algorithme 4.6, ligne 82) ne remet donc pas en cause la validité du jeton.

M3-SearchQueue : Au lancement de ce mécanisme, le S_i incrémente son horloge logique, diffuse un message *SEARCH_ QUEUE* estampillé par la nouvelle valeur de l'horloge et arme un temporisateur *ReconnectionTimer* de $2T_{msg}$: S_i devient donc candidat. Le message *SEARCH_ QUEUE* n'est traité que par les sites ayant une estampille inférieure. Tous les sites, attendant la section critique sans avoir de position, ré-émettent alors leur requête (algorithme 4.6, ligne 102). Ces ré-émissions s'accompagnent d'une incrémentation du compteur *cmptReq* garantissant ainsi l'absence de prédécesseurs directs corrects. Elles ne remettent donc pas en cause la validité des structures. Pour le site S_i, on distingue alors deux cas :

1. S_i a reçut un message *SEARCH_ QUEUE* avec une estampille plus grande avant la fin du temporisateur *ReconnectionTimer* : Dans ce cas S_i ré-émet sa requête comme les autres sites sans position (algorithme 4.6, ligne 102). Cette ré-émission ne remet donc pas cause la validité des structures. De plus, elle s'accompagne d'une annulation du temporisateur *ReconnectionTimer*. Le site S_i n'est donc plus candidat.
2. Expiration du temporisateur *ReconnectionTimer* sur S_i : Les hypothèses de synchronisme lui garantissent qu'à l'expiration du temporisateur, l'ensemble des sites corrects auront reçu son message *SEARCH_ QUEUE*. Ils ont donc tous une horloge logique au moins aussi grande que celle de S_i. De plus, tous les sites sans position ont annulé leur *NEXT* (algorithme 4.6, ligne 100). Si l'on considère le message *REQUEST* préalablement émis par S_i, on distingue quatre cas :
   (a) La requête de S_i est arrivée dans un site défaillant : dans ce cas le site ne peut avoir de prédécesseur direct correct.
   (b) La requête de S_i est arrivée dans un site S_j sans *NEXT* et sans position avant qu'il ne reçoive le message *SEARCH_ QUEUE* de S_i : dans ce cas S_j a pris S_i comme *NEXT*, mais l'a effacé à la réception du message *SEARCH_ QUEUE* (algorithme 4.6, ligne 100). S_i ne peut avoir de prédécesseur direct correct.
   (c) La requête de S_i est arrivée dans un site S_j après qu'il ne reçoive le message *SEARCH_ QUEUE* de S_i : dans ce cas S_j ignore la requête de S_i qui porte une estampille trop ancienne. Cette requête est donc perdue et S_i ne peut avoir de prédécesseur direct correct.
   (d) La requête de S_i est arrivée dans un site S_j sans *NEXT* mais avec position, avant qu'il ne reçoive le message *SEARCH_ QUEUE* de S_i : dans ce cas S_j a pris S_i comme *NEXT*. Or, S_i n'ayant pas de position, les hypothèses de récurence et le corollaire 1 implique S_j possède la plus grande position.

On peut donc conclure qu'à l'expiration du temporisteur *ReconnectionTimer* le site S_i ne possède au plus qu'un prédécesseur direct correct et que ce dernier ne peut être que le site qui possède la plus grande position. On considère maintenant le traitement de l'expiration du temporisateur sur S_i et on distingue trois cas :

(a) S_i a trouvé un site S_j avec une position et sans *NEXT* : Le dimensionnement du temporisateur assure aussi au site S_i d'avoir bien reçu toutes les réponses *POSITION*. Le site S_j possède donc la plus grande des positions des sites corrects et puisqu'il n'a pas de *NEXT*, le site S_i ne peut avoir de prédécesseur direct correct (voir ci-dessus). La ré-émission d'une requête (algorithme 4.6, ligne 87) ne remet donc pas en cause la validité des structures quel que soit son ordre de traitement.

(b) S_i a trouvé un site S_j avec une position et un *NEXT* : Comme précédemment le dimensionnement du temporisateur assure d'avoir trouvé le site avec la plus grande position. Mais ici puisqu'il a un *NEXT* (S_k), le site S_i émet un message *CONNECTION* en destination de S_j (algorithme 4.6, ligne 85). Il nous faut alors distinguer trois cas :

i. S_k est défaillant : Dans ce cas le traitement du message *CONNECTION* par S_j (mise-à-jour de son *NEXT* vers S_i, algorithme 4.6, ligne 91) ne remet pas en cause la validité des structures tant que le *NEXT* de S_j reste S_k.

ii. S_k n'est pas défaillant et n'est pas S_i : Dans ce cas, S_k n'avait pas encore reçu l'acquitement de S_j lorsqu'il a reçu le message *SEARCH_ QUEUE*. En effet, dans le cas contraire il aurait communiqué à S_i sa position et celle-ci aurait été, par hypothèse, plus grande que celle de S_j. S_k a donc ré-émis une requête et son compteur *cmptReq* ne correspond plus au message *COMMIT* pendant. Le traitement du message *CONNECTION* par S_j ne remet donc pas en cause la validité des structures tant que le *NEXT* de S_j reste S_k.

iii. S_k et S_i ne font qu'un : Dans ce cas, puisque S_i est déja le *NEXT* de S_j, le traitement du message *CONNECTION* par S_j ne change rien tant qu'il n'a pas modifié son *NEXT*.

Dans tous les cas, le traitement du message *CONNECTION* ne remet pas en cause la validité des structures si le *NEXT* n'est pas modifié. Or, il n'existe que deux façons pour changer de *NEXT* :

i. À l'envoi du jeton au *NEXT* : Dans ce cas le changement de *NEXT* s'accompagne d'une perte de la position. Celle-ci sera alors détectée à la réception du message *CONNECTION* par S_j (algorithme 4.6, ligne 90). Le traitement de ce message n'engendrera donc pas de mise à jour du *NEXT*, mais uniquement un envoi de jeton (algorithme 4.6, ligne 94). Le traitement de ce dernier ne pause alors pas de problème pour la validité du jeton, comme montré dans la partie (2) de la preuve du *M1-SearchPred*.

ii. À la réception d'un message *CONNECTION* : Les hypothèses de synchronisme et le dimensionnement du temporisateur *ReconnectionTimer* à $2T_{msg}$ garantissent, au site S_i, qu'au moment de l'expiration de ce temporisateur, tous les autres sites candidats le sont depuis au plus T_{msg}. Ils ne pourront donc pas émettre un message *CONNECTION* concurrent avant T_{msg} [13], ce qui laisse le temps au message *CONNECTION* de S_i d'arriver au site S_j.

Le traitement du message *CONNECTION* ne remet pas en cause la validité des structures quel que soit son ordre de traitement.

(c) S_i n'a pas trouvé de site avec une position : Dans ce cas les hypothèses temporelles lui garantissent qu'il n'y a plus de site correct possédant une position. Or, puisque le jeton est valide, aucun autre site correct ne peut donc posséder le jeton, ni être le destinataire d'un jeton portant un compteur valide. Enfin puisque S_i n'a pas reçu de message *SEARCH_ QUEUE* avec une estampille plus grande que la sienne depuis au moins $2T_{msg}$, il a la garantie de n'avoir pas encore reçu les messages *SEARCH_ QUEUE* des autres sites corrects candidats au moment de l'expiration du temporisateur *ReconnectionTimer*. La génération d'un jeton (algorithme 4.6, ligne 82) ne remet donc pas en cause la validité du jeton. Et puisque celle-ci s'accompagne d'une auto-attribution de la position 0 (algorithme 4.6, ligne 81), les traitements par S_i des messages *SEARCH_ QUEUE* pendant n'engendreront pas de nouveau jeton (S_i répondra par un message *POSITION*).

On peut donc dire qu'à $t+1$ le système possède une *chaîne valide*, des *positions valides*, des *prédécesseurs connus valides* et un *jeton valide*. On suppose aussi qu'en absence de toute nouvelle expiration de temporisateur et de nouvelle demande d'entrée et de sortie de section critique, les traitements des messages en transit et des messages engendrés par ces traitements (retransmissions des requêtes) ne remettent jamais en cause les quatres propriétés de *validité*.

Pour terminer cette preuve par récurrence vérifions que l'état initial satisfait aux hypothèses de récurrence. Dans cet état seul la racine possède une position et ce site possède aussi le seul jeton du système. De plus, aucun autre site n'attendant la section critique la chaîne des *NEXT* est réduite à la racine. Enfin aucun message n'a été envoyé.

Par récurrence on peut donc dire que notre algorithme vérifie le lemme 2. □

Théorème 1. *À tout moment, il n'y a au plus qu'un site correct en section critique.*

Démonstration. Le lemme 2 implique qu'à tout moment, il n'y a au plus qu'un site *correct* qui possède un jeton. Or seul les sites possédant un jeton peuvent accéder à la section critique. L'algorithme vérifie donc la propriété de sûreté. □

4.6.2 Vivacité

Nous allons maintenant prouver la propriété d'équité faible. Cette propriété de vivacité a été introduite dans la section 2.2.2 et assure à tout site qui le demande d'accéder à la

13. Un temporisateur, d'une durée de $2T_{msg}$, armé il y a T_{msg}, n'expirera que dans T_{msg}.

section critique si les accès des autres sites sont finis. Rappelons que la propriété d'équité faible implique celle de la progression.

Plan de la preuve : l'idée de cette partie de la preuve est d'utiliser les propriétés des positions. On procèdera en plusieurs étapes :

1. Pour commencer, nous montrerons que notre algorithme garantit que toute perte du jeton sera suivie d'une régénération du jeton (lemme 3).
2. Puis, nous montrerons que notre algorithme garantit à tout site correct, ayant obtenu une position, d'accéder à la section critique en un temps fini, quel que soit le nombre de sites défaillants (lemme 4).
3. Ensuite, nous montrerons qu'en absence de défaillance et d'expiration du temporisateur *CommitTimer*, un site correct désirant entrer en section critique finit toujours par recevoir une position (lemme 5).
4. Nous montrerons que l'expiration du temporisateur *CommitTimer* (mécanisme *M3-SearchQueue*) conduit à retrouver un état du système assurant au site correct de pouvoir acquérir une position (lemme 6).
5. Enfin, nous montrerons que tout site désirant accéder à la section critique fini toujours par obtenir une position (lemme 7).

Lemme 3. *Si le système est dépourvu de jeton et qu'au moins un site correct désire accéder à la section critique, alors un de ces sites attendant le jeton le re-génèrera dans un temps fini quel que soit le nombre de sites défaillants.*

Démonstration. On suppose ici qu'au moins un des sites corrects désire accéder à la section critique, qu'aucun site ne possède le jeton et qu'il n'y a pas de message *TOKEN* en transit. On distingue alors deux cas :

- *aucun site ne possède une position :* Dans ce cas, aucun des sites attendant la section critique n'a reçu d'acquittement. Or, ces sites ont armé un temporisateur *CommitTimer* à l'émission de leur requête. Dans un temps fini ce temporisateur va donc expirer et déclencher le mécanisme *M3-SearchQueue*. Puisque chacun des sites lançant ce mécanisme ajoute son propre identifiant dans l'estampille du message *SEARCH_QUEUE*, un de ces messages contient une estampille plus grande que tous les autres (suivant la relation d'ordre total définie à la section 4.5.5). De plus, le dimensionnement des temporisateurs évitant qu'un site battu ne relance une élection concurrente, un site finira par voir expirer son temporisateur *ReconnectionTimer*. Et puisqu'aucun site ne possède de position, il ne peut avoir reçu de réponse *POSITION*. Il re-générera donc le jeton dans un temps fini : écoulement des deux temporisateurs *CommitTimer* et *ReconnectionTimer*.
- *au moins un des sites possède une position :* On suppose alors que le site correct S_i possède la plus petite des positions des sites corrects. S_i a donc traité un message *COMMIT*, ce qui a armé un temporisateur *TokenTimer*. D'autre part, le lemme 2 assure que les prédécesseurs sont valides. S_i ne peut donc pas connaître de prédécesseurs corrects. À l'expiration du temporisateur *TokenTimer*, le mécanisme *M1-SearchPred* ne trouvera donc pas de prédécesseur et lancera après un temps fini, le mécanisme *M2-SearchPos* qui arme un temporisateur *ReconnectionTimer*.

Puisqu'il possède la position la plus petite, il ne recevra pas de réponse *POSITION* et re-générera donc le jeton dans un temps fini : écoulement des deux temporisateurs *CommitTimer*, échec de la recherche d'un prédécesseur connu correct (voir section 4.5.3) et *ReconnectionTimer*.

Dans les deux cas, le jeton sera bien régénéré dans un temps fini. □

Lemme 4. *Quel que soit le nombre de sites défaillants, tout site correct attendant la section critique et ayant obtenu une position accédera en un temps fini à la section critique.*

Démonstration. On considère un site correct S_i attendant la section critique et ayant reçu la position p. Suivant le lemme 1, on peut dire que S_i conserve cette position tant qu'il n'a pas accédé à la section critique. De plus, le lemme 2 assure que le jeton et les postions sont valides. Le jeton, s'il existe, est donc détenu par le site possédant la plus petite position. À la fin de sa section critique, ce site enverra le jeton à son *NEXT* qui possède (ou recevra) une position strictement plus grande puisque l'ensemble des positions est valide. La valeur de la position du site exécutant la section critique étant croissante et le jeton en cas de disparition étant re-généré en un temps fini par un site attendant la section critique (lemme 3), elle finira donc par atteindre la valeur p. Le site S_i accédera donc dans un temps fini à la section critique. □

Lemme 5. *En absence de défaillance et d'expiration du temporisateur CommitTimer, un site correct désirant entrer en section critique finit toujours par recevoir une position.*

Démonstration. En absence de défaillance et d'expiration du temporisateur *CommitTimer*, notre algorithme se comporte comme l'algorithme original de Naimi-Tréhel. La propriété de vivacité de cet algorithme implique donc que tout site S_i correct désirant posséder le jeton finira par le recevoir d'un site S_j. La requête de S_i arrivera donc au site S_j. Dans notre algorithme, S_j lui enverra donc sa position dans un acquittement ou directement dans le jeton. □

Lemme 6. *L'expiration du temporisateur CommitTimer lance un mécanisme M3-SearchQueue qui permet retrouver, en un temps fini, un état du système qui garantit à tous les sites corrects le désirant d'obtenir une position en un temps fini, s'il n'y a pas d'autre défaillance ni expiration de temporisateur CommitTimer.*

Démonstration. Comme précédemment, l'idée de la preuve de ce lemme est d'utiliser la vivacité de l'algorithme original de Naimi-Tréhel. Mais contrairement au lemme précédent, le déroulement de l'algorithme de Naimi-Tréhel peut être modifié par l'expiration d'un temporisateur *CommitTimer* dans un site qui n'a pas reçu de position. Dans ce cas, l'envoi de messages *SEARCH_ QUEUE* par le site S_i propage une nouvelle estampille temporelle dans le système. Les hypothèses temporelles garantissent, qu'à l'expiration du temporisateur sur le site S_i, toutes les requêtes en transit, portant l'ancienne estampille, ont été annulées (non traitées par un site). De plus, à la réception du message *SEARCH_ QUEUE*, les sites n'ayant pas de position ont annulé leur *NEXT*. Il ne reste alors de l'ancien système que la chaîne des *NEXT* formée par les sites possédant une position. Les autres sites corrects désirant accéder à la section critique ont ré-émis leur requête vers S_i. Enfin une

forêt des $LAST$ a été construite autour de ces ré-émissions. Puisque l'ensemble des sites ont soit un $LAST$ vide, soit S_i pour $LAST$, cette forêt exclut donc tous les sites défaillants.

Il nous faut donc montrer que l'état du système, permettra au site le désirant d'obtenir une position. Autrement dit que cet état (chaîne des $NEXT$, arbre des $LAST$ et requête en transits) soit un état possible de l'algorithme de Naimi-Tréhel. Nous introduisons donc une bijection entre cet état $\mathcal{E}_{M3-SearchQueue}$ et $(\mathcal{A}_{\text{initial}}, \mathcal{S}_{\text{req}})$, où :

- $\mathcal{A}_{\text{initial}}$ est un état initial : un arbre des $LAST$ unique formé de sites n'attendant pas la section critique et dont la racine possède le jeton.
- $\mathcal{S}_{\text{req}}$ est une séquence d'envois de requêtes ainsi que de leur traitement pour un certain nombre de sites du système.

L'arbre $\mathcal{A}_{\text{initial}}$ s'obtient alors de la façon suivante :

- tous les sites ayant une position dans $\mathcal{E}_{M3-SearchQueue}$ prennent pour $LAST$ dans $\mathcal{A}_{\text{initial}}$ leur prédécesseur dans $\mathcal{E}_{M3-SearchQueue}$.
- le site S_i prend pour $LAST$ le site qui possède le jeton dans $\mathcal{E}_{M3-SearchQueue}$.
- tous les autres sites prennent S_i pour $LAST$.

On considère alors la séquence de requêtes $\mathcal{S}_{\text{req}}$ suivante :

- successivement, tous les sites possédant une position dans $\mathcal{E}_{M3-SearchQueue}$ émettent une requête (qui sera traitée avant l'émission de la suivante) dans l'ordre croissant des positions.
- l'ensemble des sites attendant la section critique mais n'ayant pas de position dans $\mathcal{E}_{M3-SearchQueue}$, émettent une requête qui reste en transit.

L'exécution par l'algorithme de Naimi-Tréhel de la séquence de requêtes $\mathcal{S}_{\text{req}}$ sur l'arbre $\mathcal{A}_{\text{initial}}$ donne l'état $\mathcal{E}_{M3-SearchQueue}$. On peut donc dire qu'en absence de toute nouvelle défaillance et de toute nouvelle expiration du temporisateur *CommitTimer*, l'ensemble des sites désirant obtenir la section critique accédera à une position en un temps fini. □

Lemme 7. *Tout site correct qui désire accéder à la section critique finit toujours par obtenir une position.*

Démonstration. Un site correct désirant accéder à la section critique et qui ne reçoit pas de position finira toujours par déclencher le mécanisme *M3-SearchQueue* à l'expiration de son temporisateur *CommitTimer*. Le lemme 6 garantit à ce site d'obtenir une position en absence de nouvelle défaillance et si le dimensionnement du temporisateur *CommitTimer* permet l'acheminement de cet acquittement. Or, nous sommes en système synchrone, on peut donc dimensionner le temporisateur de *CommitTimer* de manière à ce qu'une requête finisse par atteindre la racine. De plus, le système est ici statique. Le nombre de pannes franches est donc limité à $n-1$. Il arrive donc un instant où il n'y aura plus de défaillance. Les sites corrects finissent donc tous par recevoir une position. □

Théorème 2. *Si les durées des sections critiques sont finies, tout site correct qui désire accéder à la section critique, finira par y accéder en un temps fini quel que soit le nombre de défaillances.*

Démonstration. Puisque le lemme 7 garantit à tous les sites désirant d'accéder à la section critique d'obtenir une position et puisque le lemme 4 garantit que tout site ayant obtenu une position est assuré d'accéder à la section critique quel que soit le nombre

de défaillances, on peut dire que notre algorithme vérifie la propriété d'équité faible de l'exclusion mutuelle. □

Chapitre 5

Évaluation des performances

Sommaire

Dans ce chapitre nous allons tester et évaluer notre algorithme tolérant aux défaillances. Dans une première section nous présenterons la plate-forme de test qui a servi à la réalisation de ces expérimentations. Puis, dans une deuxième section nous étudierons les résultats obtenus.

5.1 Plate-forme d'expérimentations *G-Mutex*

Afin de réaliser l'étude comparative des deux approches présentées dans le chapitre 4, nous avons réalisé une plate-forme générique d'évaluation d'algorithmes d'exclusion mutuelle : *G-Mutex*.

Cette plate-forme a été réalisée en C suivant l'architecture présentée dans la figure 5.1. Nous y avons isolé, sous forme de bibliothèques : la couche de communication, l'algorithme

d'exclusion mutuelle, l'utilisation applicative de cet algorithme, un injecteur de fautes et un service de journalisation. Cette architecture modulaire, permet à la plate-forme de s'adapter à de multiples environnements distribués : émulation d'un système distribué sur une machine, déploiement réel dans un cluster ou dans une grille de calculs ou encore utilisation d'un simulateur à temps discret. Elle pourra aussi être facilement modifiée pour tester d'autres classes d'algorithmes distribués.

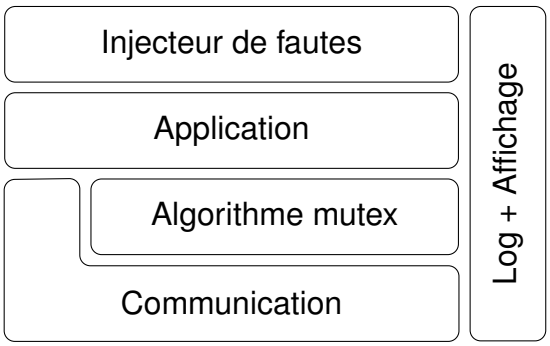


FIGURE 5.1 – Plate-forme générique permettant d'évaluer des algorithmes d'exclusion mutuelle

5.1.1 Module de communication

Le module de communication offre au reste de la plate-forme des primitives de communication : envoi, diffusion et réception de messages. L'utilisation de ces fonctions repose sur un service de nommage, lui aussi fourni par ce module. Chaque nœud reçoit : un identifiant global unique, un identifiant de groupe et un identifiant local (unique dans son groupe). Les trois fonctions de communication trouvent donc deux déclinaisons suivant qu'il s'agisse d'une communication globale ou interne au groupe.

Chacune de ces instances est accompagnée d'un service de déploiement et le module offre une fonction d'initialisation qui reste bloquante tant que l'ensemble de ses fonctionnalités n'est pas opérationnel. Pour simplifier l'implémentation des modules des couches supérieures, l'ensemble des fonctions offertes doit être *thread-safe*.

Plusieurs implémentations de ce module ont été réalisées. Elles utilisent :
- l'interface MPI [For94] et le lanceur runmpi. [14]
- des sockets UDP et un lanceur implémenté en C.
- des sockets TCP et un lanceur implémenté en C.

5.1.2 Module d'exclusion mutuelle

Chaque algorithme d'exclusion mutuelle est implémenté sous forme d'une bibliothèque offrant aux applications l'interface générique suivante :

14. Si la deuxième version de cette interface (MPI2 [For97]) prévoit une implémentation *thread-safe*, ce n'était malheureusement pas le cas des implémentations libres, pendant les premières années de nos travaux. Il n'a donc pas été possible d'utiliser les fonctionnalités offertes par ces implémentations de MPI.

- *requestCS()* : l'appel à cette fonction reste bloquant tant que l'appelant ne peut accéder à la section critique
- *releaseCS()* : l'appel à cette fonction libère la section critique.
- *msgProcessing(msg_t *msg)* : fonction permettant à l'application de transmettre un message de l'algorithme aux modules.

5.1.3 Module applicatif

Le module applicatif permet de simuler l'application utilisant l'algorithme d'exclusion mutuelle. Outre une fonction d'initialisation, ce module offre à la plate-forme une fonction *runBot()* dont l'appel reste bloquant jusqu'à la fin de l'expérience. Les implémentations de ce module doivent donc intégrer un algorithme de terminaison.

Plusieurs instances de ce module ont été implémentées :

- une version en ligne de commande : chaque application est lancée dans un terminal au travers duquel sont faites les demandes d'accès à la section critique.
- une version graphique : pour l'ensemble des sites, les demandes d'accès à la section critique sont commandées au travers d'une interface graphique (réalisée en *Java Swing*). Cette interface permet aussi d'afficher les données de l'algorithme (via une interface optionnelle d'affichage).
- un contrôle par scenarii : chaque application exécute une série d'accès à la section critique suivant un scénario. Ce dernier fixe les durées et les délais entre deux accès. Un générateur aléatoire de scénario a été implémenté. Les deux valeurs choisies suivent deux distributions de Poisson dont les valeurs moyennes sont paramétrables.

5.1.4 Module de journalisation et module d'affichage

Un module servant à la journalisation d'informations permet à chacun des autres modules de la plate-forme d'enregistrer les mesures relatives à l'expérience. Dans sa version exhaustive, la date de chaque accès à la section critique ainsi qu'une valeur associée au nombre d'utilisations d'un jeton sont enregistrées. Une application autonome permet alors de vérifier la propriété de sûreté (unicité du jeton) en analysant les journaux.

Un module d'affichage permet à l'ensemble de la plateforme d'afficher des informations sur l'état courant. Le cas échéant, ces informations peuvent être préfixées par l'identifiant du nœud.

5.1.5 Module d'injection de fautes

Le module d'injection offre à chaque processus de l'application, une variable globale indiquant son état : *correct* ou *défaillant*. Le passage de l'état correct à l'état défaillant est contrôlé suivant les implémentations : par le terminal, par une interface graphique, par un signal ou par une date. L'injection synchronisée de plusieurs défaillances peut se faire par date, par signal ou par diffusion d'un message.

Enfin, notons que ce module offre à l'application une interface permettant de forcer la défaillance. Il est donc possible d'injecter la défaillance à un endroit précis du code

applicatif. De plus, suivant l'implémentation cette défaillance peut entraîner la défaillance d'autres nœuds.

5.2 Évaluation des performances

Après avoir comparé théoriquement notre algorithme à l'algorithme de Naimi-Tréhel tolérant aux défaillances, nous présentons maintenant une étude de performance comparative de ces deux approches. Cette étude portera sur trois axes : l'impact du type d'application sur ces algorithmes, le dimensionnement des temporisateurs de détection et le dimensionnement du paramètre k de notre extension (nombre de prédécesseurs connus).

5.2.1 Spécification des expériences

5.2.1.1 Environnement

L'ensemble des expériences présentées ici a été réalisées sur une grappe (*cluster*) dédiée du Laboratoire d'Informatique de Paris VI (LIP6). Cette grappe est composée de 20 nœuds. Chacun de ces nœuds est équipé de deux processeurs Xeon à 2.8Ghz et de 2Go de Ram. Ils fonctionnent tous sur un noyau Linux 2.6 et sont reliés par un réseau ethernet dédié de 1 Gbit/s. En activant l'*hyperthreading* sur chacun des 40 processeurs, nous émulons un système composé de 80 nœuds logiques ($N = 80$).

Nous utilisons ici la plate-forme de test *G-Mutex* présentée à la section 5.1. Parmi les différents modules offerts, nous avons utilisé : l'implémentation UDP de la bibliothèque de communication, la simulation d'application par scenarii, ainsi que le module d'injections de fautes synchronisées. D'autre part, l'utilisation de la version *exhaustive* de la journalisation nous a permis d'utiliser l'utilitaire de vérification de la sûreté (présenté à la section 5.1.4).

Pour cette étude de performance, nous utiliserons l'implémentation de notre algorithme qui contient le mécanisme de pré-acquittement.

5.2.1.2 Protocole

Ces expériences suivent le protocole suivant : pour chaque expérience, tous les nœuds *corrects* accèderont 5 fois à la section critique, soit 400 sections critiques au total. Ces demandes d'accès à la section critique sont définies par un scénario. Comme nous l'avons expliqué à la section 5.1.3, les différents délais d'un scénario suivent des lois de Poisson dont les moyennes paramètrent les expériences.

L'ensemble des scenarii est rejoué pour chacune des deux extensions en injectant simultanément : aucune, 1, 3, 5, 8, 20 ou 40 défaillances. Ces défaillances sont injectées au milieu de l'expérience sans faute, la durée de cette expérience étant estimée préalablement suivant la théorie des files d'attente. Les nœuds victimes de défaillance n'exécuteront donc pas leurs 5 sections critiques.

Enfin, chacune des expériences a été exécutée vingt fois. Les résultats présentés ici représentent la moyenne des mesures de ces expériences.

5.2.1.3 Paramètres

En plus du paramètre k déjà présenté à la section 4.5.1 et propre à notre extension, nous introduisons ici deux autres paramètres de comparaisons.

Le premier est basé sur le **degré de parallélisme** de l'application. Ainsi, une application répartie utilisant un algorithme d'exclusion mutuelle se caractérise par :

- α : le temps moyen d'exécution d'une section critique
- β : le temps entre la fin d'une section critique et l'émission d'une nouvelle requête sur le même nœud.
- ρ : le ratio β/α

Ainsi, nous avons considéré trois types d'application qui diffèrent par leur degré de parallélisme (respectivement petit, moyen et grand). Les valeurs de ρ correspondant à ces applications sont respectivement :

- $\rho = 1$: une application à faible degré de parallélisme dans laquelle presque tous les sites attendent la section critique. La file des *NEXT* est alors longue.
- $\rho = N$: une application à parallélisme moyen dans laquelle quelques sites attendent la section critique. La file des *NEXT* est petite.
- $\rho = 2 * N$: une application hautement parallèle dans laquelle il n'y a que peu ou prou de concurrence d'accès à la section critique. La file des *NEXT* est alors vide ou réduite à un seul site.

Le deuxième paramètre est basé sur le **dimensionnement des temporisateurs de détection**. Nous définissons trois valeurs de temporisateurs en fonction du temps moyen $pingAvg$ (50ms) et du temps maximum $pingMax$ (150ms) de transmission du message aller d'une requête "ping" dans notre grappe. :

- Temporisateur passif : $PassT = (N-1) * pingMax = 11.85s$
- Temporisateur intermédiaire : $InterT = (N-1) * pingAvg = 3.95s$
- Temporisateur agressif : $AggrT = log2(N) * pingAvg = 0.32s$

Pour chaque expérience, on utilisera la même valeur pour les temporisateurs *CommitTimer* et *TokenTimer*. Le timer de reconnexion *ReconnectionTimer* sera quant à lui fixé à $1s$ (*i.e.*, $6 * pingMax$ pour faire un aller-retour).

5.2.1.4 Métriques

Dans chaque étude nous considérerons trois métriques :

- **le nombre de messages envoyés :** une diffusion est comptée comme un seul message envoyé.
- **le nombre total de messages reçus :** une diffusion est comptée comme n messages reçus, où n est le nombre de sites composant le système.
- **le temps moyen d'attente du jeton :** délais moyen s'écoulant entre l'envoi d'une requête et l'accès à la section critique.

La différence entre le nombre de messages envoyés et reçus caractérisera donc l'utilisation de diffusion. Considérer séparément ces deux métriques va nous permettre d'analyser le comportement des extensions dans des systèmes munis, ou non, d'un service de diffusion.

Il est ici à remarquer que, contrairement aux études de performances de la deuxième partie, nous ne présenterons pas ici de mesure des écarts types. En effet, les écarts, entre les sites qui souffrent des défaillances et les autres, rendent difficilement interprétable cette métrique.

Notons pour finir que la mesure du temps d'attente, en absence de défaillance, est directement liée à la taille de la file d'attente et donc à la valeur du paramètre ρ.

5.2.2 Effet de bord de l'injection de défaillance sur les métriques

Pour étudier les résultats présentés dans les sections suivantes, il nous faut garder à l'esprit l'effet de bord induit par l'injection des défaillances. En effet, ces dernières ont pour effet direct de diminuer le nombre de participants à la deuxième partie de l'expérience (après l'injection des défaillances). La conséquence sur les mesures est double.

D'une part, le nombre de sections critiques totales exécutées va dépendre directement du nombre de défaillances : plus il y aura de sites défaillants, moins il y aura d'accès à la section critique sur la durée de l'expérience.

D'autre part, cette diminution du nombre de participants a aussi pour effet de diminuer la concurrence d'accès à la section critique. Il y a alors moins de nœuds dans l'arbre des $LAST$ et donc moins de retransmissions de requêtes. De plus, la file d'attente se trouve réduite, ce qui diminue naturellement les délais pour accéder à la section critique.

Ces effets ne pouvant être corrigés par une simple pondération, il faudra donc en tenir compte lors de l'analyse des résultats des différentes expériences. Notons que cela sera facilité par l'aspect comparatif de cette étude : à nombre identique de défaillances, les conditions sont les mêmes pour les deux extensions.

5.2.3 Impact du type d'application

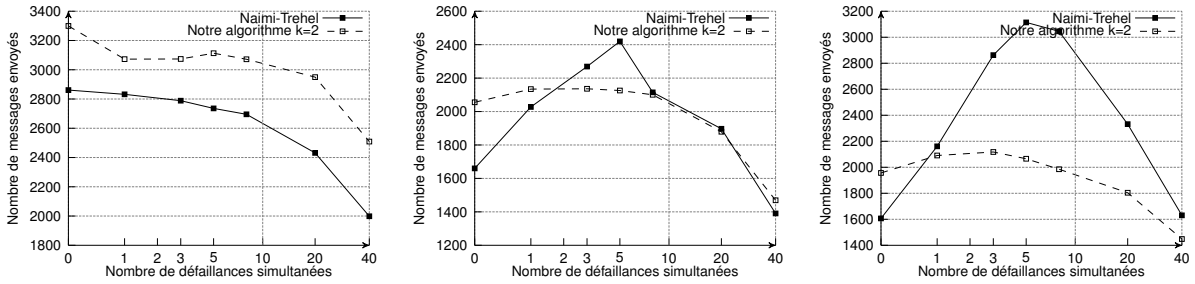
Dans cette première étude, nous allons nous intéresser à l'influence du degré de parallélisme sur chacune des deux extensions. Nous avons donc renouvelé les mêmes expériences en faisant varier le paramètre ρ suivant les trois valeurs présentées ci-dessus. Les autres paramètres restent quant à eux fixes : la durée des temporisateurs détection ($CommitTimer$ et $TokenTimer$) est fixé à $InterT$ (3.95 s) et le nombre k de prédécesseurs connus est fixé à 2.

Pour chacune des métriques étudiées, nous retrouverons trois graphiques correspondant à ces trois valeurs du paramètre ρ. Sur chacun de ces graphiques, sont présentées les performances respectives des deux extensions suivant le nombre de défaillances injectées : aucune, 1, 3, 5, 8, 20 ou 40 (abscisse des graphes).

5.2.3.1 Nombre de messages envoyés

Nous commençons l'étude de l'impact du degré de parallélisme par la mesure du nombre total de messages émis au cours de l'expérience. Les figures 5.2(a), 5.2(b) et 5.2(c) présentent cette métrique pour des valeurs de ρ respectivement fixées à 1, N et $2N$.

En absence de défaillance et donc de recouvrement, on observe une diminution du nombre de messages envoyés lorsque ρ augmente, et ce quelle que soit l'extension consi-



(a) $\rho = 1$ (b) $\rho = N$ (c) $\rho = 2N$

FIGURE 5.2 – Impact du type d'application : Nombre de messages envoyés

dérée. Ce phénomène caractérise une interaction entre le paramètre ρ et les mécanismes de surveillance. En effet, la taille de la file d'attente est proportionnelle au degré de parallélisme. La diminution du ratio ρ engendre donc une augmentation du temps moyen d'obtention de la section critique, et par conséquent une augmentation du nombre de fausses suspicions.

Si l'on compare maintenant sur ces expériences sans défaillance les deux approches, on observe que notre algorithme présente un surcoût en nombre de messages envoyés. Ces messages supplémentaires résultent des mécanismes d'acquittement et de pré-acquittement.

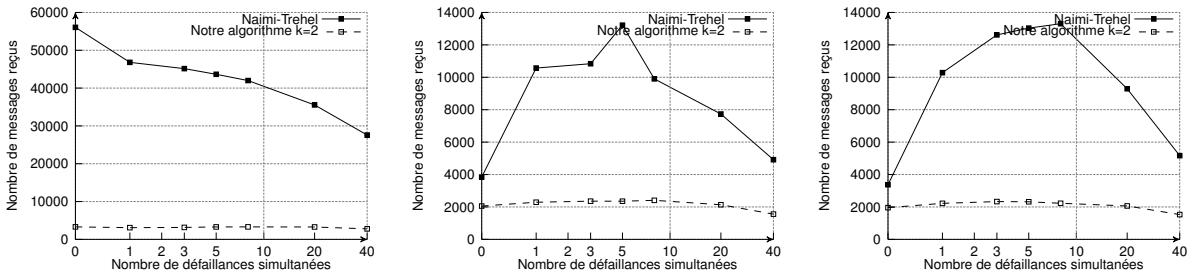
Quand $\rho = 1$ avec un petit nombre de défaillances, la différence entre les deux algorithmes diminue (figure 5.2(a)). Rappelons que pour ce type d'application peu parallélisées, la taille de la file d'attente est très importante. Il y a donc une forte probabilité qu'elle soit touchée par une défaillance. Dans notre solution, la connaissance des prédécesseurs permet une réparation locale de la chaîne des *NEXT*. Ce type de recouvrement se montre alors moins coûteux en termes de messages émis que la ré-initialisation totale des structures de l'extension de Naimi-Tréhel. Cependant, cette stratégie atteint sa limite lorsque 50% des nœuds sont en défaillance. La longue file étant criblée de défaillances, il devient alors moins coûteux en nombre de messages envoyés de réinitialiser tout le système que de la réparer.

Lorsque ρ augmente, le nombre de sites qui ne sont pas en attente du jeton augmente aussi. Ces sites sont alors amenés à envoyer des requêtes le long de l'arbre des *LAST* endommagé par les défaillances. Sur les graphes 5.2(b) et 5.2(c), on observe une légère augmentation du nombre de messages émis avec l'injection d'une défaillance, puis un baisse résultant de la diminution du nombre de sites corrects (voir section 5.2.2). Cette limitation de l'augmentation du coût du recouvrement lorsque le nombre de défaillances augmente, s'explique par le *recouvrement global* induit par nos mécanismes. En effet, la perte d'une seule de ces requêtes lance une reconstruction de l'arbre des *LAST* autour du site élu (mécanisme *M3-SearchQueue*, voir section 4.5.5).

À l'inverse, dans l'extension de Naimi-Tréhel, chaque perte de requête lance un ou plusieurs recouvrements individuels (voir section 4.4.2). Rappelons aussi que les sites défaillants ne sont pas directement exclus de l'algorithme, les pertes de requêtes vont donc s'étaler dans le temps. Avec l'augmentation du nombre de défaillances et donc la dégradation de l'état de l'arbre des *LAST*, le nombre de *recouvrements individuels* devient

de plus en plus important. Cependant, il arrive un moment où le nombre de défaillances finit par entraîner la perte du jeton. Le mécanisme de ré-initialisation global prend alors la place des recouvrements individuels. Ceci explique les pics dans l'algorithme de Naimi-Tréhel. Comme on peut le voir sur la figure 5.2(c), ce comportement s'intensifie lorsque ρ augmente, puisque la taille de la file diminue et avec elle la probabilité de perdre le jeton.

5.2.3.2 Nombre de messages reçus



(a) $\rho = 1$ (b) $\rho = N$ (c) $\rho = 2N$

FIGURE 5.3 – Impact du type d'application : Nombre de messages reçus

Comparons maintenant sur les figures 5.3(a), 5.3(b) et 5.3(c) le nombre total de messages reçus par les sites du système pour les différentes valeurs de ρ. Une première analyse globale de ces courbes montre que notre algorithme se montre toujours le moins coûteux en messages reçus, ce qui marque une utilisation réduite de la diffusion.

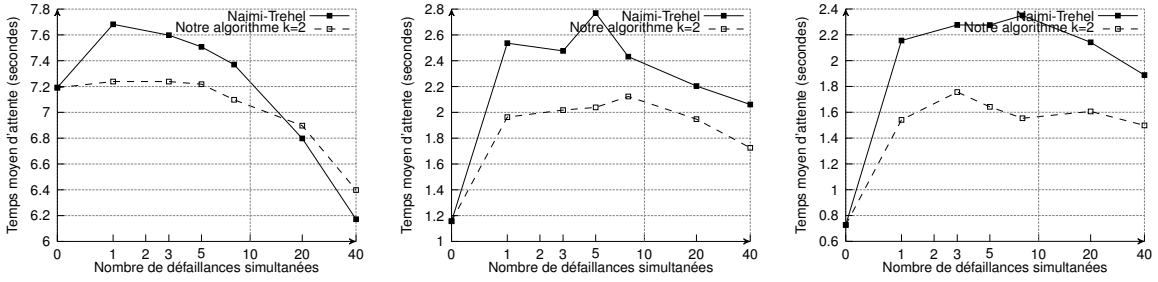
Comme pour l'analyse du nombre de messages envoyés, nous allons pouvoir étudier le coût des mécanismes de détection en considérant plus spécifiquement les scenarii sans défaillance. Pour Naimi-Tréhel, quand $\rho = 2N$, le nombre de messages reçus est de 3.700 mais pour $\rho = 1$, ce chiffre atteint les 56.000 messages. À l'inverse notre algorithme est peu sensible aux changements de comportement. Lorsque ρ diminue le *temps d'attente du jeton* augmente et avec lui le nombre de fausses suspicions. Or, à chaque suspicion d'une défaillance, l'algorithme de Naimi-Tréhel procède à une diffusion du message *CONSULT* tandis que notre algorithme utilise un simple *ping* du prédécesseur. Cette limitation du coût de la surveillance dans un système dépourvu de service de diffusion couvre alors le surcoût du mécanisme d'acquittements.

Si l'on regarde maintenant les performances avec des défaillances, on mesure l'utilisation des diffusions dans les mécanismes de recouvrement. Pour l'algorithme de Naimi-Tréhel, on retrouve ici de façon amplifiée la limitation observée pour le nombre de messages émis, à savoir : une très forte charge lorsqu'il n'y a pas de ré-initialisation du système.

On peut donc conclure de ces deux premières études que le *temps d'attente du jeton* et, implicitement avec lui, la durée en section critique ont une grande influence sur la complexité de l'extension proposée par Naimi-Tréhel. Au contraire, notre algorithme se montre peu sensible au type d'application. De ce point de vue il apparaît donc comme plus polyvalent.

5.2.3.3 Temps moyen d'attente du jeton

L'étude des figures 5.4(a), 5.4(b) et 5.4(c) permet de comparer les délais d'obtention des deux approches pour les différents types d'application.



(a) $\rho = 1$ (b) $\rho = N$ (c) $\rho = 2N$

FIGURE 5.4 – Impact du type d'application : Temps moyen d'attente du jeton

Ces figures permettent tout d'abord de vérifier que ces deux extensions présentent, en absence de défaillance, le même temps moyen d'attente du jeton. En comparant ces délais, on peut aussi y retrouver la relation liant la taille de la chaîne des *NEXT* et les différents degrés de parallélisme : 7.2s d'attente en moyenne pour $\rho = 1$, 1.2s pour $\rho = N$ et 0.7s pour $\rho = 2N$.

Globalement, on remarque qu'en présence de défaillances notre algorithme présente un temps moyen d'attente du jeton plus faible pour presque toutes les expériences. Cette différence s'explique par des temps de recouvrement plus rapides que ceux de Naimi-Tréhel. Ce dernier est pénalisé par des temporisateurs de recouvrement plus larges ($3T_{msg}$) et par l'accumulation des recouvrements individuels.

Seule exception à cette remarque, l'injection de 50% de défaillances dans une application faiblement parallèle ($\rho = 1$). Dans ce cas, il est alors plus facile de réinitialiser une longue file très endommagée que d'essayer de la réparer.

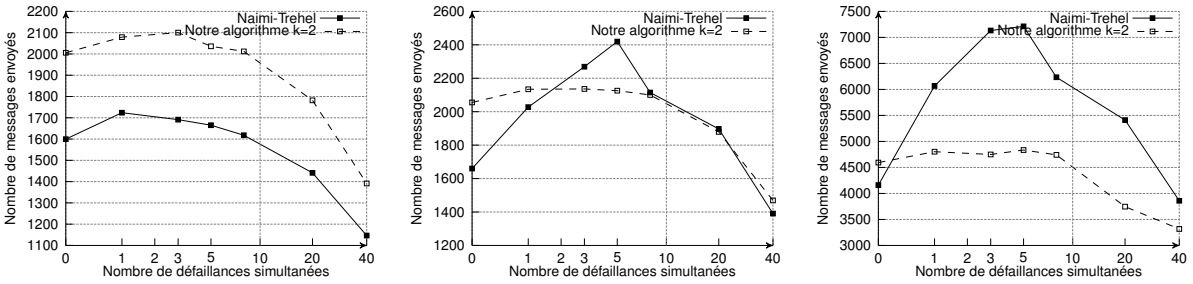
Pour finir cette étude, on peut remarquer un résultat intéressant pour les mesures avec $\rho = 1$: Lorsque le nombre de défaillances varie de 0 à 5, le temps d'attente du jeton pour notre algorithme n'augmente pas (figure 5.4(a)). Ceci illustre bien le phénomène d'**isolation des défaillances** présenté à la section 4.5.7 qui permet de paralléliser le recouvrement des défaillances avec la circulation du jeton. Comme ici la file des *NEXT* est grande, le recouvrement des défaillances est entièrement couvert par le temps d'attente. Aussi lorsque le jeton sera envoyé au site en défaillance, la file sera déjà réparée.

5.2.4 Impact du dimensionnement des temporisateurs

Nous nous proposons maintenant de mesurer l'impact du dimensionnement des temporisateurs de détection sur les performances des deux extensions. Ainsi, nous allons ici considérer des applications à parallélisme moyen ($\rho = N$), et renouveler les expériences avec les trois types de temporisateur : passif ($PassT$), intermédiaire ($InterT$) et agressif ($AggreT$). Enfin nous conservons ici la valeur $k = 2$ pour le nombre de prédécesseurs connus.

Comme dans l'étude précédente, les résultats seront présentés pour chacune des métriques par trois graphiques correspondant aux trois types de temporisateur. Sur ces graphiques nous retrouverons en abscisse, le nombre de défaillances injectées simultanément (0, 1, 3, 5, 8, 20 ou 40).

5.2.4.1 Nombre de messages envoyés



(a) Temporisateurs passifs (b) Temporisateurs intermédiaires (c) Temporisateurs agressifs

FIGURE 5.5 – Impact des temporisateurs : Nombre de messages envoyés

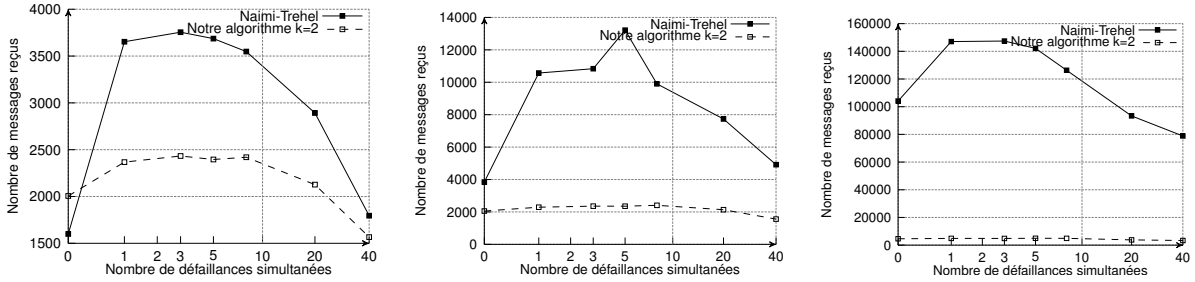
En comparant le nombre de messages envoyés en absence de défaillance sur les figures 5.5(a), 5.5(b) et 5.5(c), on retrouve le surcoût dû aux messages *COMMIT*. Ce surcoût devient proportionnellement moins important lorsque le temporisateur de suspicion diminue. En effet, plus le temporisateur est agressif et plus le nombre de fausses suspicions sera important. Les messages engendrés par ces fausses suspicions deviennent alors proportionnellement beaucoup plus importants que ceux des mécanismes d'acquittement.

Le dimensionnement du temporisateur agit aussi sur le mécanisme de recouvrement. Ainsi, avec un temporisateur relativement large (figure 5.5(a)), la majorité des sites sont en attente du jeton lorsque les défaillances sont enfin détectées. Les comportements des deux algorithmes sont alors relativement semblables : dans les deux cas il y a une élection pour reconstruire l'arbre. À l'inverse, pour des temporisateurs plus agressifs (figures 5.5(b) et 5.5(c)), le coût du recouvrement en nombre de messages pour l'algorithme de Naimi-Tréhel augmente énormément. En effet, pour ce dernier, une diminution du temporisateur s'accompagne d'une augmentation de l'importance des *recouvrements individuels* (voir section 4.4.2). En comparaison, le nombre de messages envoyés reste stable avec notre algorithme.

5.2.4.2 Nombre de messages reçus

Comme nous l'avons déjà vu dans l'étude du paramètre ρ (section 5.2.3.2), l'ensemble des courbes (figures 5.6(a), 5.6(b), et 5.6(c)) met globalement en évidence l'utilisation de diffusions dans l'algorithme de Naimi-Tréhel.

En modifiant le nombre de fausses suspicions, le dimensionnement des temporisateurs a un énorme impact sur le nombre de diffusions engendrées par le mécanisme de surveillance de l'algorithme de Naimi-Tréhel. Ceci représente une vraie limite pour le choix

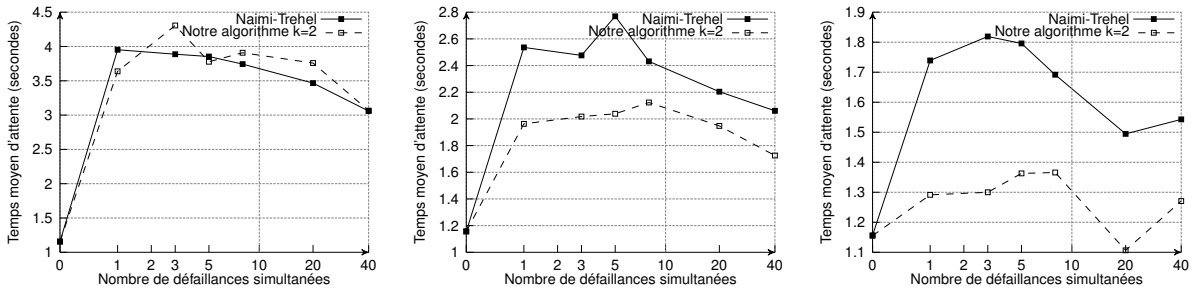


(a) Temporisateurs passifs (b) Temporisateurs intermédiaires (c) Temporisateurs agressifs

FIGURE 5.6 – Impact des temporisateurs : Nombre de messages reçus

du temporisateur chez Naimi-Tréhel. Avec le temporisateur $AggrT$ on dépasse les 100.000 messages. En comparaison notre algorithme est peu sensible au choix du temporisateur et reste aux environs de 2.000 messages reçus.

5.2.4.3 Temps moyen d'attente du jeton



(a) Temporisateurs passifs (b) Temporisateurs intermédiaires (c) Temporisateurs agressifs

FIGURE 5.7 – Impact des temporisateurs : Temps moyen d'attente du jeton

Nous venons d'observer le surcoût, en nombre de messages émis et reçus, résultant de l'utilisation d'un temporisateur agressif. Mais y a-t-il un intérêt à choisir ce type de temporisateur ?

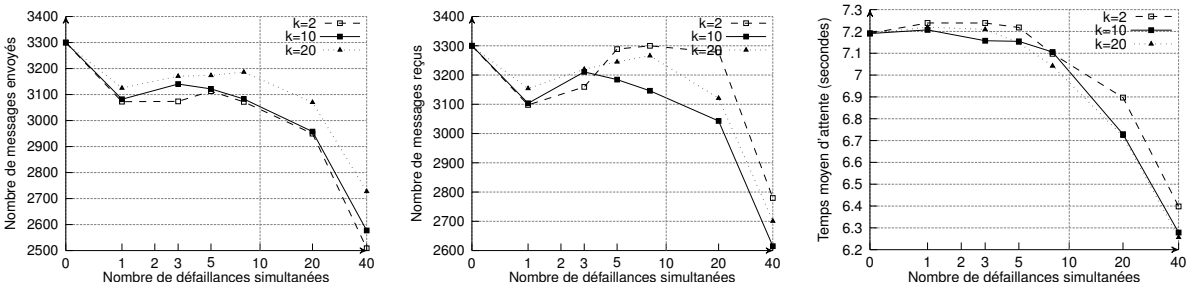
Pour répondre à cette question, nous comparerons le *temps moyen d'attente du jeton* en cas de défaillances pour les trois temporisateurs (figures 5.7(a), 5.7(b), et 5.7(c)). Pour un temporisateur $PassT$, l'ordre de grandeur est de $4s$ pour les deux algorithmes. Si on prend un temporisateur $InterT$, la latence se réduit à $2.1s$ pour notre algorithme contre $2.4s$ pour l'algorithme de Naimi-Tréhel. Enfin, en choisissant $AggrT$, on obtient un temps d'attente de $1.3s$ pour notre solution contre $1.8s$ pour celle de Naimi-Tréhel.

Nous observons donc un réel gain de latence lorsqu'on diminue la durée du temporisateur. Ce gain se révèle par ailleurs légèrement plus grand pour notre algorithme.

5.2.5 Impact du paramètre k fixant le nombre de prédécesseurs connus

Notre algorithme étant basé sur la connaissance de k prédécesseurs, il peut être intéressant de mesurer l'impact de ce paramètre sur les performances de notre algorithme. Ce paramètre étant propre à notre extension, celle de Naimi-Tréhel n'apparaîtra pas dans ces expériences. Ainsi, les figures 5.8(b), 5.8(a) et 5.8(c) présentent respectivement le nombre de messages envoyés, le nombre de messages reçus et le temps moyen d'attente de la section critique pour des valeurs de $k = 2$, $k = 10$, $k = 20$.

Pour étudier l'impact de k sur les mécanismes de recouvrement, nous avons choisi une application où la file des *NEXT* est longue ($\rho = 1$). En effet, avec un ρ plus grand, l'impact devient moins visible. Il est même inexistant avec $\rho = 2N$ puisqu'il n'y a plus de file des *NEXT*. Pour ces expériences, les temporisateurs *TokenTimer* et *CommitTimer* sont fixés à la valeur intermédiaire *InterT*.



(a) Nombre de messages envoyés (b) Nombre de messages reçus (c) Délais d'obtention

FIGURE 5.8 – Impact du paramètre k

Comme nous pouvons l'observer sur l'ensemble de ces figures, en absence de défaillance la valeur de ce paramètre n'a pas d'influence sur l'algorithme. En effet, les informations relatives aux prédécesseurs sont véhiculées sur le même message *COMMIT* et le nombre de prédécesseurs n'influe pas sur le fonctionnement de la détection : "attente du commit" ou "ping du premier prédécesseur".

Lorsqu'il y a quelques défaillances, nous observons que le nombre de messages envoyés augmente proportionnellement à la valeur de k. En effet, le coût de la recherche du premier prédécesseur non fautif augmente avec k.

Pour un nombre de défaillances plus grand, la probabilité d'avoir au moins 2 sites consécutifs défaillants devient plus forte, d'où une augmentation du nombre de messages reçus pour $k = 2$ sur la figure 5.8(b) qui caractérise une augmentation de l'utilisation de la diffusion.

Mais si l'utilisation d'un trop petit nombre de prédécesseurs augmente le nombre de diffusions, l'utilisation d'un trop grand k finit par augmenter le nombre de messages envoyés. Ainsi, sur la figure 5.8(a), on peut voir que le nombre de messages émis pour $k = 20$ augmente sensiblement dès lors qu'il y a plus de 3 défaillances.

Enfin, en présence de nombreuses défaillances, on peut relever un léger gain en latence (Figure 5.8(c)) pour des valeurs élevées de k ($k = 10$ ou $k = 20$).

Cette étude montre que le dimensionnement du nombre de prédécesseurs connus relève d'un compromis entre le coût de la recherche du premier prédécesseur correct (mécanisme *M1-SearchPred*) et la probabilité d'avoir à utiliser la diffusion (mécanisme *M3-SearchQueue*). L'utilisation d'une valeur de k appropriée permet de limiter le coût en messages tout en maintenant de bonnes performances pour les délais de recouvrement.

5.3 Étude théorique du dimensionnement du nombre de prédécesseurs connus

Si notre algorithme a montré de bonnes performances avec de petites valeurs de k, ce dimensionnement reste directement lié à la taille du système : 80 sites dans cette étude. Il peut alors être intéressant de chercher une relation entre le dimensionnement de k, le nombre de sites présents dans le système et le nombre de défaillances.

Pour simplifier les calculs, nous allons considérer uniquement la file d'attente et les défaillances qui s'y produisent. Ainsi nous allons calculer la probabilité $P(k, f, n)$ pour une file de n sites de résister à f défaillances si chaque site connait k prédécesseurs. Autrement dit :

$P(k, f, n)$: la probabilité d'avoir moins de k sites défaillants consécutifs parmi les n sites qui composent la file d'attente sachant qu'il s'y produit f défaillances.

Pour calculer cette probabilité, nous allons considérer son complémentaire, soit la probabilité d'avoir au moins k sites défaillants consécutifs. On définit alors le dénombrement $U_k(f, n)$ comme :

$U_k(f, n)$: le nombre de répartitions des f défaillances dans une file de n sites tel qu'il y ait au moins k défaillances consécutives.

Pour ne pas comptabiliser deux fois la même répartition lors du dénombrement, nous allons considérer l'emplacement du premier site non défaillant (noté *ok*) parmi les k premiers sites de la file. On définit ainsi un partitionnement, présenté dans la figure 5.9, des répartitions des défaillances (notées F) dans la file.

On distingue alors deux cas : soit le premier site correct appartient aux k premiers sites de la file, soit les k premiers sites sont défaillants. Dans ce dernier cas, puisqu'il y a déjà k sites consécutifs défaillants, on peut placer les autres $f-k$ défaillances librement dans les $n-k$ sites restants. A l'inverse, si le premier site correct $(i+1)$ appartient aux k premiers sites de la file, les $f-i$ défaillances restantes doivent être réparties dans les $n-i-1$ derniers sites de façon à avoir au moins k défaillances consécutives ; soit $U_k(f-i, n-i-1)$ répartitions possibles. On obtient donc une définition récursive de $U_k(f, n)$:

$$\forall n \geq k,\ \forall f \geq n,\ U_k(f,n) = \begin{cases} 1 & \text{Si } f = n \\ \sum_{i=0}^{k-1} U_k(f-i, n-i-1) + C_{n-k}^{f-k} & \text{Sinon} \end{cases}$$

Nous avons programmé cette fonction afin d'étudier le dimensionnement du nombre de prédécesseurs connus k. Ainsi, nous avons réalisé une première série de calculs pour

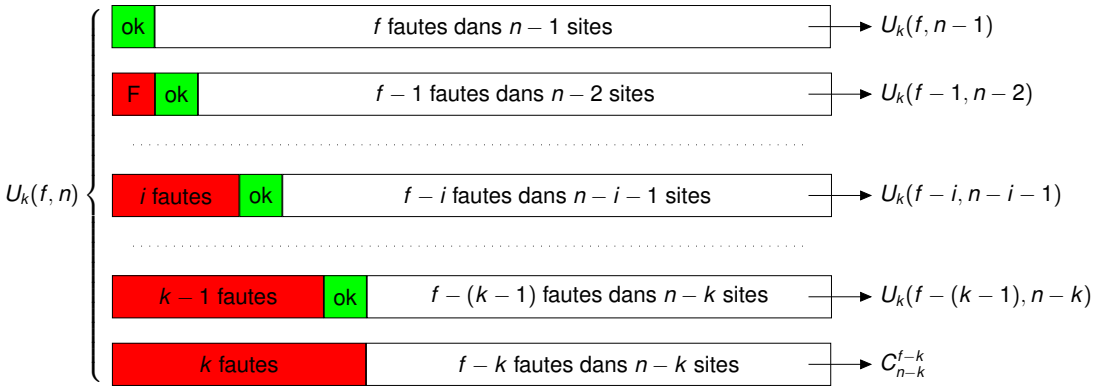


FIGURE 5.9 – $U_k(f,n)$: Nombre de répartitions de f défaillances dans n sites avec au moins k défaillances consécutives.

une file d'attente de 1000 sites. La figure 5.10 présente la probabilité de résister, suivant le nombre de prédécesseurs connus, pour 1%, 10%, 25% et 50% de sites défaillants.

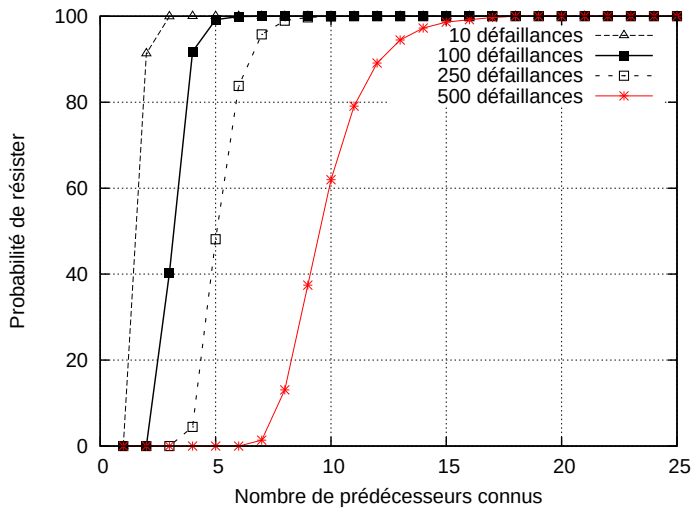
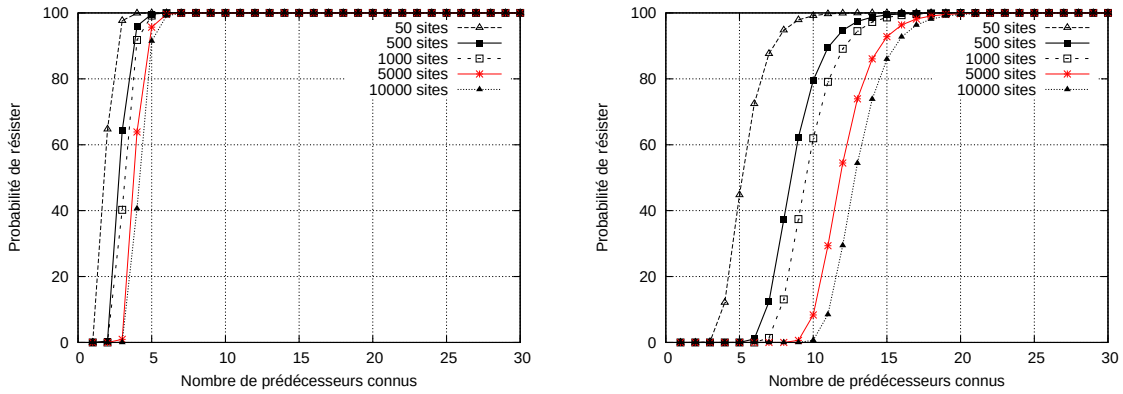


FIGURE 5.10 – Probabilité de résister suivant le nombre de prédécesseurs connus pour une file de 1000 sites.

Sur cette figure on remarque que ces probabilités suivent des fonctions à effet de seuil. Un bon dimensionnement permettra donc de résister aux défaillances avec une probabilité proche de 100%. De plus, une fois dépassé la valeur de ce seuil de résistance, l'ajout de prédécesseurs supplémentaires n'apporte presque plus rien. Enfin, ces courbes montre que les valeurs de k correspondant à ces seuils de résistance sont relativement faible. Ainsi, pour bien résister à 25% de sites défaillants il suffit de connaitre seulement 8 prédécesseurs.

Pour étudier l'évolution du dimensionnement du paramètre k en fonction de la taille du système, nous avons réalisé une deuxième série de calculs. Les figures 5.11(a) et 5.11(b) présentent, pour respectivement 10% et 50% de sites défaillants, la probabilité de résister,

suivant le nombre de prédécesseurs connus, avec des files de 50, 500, 1000, 5000 et 10000 sites.



(a) Pour 10% de défaillances

(b) Pour 50% de défaillances

FIGURE 5.11 – Probabilité de résister suivant le nombre de prédécesseurs connus.

Sur ces deux graphiques on retrouve des fonctions à effet de seuil. Dans le cas d'une défaillance de 10% des sites de la file (figure 5.11(a)), on observe une faible augmentation de ces seuils lorsque le nombre de sites total augmente. Ainsi, le seuil de résistance pour un système composé de 10000 sites est de seulement 8 prédécesseurs. Lorsque le nombre de sites défaillants passe à 50%, les seuils augmentent plus vite lorsque la taille des files augmente et l'effet de seuil est un peu moins marqué. Néanmoins, dans un système de 10000 sites il suffit de connaitre 20 prédécesseurs pour résister à 50% de sites défaillants avec une probabilité proche de 100%.

De cette étude on peut conclure, d'une part, qu'un bon dimensionnement de k doit se situer légèrement après la valeur du seuil de résistance et, d'autre part, que notre mécanisme s'adapte particulièrement bien aux systèmes répartis à grande échelle.

5.4 Conclusions

Dans cette première partie, nous avons présenté un état de l'art sur la tolérance aux défaillances dans les algorithmes distribués d'exclusion mutuelle reposant sur la circulation d'un jeton. Nous avons ensuite proposé une synthèse des hypothèses imposées par ces algorithmes.

Partant de cette étude, nous avons proposé un nouvel algorithme d'exclusion mutuelle tolérant aux défaillances basé sur celui de Naimi-Tréhel. Cet algorithme conserve les bonnes propriétés de passage à l'échelle de l'algorithme initial. Ainsi, le coût moyen pour l'acheminement d'une requête et du jeton reste en $\mathcal{O}(log(n))$. De plus, notre algorithme limite l'utilisation de la diffusion tant pour le recouvrement que pour la surveillance des sites. Enfin, notre algorithme permet lors d'un recouvrement de conserver au maximum les requêtes pendantes. Ceci permet non seulement de réduire le nombre des ré-émissions et donc le coût d'un recouvrement, mais aussi de conserver les bonnes propriétés d'équité

induites par l'utilisation d'une file d'attente répartie. Ainsi, un site ayant obtenu une position à la garantie de voir son ordre d'arrivée respecté quel que soient le nombre et le type de défaillance.

Nous avons conduit plusieurs séries d'expériences pour tester et évaluer notre algorithme dans un environnement réel. Nous avons ainsi montré qu'il conservait de bonnes performances, quel que soit le degré de parallélisme des applications. Nous avons aussi pu vérifier qu'il permettait l'utilisation de temporisateur de détection très court et que ces derniers amélioraient sensiblement les performances de l'algorithme. De plus, il s'est montré, durant l'ensemble de ces expériences, particulièrement bien adapté aux environnements dépourvus de service de diffusion.

Deuxième partie

Algorithme d'exclusion mutuelle adapté aux topologies de type grille

Chapitre 6

Un algorithme générique de composition

Sommaire

Ces dernières années ont vu l'émergence des Grilles de calculs. Ces structures informatiques de grande taille fédèrent un grand nombre de machines regroupées dans des

grappes (cluster). Ainsi, en agrégeant les ressources de milliers d'ordinateurs, les grilles offrent aux applications de nouvelles opportunités en termes de puissance de calcul, de capacité de stockage et de distribution des programmes.

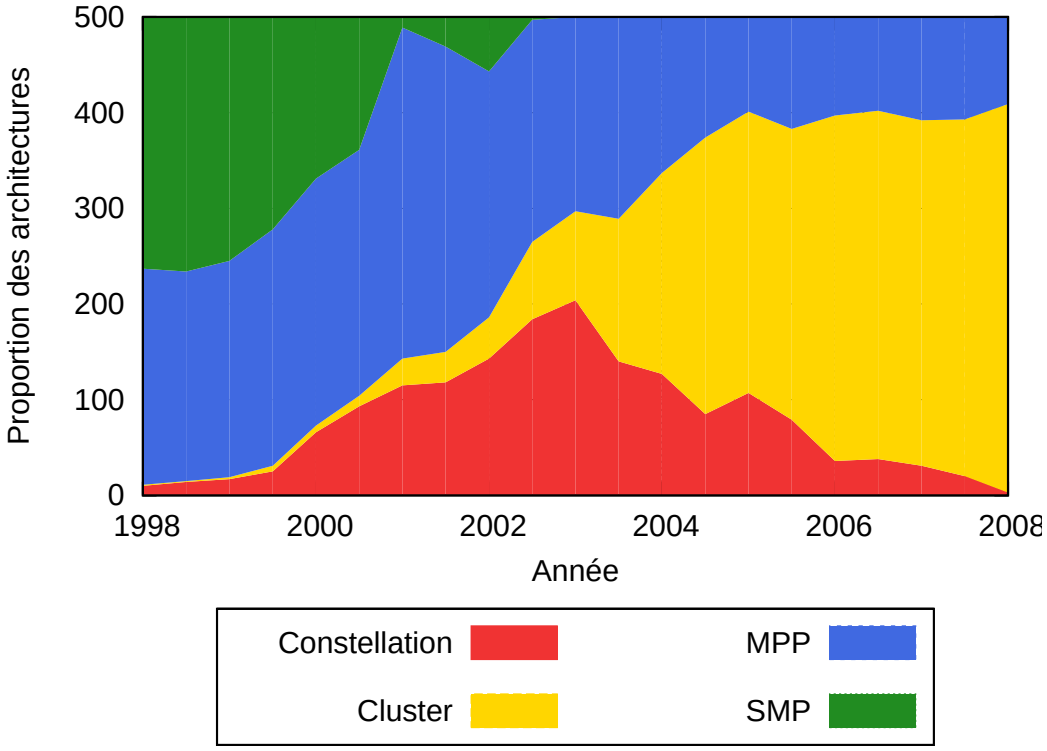


FIGURE 6.1 – Evolution de la répartition du *Top500* suivant l'architecture

Pour mesurer l'importance qu'elles prendront dans l'avenir, on peut étudier l'évolution du type des architectures utilisées dans le *Top500*[Fei05]. Sur la figure 6.1, on remarque, qu'ayant fait leur apparition dans le classement il y a dix ans à peine, les clusters représentent en novembre 2007, 81.20% du *Top500*. Dans le même temps les systèmes mono processeur (SMP) ou communiquant par mémoire partagée (Constellation [15]) ont complètement disparu.

D'autre part, si l'on regarde plus précisément la liste on remarque qu'il y a de plus en plus d'interconnexions de clusters extrêmement distants. D'ailleurs dans ces prospectives Steve Chen lie l'arrivée au petaflop à celle des super-grilles de calcul.

Le principal problème posé par cette architecture réside dans l'hétérogénéité des réseaux de communication : plusieurs ordres de grandeur séparent la latence et le débit internes aux clusters, de la latence et du débit des réseaux reliant ces clusters. Or les algorithmes d'exclusion mutuelle que nous avons présentés dans le chapitre 2 utilisent des abstractions de la topologie physique du réseau et ne prennent pas en compte ces différences.

15. Architecture décrite par Gordon Bell et Jim Gray [BG02] où plusieurs SMP sont reliés par une mémoire partagée de grande taille

Dans ce chapitre, nous présentons les différentes approches existantes qui permettent de tenir compte des contraintes propres aux topologies de types grille. Puis, nous proposons un algorithme générique de composition d'algorithmes d'exclusion mutuelle. Enfin, nous donnerons une preuve générique garantissant que toutes les compositions obtenues respectent les propriétés de sûreté et de vivacité de l'exclusion mutuelle.

6.1 État de l'art

Plusieurs auteurs ont adressé le problème de l'extensibilité de l'exclusion mutuelle en adaptant ou composant des algorithmes existants. Le but est de réduire le nombre de messages ou le temps d'attente pour entrer dans une section critique, en exploitant la topologie logique ou physique des cluster. Fondamentalement, les algorithmes se classifient en deux grandes catégories.

Dans la première, l'ordre des demandes dans la file d'attente est modifié pour satisfaire en premier les requêtes avec une haute priorité. La priorité d'une requête peut alors changer dynamiquement en fonction de sa localité. Ainsi, les demandes issues d'un même cluster peuvent être satisfaites en premier.

Dans la seconde catégorie, les auteurs ne modifient pas un algorithme, mais composent différents algorithmes d'exclusion mutuelle, en les projetant sur la hiérarchie induite par la topologie des grilles avec un algorithme au niveau intra-cluster et un algorithme au niveau inter-clusters.

6.1.1 Algorithmes à priorités

Bertier et al. [BAS04] ont adapté l'algorithme de Naimi-Tréhel en changeant l'ordre des requêtes pendantes dans la file des $NEXT$ afin de satisfaire en priorité les requêtes intra-cluster. Ceci est mis en œuvre par un mécanisme de préemption du jeton par les nœuds locaux à un cluster. Pour éviter la famine et contrôler le degré de localité, un seuil limite le nombre maximum de préemptions. Tant que le nombre de préemptions locales est inférieur au seuil, la file des $NEXT$ est modifiée pour satisfaire en priorité les requêtes locales. Une requête d'entrer en section critique suit l'arbre des $LAST$ jusqu'à ce qu'elle atteigne le dernier nœud du cluster ayant demandé la section critique. Ce nœud est appelé la *racine locale*. Si la variable $NEXT$ de la racine locale existe, elle désigne alors un nœud appartenant à un cluster distant. Lorsque la racine locale reçoit une requête d'un des nœuds de son propre cluster et si le seuil de préemption n'est pas atteint, une préemption locale du jeton a lieu. La racine locale met alors à jour son $NEXT$ pour désigner le nœud initiateur de la requête. Ce dernier devient alors la nouvelle racine locale et hérite du $NEXT$ de l'ancienne racine.

Dans [LMRN06], Moallemi et al. proposent d'appliquer l'approche de Bertier et al. [BAS04] à notre version tolérante aux fautes de Naimi-Tréhel publiée dans [SAS06a]. L'objectif est de fournir un algorithme à jeton tolérant aux fautes et hiérarchique. Les auteurs considèrent qu'il existe au moins un nœud non fautif dans le cluster possédant le jeton et ajoutent des diffusions entre clusters pour transmettre les requêtes et le jeton. Aucune étude de performances n'a été conduite pour évaluer cet algorithme.

D'autres algorithmes permettant de gérer des priorités ont été proposés. S'ils ne sont pas directement utilisables dans une topologie de type grille, il peut être possible de les adapter pour prendre en compte les caractéristiques propre à ces environnements.

Ainsi Mueller propose une extension [lue98] de l'algorithme de Naimi-Tréhel [NT96] dans laquelle il ajoute le concept de priorité. Des files locales sont introduites sur chaque nœud. Elles forment une file virtuelle globale ordonnée en fonction des priorités des demandes puis par un ordre FIFO lorsque les priorités sont égales. Chaque requête possède une priorité et l'algorithme essaye d'abord de satisfaire les requêtes les plus prioritaires. La file des $NEXT$ est donc remplacée par un ensemble de files locales et le jeton accumule les files des nœuds qu'il traverse. Quand un nœud publie une requête, elle se propage dans l'arbre des $LAST$ jusqu'à ce qu'elle atteigne un nœud qui attend le jeton avec une priorité égale ou supérieur. La requête est alors mise dans la file locale de ce nœud. Quand un nœud obtient le jeton, il ajoute sa file locale au jeton. Dans l'article, la notion de proximité physique et de latence entre cluster n'est pas considérée. Toutefois, si les priorités sont assignées aux requêtes en fonction de leur cluster, les requêtes de même priorités seront satisfaites dans l'ordre, entraînant la préemption des requêtes des autres clusters.

D'autres algorithmes pourraient ainsi être adaptés en re-définissant les priorités suivant la localité. Ces modifications pouvant toutefois rendre inefficace l'algorithme original demandent à être évaluées. D'autre part, elles ne peuvent bénéficier directement de la preuve de l'algorithme initial, notamment en ce qui concerne la vivacité. Enfin, si la priorité permet de réduire le délai moyen d'obtention en optimisant les accès à la section critique, elle ne permet pas directement de réduire la complexité en nombre de messages. Ceci peut être une limite au déploiement sur des systèmes à grande échelle.

6.1.2 Composition d'algorithmes

Plusieurs auteurs [CSL90a], [HT01], [BAS06], [Erc04], [MK94] et [ON02] proposent de composer des algorithmes d'exclusion mutuelle avec un algorithme au niveau intra-cluster et un second au niveau inter-clusters. L'objectif est de minimiser le trafic et le temps d'attente avant d'entrer en section critique. Dans la majorité de ces travaux, l'algorithme entre clusters (groupes) est différents de l'algorithme à l'intérieur d'un groupe.

Chang et al. [CSL90a] présentent une approche hybride basée sur deux algorithmes de diffusion différents : l'algorithme de Singhal [Sin92] au sein d'un groupe, et l'algorithme de Maekawa [Mae85] entre les groupes. Le premier s'appuie sur une structure dynamique alors que le dernier repose sur une approche à base de vote. Les auteurs affirment que puisqu'aucun algorithme ne peut minimiser à la fois le nombre des messages et les délais d'obtention de la section critique, leur combinaison est idéale car l'algorithme de Maekawa a une complexité faible en messages tandis que celui de Singhal minimise les délais entre deux exécutions de section critique. Les simulations montrent que, comparé à l'algorithme plat de Maekawa, l'algorithme hybride proposé réduit de façon significative le trafic et le temps d'attente, en particulier si le système possède une bonne localité des requêtes au sein des clusters. De la même façon, la solution de Omara et al. [ON02] est hybride avec l'algorithme de Maekawa au niveau intra-cluster et un algorithme modifié de Singhal assurant une meilleure équité au niveau inter-clusters.

Dans Housni et al. [HT01], les nœuds avec la même priorité sont réunis au sein du

même groupe. L'algorithme entre les groupes est différent de l'algorithme au sein d'un groupe. Ils considèrent que les routeurs assurent l'interface entre les algorithmes des deux niveaux. L'algorithme à jeton de Raymond [Ray89] basé sur un arbre est utilisé à l'intérieur d'un groupe et l'algorithme à diffusion de Ricart-Agrawala [RA81] entre les groupes. Si le routeur reçoit une permission de tous les autres groupes, il génère un jeton et le donne au nœud racine de son groupe. Quand toutes les requêtes en attente dans le groupe sont satisfaites, le jeton est remis au routeur qui le détruit. Les routeurs peuvent avoir des priorités différentes, c'est à dire qu'un nœud d'un groupe donné peut se faire préempter si son routeur reçoit une requête plus prioritaire d'un autre groupe. Ce dernier interrompt alors la racine de son groupe pour lui demander son jeton. Dès que le routeur obtient le jeton, il envoie sa permission au routeur le plus prioritaire.

Erciyes [Erc04] propose une architecture qui consiste en un anneau de clusters, un cluster correspondant à un nœud de l'anneau. Chacun de ses nœuds est représenté par un coordinateur qui exécute les demandes de section critique ainsi que le relâchement du jeton pour l'ensemble des nœuds du cluster. Toutefois, à l'intérieur d'un cluster, un algorithme centralisé d'exclusion mutuelle est utilisé, c'est à dire qu'un nœud demande à entrer en section critique à son coordinateur et y rentre lorsque celui-ci lui répond. Quand un nœud sort d'une section critique, il envoie un message au coordinateur. Les auteurs proposent deux algorithmes pour le niveau inter-clusters : l'algorithme à base de permissions de Ricart-Agrawala [RA81] et l'algorithme à base de jeton de Le Lann [Lan78]. Toutefois, dans les deux cas, les messages sont échangés en suivant la structure en anneau. L'étude théorique de l'algorithme composé comparé à la version plate originale montre, d'une part, un gain d'un ordre de magnitude en termes de complexité, mais au prix d'un temps de réponse plus long et, d'autre part, une diminution du nombre de messages pour entrer en section critique quand le nombre de clusters augmente. Toutefois, avec un grand nombre de clusters et un grand nombre de requêtes concurrentes, le coordinateur devient un goulot d'étranglement et le temps d'obtention du jeton augmente.

Comme dans l'algorithme de Erciyes [Erc04], Madhuram et al. [MK94] présentent un algorithme à deux niveaux avec un algorithme centralisé au niveau intra-cluster. Les coordinateurs du niveau inter-clusters ne sont pas organisés en anneau et peuvent communiquer les uns avec les autres. Les auteurs affirment qu'un algorithme centralisé est un bon choix car la complexité en nombre de message est faible : trois messages sont échangés par demande de section critique. Tous les messages sont estampillés et le coordinateur met dans une file d'attente *InternalQ* les requêtes qu'il reçoit des nœuds de son cluster en fonction de cette estampille. Une seconde file d'attente est utilisée pour stocker les demandes des autres coordinateurs : *GlobalQ*. Comme dans l'algorithme de Erciyes [Erc04], le coordinateur agit au nom de l'ensemble des nœuds du groupe qu'il gère. Dès qu'il reçoit une requête d'un de ses nœuds, le coordinateur la diffuse aux autres coordinateurs, lesquels l'enregistrent dans leur file d'attente globale *GlobalQ*. En séparant les queues internes et externes, les auteurs fournissent aussi un mécanisme de préemption locale pour satisfaire d'abord les requêtes locales avant les requêtes d'autres coordinateurs (en imposant un seuil pour éviter la famine). Ils affirment que en cas de fortes demandes de section critique (correspondant à des applications faiblement parallèles) la préemption améliore les performances de l'application. L'algorithme hiérarchique proposé repose sur une adaptation de l'algorithme de Ricart-Agrawala au niveau inter-clusters, mais les

auteurs affirment que tout algorithme à base d'estampilles peut aussi être utilisé.

Dans [BAS06], Bertier et al. proposent une approche à base de composition qui repose sur l'algorithme de Naimi-Tréhel aux deux niveaux. Les auteurs proposent que les messages échangés par les nœuds des différents clusters passent systématiquement par des nœuds distincts appelés mandataires (proxy) qui se comportent comme des routeurs. Il existe un proxy par cluster. Par conséquent, chaque message envoyé par un nœud A vers un nœud B transite par les proxys de A puis B avant d'être acheminé jusqu'à B. Les proxys peuvent alors collecter des informations sur le passage des jetons entre les clusters et prendre des décisions en fonction de ces informations. Certains messages inter-clusters peuvent donc être traités localement par un proxy sans qu'il soit nécessaire de les acheminer jusqu'au proxy du récepteur. De plus, en modifiant la file d'attente des $NEXT$, les requêtes intra-cluster sont satisfaites avant les requêtes inter-clusters. Pour éviter les famines, un seuil limite le nombre maximum de sections critiques qui peuvent être exécutées au sein du même cluster en présence de requêtes externes. L'expérimentation a été réalisée sur un cluster dédié dans lequel une grille avec des latences hétérogènes a été émulée. Les auteurs ont montré qu'en exploitant la localité, le délai pour obtenir un jeton et le nombre de messages inter-clusters sont réduits comparé à l'algorithme plat de Naimi-Tréhel, en particulier pour les applications avec un faible degré de parallélisme.

L'ensemble de ces travaux partage donc l'idée de composer hiérarchiquement des algorithmes d'exclusion mutuelle. S'ils sont proches à beaucoup d'égards, chacun compose de façon "fixe" des algorithmes qui donneront à la composition ces propriétés. On peut alors penser que telle ou telle composition est plus adaptée à telle ou telle application. Cependant il n'existe pas d'étude exhaustive comparant ces compositions.

De plus, la majorité de ces travaux ne prend pas en compte la topologie physique réelle de la grille et l'hétérogénéité des latences. En effet, si ces travaux partagent la notion de groupe, ceux-ci ne sont pas toujours définis par la topologie physique et le regroupement des machines ne prend pas toujours la latence des communications comme principal critère. Les résultats des évaluations de ces travaux, lorsqu'ils sont présentés, sont basés sur une simulation ou sur une émulation de la plate-forme et non sur une grille réelle.

Notre approche s'inscrit en continuité de ces travaux. Elle reprend l'idée d'une composition hiérarchique et celle d'un nœud responsable dans chaque cluster. Mais plutôt que de proposer une nouvelle composition fixe, nous proposons une solution générique pour composer les algorithmes. Cet algorithme général permettra de composer l'ensemble des algorithmes d'exclusion mutuelle de la littérature. Il sera alors possible de choisir la meilleure composition en fonction non seulement de la topologie cible réelle (une grille) mais également du comportement de l'application. L'aspect générique de notre approche permet aussi de factoriser la preuve de ces compositions.

Enfin, notons que si des auteurs ont introduit artificiellement dans leur solution un mécanisme de préemption [BAS06, MK94] pour exploiter la localité entre les machines d'un cluster, celle-ci sera naturellement induite par notre algorithme de composition. Ceci évite toute intrusion dans les algorithmes composés.

6.2 Approche par composition pour l'exclusion mutuelle

6.2.1 Architecture hiérarchique

Notre approche consiste à composer de façon hiérarchique et générique des algorithmes d'exclusion mutuelle. Dans chaque cluster, les nœuds participent à une instance d'un algorithme d'exclusion mutuelle que l'on nommera par la suite algorithme *intra-cluster*. Ces algorithmes locaux constituent la couche basse de notre architecture et n'utilisent pour communiquer que des messages circulant sur les réseaux locaux des clusters. On appellera ces messages "messages *intra*".

Pour interconnecter les différentes instances d'algorithme d'exclusion mutuelle de cette couche basse, on ajoute une autre instance transversale répondant au nom d'algorithme *inter-cluster*. Ne participe à cet algorithme qu'un unique représentant de chacun des clusters. Ces sites communiquent entre eux au travers des liens longues distances (WAN) en échangeant des messages dits *inter*.

Le choix des algorithmes intra-cluster est libre et rien n'empêche d'instancier des algorithmes différents. Mais pour des raisons de lisibilité nous supposerons que tous les algorithmes *intra-cluster* sont des instances d'un même algorithme qui, par contre, pourra être différent de l'algorithme *inter-cluster*. Une composition se définira donc comme le choix des deux algorithmes *intra* et *inter*.

Pour la description et l'évaluation de notre approche, nous avons choisi de ne considérer que des compositions d'algorithmes d'exclusion mutuelle basés sur la circulation d'un jeton (voir la taxonomie section 2.5). Les raisons de ce choix sont les mêmes que dans le premier chapitre à savoir de bonnes propriétés en termes de passage à l'échelle. Mais ce choix ne constitue pas une restriction de notre algorithme de composition. Comme on le verra dans la preuve, notre mécanisme de composition fonctionne aussi bien avec des algorithmes d'exclusion mutuelle à permissions. On peut même envisager composer des algorithmes des deux familles.

Notre architecture hiérarchique distingue deux types de nœud, les nœuds *applicatifs* qui exécutent le code de l'application et qui ne participent qu'à l'algorithme *intra-cluster* (de leur cluster) et des nœuds *coordinateurs* (1 par cluster) qui participent à la fois à l'algorithme *intra-cluster* de leur cluster et à l'algorithme *inter-cluster*. Ces derniers, dont le rôle s'apparente à celui d'un proxy, sont les seuls à avoir une vision de la grille extérieure à leur cluster. Cette vision se limite néanmoins aux autres coordinateurs de la grille.

Notre architecture met en jeu de multiples instances d'algorithmes d'exclusion mutuelle et chacune d'elle utilise donc un jeton pour assurer l'unicité de l'accès à la section critique. Ainsi, chaque cluster possède un jeton local, le jeton *intra*, tandis que les coordinateurs s'échangent un unique jeton global, le jeton *inter*. Avec la multiplicité des jetons *intra*, plusieurs nœuds applicatifs sont susceptibles de posséder un jeton et donc de pouvoir accéder à la section critique. Assurer globalement la propriété de sûreté de l'exclusion mutuelle revient alors à garantir qu'à tout moment, un seul des nœuds applicatifs sur l'ensemble de la grille possède un jeton *intra*.

À cette propriété de sûreté doit s'ajouter un mécanisme assurant une *vivacité* globale : tout nœud applicatif de la grille voulant accéder à la section critique finira par recevoir le jeton *intra* de son cluster.

Le rôle des nœuds coordinateurs est d'assurer ces deux propriétés globales. Il se limite à celui d'une passerelle entre les algorithmes *intra-cluster* et les algorithmes *inter-cluster*. Ils ne font donc pas de requêtes d'entrée en section critique pour leur propre compte mais uniquement pour assurer une synchronisation dans la circulation des jetons. Cette synchronisation se fait en utilisant l'algorithme *inter*. Ainsi un algorithme de composition s'exécute sur chaque coordinateur pour s'assurer que seul le cluster, dont le coordinateur détient le jeton *inter*, possède un nœud applicatif en section critique. Cet algorithme de composition représente le seul lien entre les deux personnalités du coordinateur que constituent les algorithmes *intra-cluster* et *inter-cluster*. La synchronisation ne se fait qu'au travers des requêtes faites par les coordinateurs à telle ou telle instance.

À l'inverse, les nœuds applicatifs n'ont qu'une vision locale du système et ignorent tout du mécanisme de composition. Ils utilisent l'algorithme d'exclusion mutuelle *intra* au travers d'une API classique (présentée figure algorithme 2). Ainsi lorsque l'application désire accéder à la section critique, le nœud appelle la fonction *CS_Request()* de l'algorithme *intra* (ligne 123). Cette fonction est bloquante et ne se termine qu'au moment où le site applicatif peut exécuter son code concurrent (noté par la suite section critique *intra*). Au sortir de sa section critique, il utilisera la fonction *CS_Release()* (ligne 128) pour libérer la section critique locale.

La section suivante s'applique à décrire le fonctionnement de notre algorithme de composition.

6.2.2 Architecture générique de composition

Pour décrire notre algorithme de composition régissant le fonctionnement des coordinateurs, on se base sur l'automate de l'exclusion mutuelle (figure 6.2(a)) déjà introduit à la section 2.2.1. Ainsi on retrouve les trois états : REQ, CS, NO_REQ suivant que le site est respectivement en attente, exécutant, ou pas intéressé par la section critique.

Les *coordinateurs* exécutant deux instances d'algorithmes d'exclusion mutuelle (algorithmes *inter* et *intra*), on peut à tout moment définir leur comportement en fonction de leur état vis-à-vis de chacun de ces deux algorithmes. Le comportement des *coordinateurs* peut alors être traduit par un automate où chaque état se définit comme le tuple des états des instances *intra* et *inter* : état-*intra* / état-*inter*.

On introduit alors 4 états globaux correspondant à quatre tuples distincts :
- OUT : état *intra* $= CS$ et état *inter* $= NO\_REQ$
- $WAIT\_FOR\_IN$: état *intra* $= CS$ et état *inter* $= REQ$
- IN : état *intra* $= NO\_REQ$ et état *inter* $= CS$
- $WAIT\_FOR\_OUT$: état *intra* $= REQ$ et état *inter* $= CS$

Initialement, tous les coordinateurs sont dans l'état global OUT. Puisqu'ils possèdent alors tous le jeton *intra* de leur cluster respectif, on est assuré qu'il n'y a pas deux nœuds applicatifs en section critique à l'état initial. Notons que l'un des coordinateurs possède le jeton *inter* sans être dans l'état *inter* CS.

Ils se mettent alors à attendre la réception d'une requête *intra* d'un des nœuds applicatifs de leur cluster (algorithme 1, ligne 111).

Lorsqu'il reçoit une telle requête, un coordinateur passe dans l'état $WAIT\_FOR\_IN$. Pour relâcher son jeton *intra* et donc satisfaire la requête, le coordinateur doit récupérer

```
106 Coordinator Algorithm ()
107     /* Initially, it holds the intra-token */
108     while TRUE do
109         if ¬ intra.PendingRequest() then
110             state ← OUT
111             Wait for intra.PendingRequest()
112         state ← WAIT_FOR_IN
113         inter.CS_Request()
114         /* Holds inter-token. CS */
115         intra.CS_Release()
116         if ¬ inter.PendingRequest() then
117             state ← IN
118             Wait for inter.PendingRequest()
119         state ← WAIT_FOR_OUT
120         intra.CS_Request()
121         /* Holds intra-token CS */
122         inter.CS_Release()
```

Algorithm 1: Algorithme d'un coordinateur

```
123 CS_Request ()
124     ...
125     state ← REQ
126     Wait for Token
127     state ← CS

128 CS_Release ()
129     ...
130     state ← NO_REQ
131     if pendingRequest() then
132         Send Token

133 pendingRequest ()
        return { TRUE   if ∃ pending request
134            { FALSE  otherwise
```

Algorithm 2: Interface d'un algorithme d'exclusion mutuelle

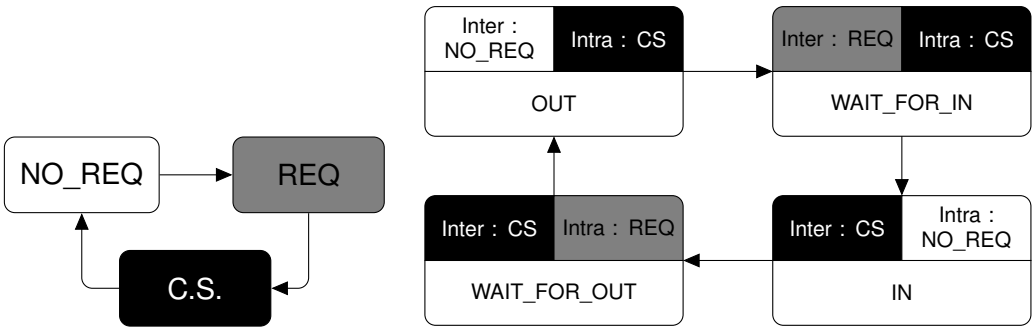
celui de l'algorithme *inter*. En effet, la possession du jeton *inter* lui garantit qu'aucun des nœuds applicatifs des autres clusters continue d'exécuter une section critique *intra*. Il émet donc (algorithme 1, lignes 113) une requête *inter*.

L'appel à la fonction bloquante *inter.CS_Request* se termine lorsque le coordinateur reçoit le jeton *inter*. Il peut dès lors libérer (algorithme 1, ligne 115) son jeton *intra* (état *intra* NO_REQ) tout en conservant celui de l'algorithme *inter* qu'il vient d'acquérir (état *inter* CS). Le coordinateur est alors dans l'état IN et y reste jusqu'a réception d'une requête *inter* (algorithme 1, ligne 117 et 118).

À la réception d'une requête *inter* le coordinateur passe de l'état IN à l'état $WAIT\_FOR\_OUT$. Il doit alors récupérer le jeton *intra* de son cluster pour s'assurer que plus aucun site applicatif n'exécute de section critique *intra*. Il émet donc une requête *intra* au travers de l'appel bloquant *intra.CS_Request* (algorithme 1, ligne 120).

Dés qu'il reçoit le jeton *intra*, le coordinateur utilise la fonction *inter.CS_Release* (algorithme 1, ligne 122) pour relâcher le jeton *inter*. S'il n'y a plus de requête en attente dans son cluster, le coordinateur retourne à l'état (OUT) et attend la prochaine requête *intra*. Dans le cas contraire il émet directement une requête dans l'algorithme *inter* pour en récupérer le jeton (algorithme 1, ligne 113).

L'algorithme des coordinateurs se résume donc à une boucle dans laquelle ils vont successivement : récupérer le jeton *inter*, relâcher l'*intra*, le récupérer pour pouvoir relâcher le jeton *inter*, etc. On peut l'expliciter facilement au travers de l'automate à 4 états présenté dans la figure 6.2(b).



(a) Automate classique (b) Automate des coordinateurs

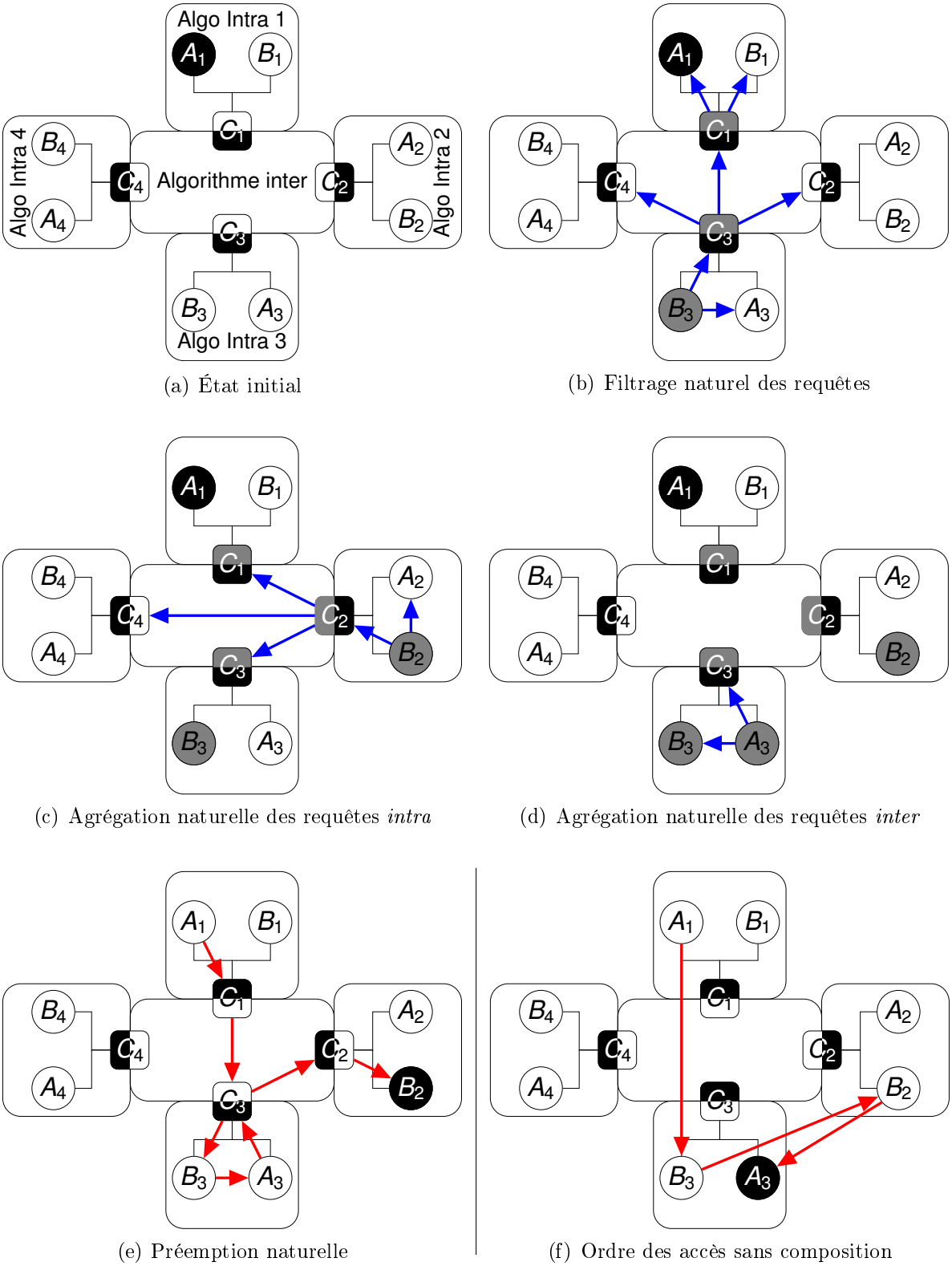
FIGURE 6.2 – Automates d'exclusion mutuelle

Assurer la propriété de sûreté revient alors à maintenir l'invariant suivant :

Invariant 7. *A tout moment il n'y a au plus un seul coordinateur dans l'état IN ou dans l'état $WAIT\_FOR\_OUT$.*

6.3 Propriétés de la composition

Pour mieux comprendre comment notre algorithme de composition limite la complexité en nombre de messages et améliore les temps d'accès à la section critique, intéressons nous



(a) État initial

(b) Filtrage naturel des requêtes

(c) Agrégation naturelle des requêtes *intra*

(d) Agrégation naturelle des requêtes *inter*

(e) Préemption naturelle

(f) Ordre des accès sans composition

FIGURE 6.3 – Effets de la composition

à l'exemple suivant (figures 6.3). On considère ici une grille composée de quatre clusters eux mêmes composés de 3 nœuds. Ainsi, dans chaque cluster i on aura 2 nœuds applicatifs A_i et B_i auxquels s'ajoutera 1 nœud coordinateur C_i. Pour simplifier cet exemple et ainsi faciliter l'étude des mécanismes mis en jeu par la composition, on choisit ici d'instancier sur les deux niveaux un algorithme simple basé sur la diffusion. Comme dans l'algorithme de Ichiro Suzuki et Tadao Kasami [SK85], un site voulant entrer en section critique diffuse une requête à tous les participants du système et lorsqu'un site quitte sa section critique, il envoie le jeton au premier nœud lui ayant demandé le jeton.

Comme on peut le voir sur la figure 6.3(a) le nœud A_1 possède, à l'état initial, le jeton *intra* du cluster 1 et son coordinateur celui de l'algorithme *inter*. Tous les autres jetons *intra* sont détenus par les autres coordinateurs (C_2, C_3, C_4).

6.3.1 Filtrage naturel des requêtes

Dans un premier temps, le nœud B_3 veut entrer en section critique. Il diffuse sa requête *intra* dans son cluster (figure 6.3(b)). Elle est ensuite retransmise dans l'algorithme *inter* par son coordinateur C_3 qui passe dans l'état $WAIT\_FOR\_IN$. Seul le coordinateur C_1 rediffuse la requête dans son cluster. On peut voir ici un premier gain en termes de nombre de messages : non seulement on se limite à 3 messages *inter* du fait de la diffusion hiérarchique, mais on évite aussi de rediffuser la requête dans les clusters 3 et 4. En effet, comme les coordinateurs C_2 et C_4 sont dans l'état OUT, aucun des nœuds applicatifs de leur cluster ne peut être en section critique. Il en résulte un *filtre naturel* dans la rediffusion des requêtes.

6.3.2 Agrégation naturelle des requêtes *intra*

Après le nœud B_3, le nœud B_2 désire à son tour entrer en section critique et diffuse une requête *intra* dans le cluster 2. Comme le montre la figure 6.3(c), le mécanisme de *filtre* permet là encore d'économiser la rediffusion dans les clusters dont les nœuds n'exécutent pas la section critique. Mais ici, la composition évite aussi de diffuser une requête dans le cluster 1 dont le coordinateur est déjà dans l'état $WAIT\_FOR\_OUT$ et donc en attente de son jeton *intra*. La requête *intra* émise par C_1 pour satisfaire la requête du site B_3, servira aussi à satisfaire celle du site B_2. On observe donc un mécanisme d'*agrégation naturelle* des messages *intra*.

6.3.3 Agrégation naturelle des requêtes *inter*

Enfin, si une deuxième requête a lieu dans le cluster 3 (émise par le site A_3), on peut observer sur la figure 6.3(d) qu'il n'y a plus de rediffusion au niveau *inter*. En effet, le coordinateur C_3 qui est déjà dans l'état $WAIT\_FOR\_IN$ n'a pas besoin de renouveler sa requête au niveau *inter*. Comme il le fait pour les messages *intra*, l'algorithme de composition agrège donc naturellement les requêtes locales limitant ainsi le nombre de messages *inter*.

6.3.4 Préemption naturelle

On a maintenant trois requêtes concurrentes dans la grille. La figure 6.3(e) montre comment la circulation des différents jetons permet de les satisfaire. Ainsi, au sortir de la section critique du nœud A_1, C_1 récupère le jeton *intra* du cluster 1 et transmet le jeton *inter* au coordinateur C_3. Ce dernier peut alors envoyer le jeton *intra* du cluster 3 au nœud B_3 passant ainsi de l'état $WAIT\_FOR\_IN$ à l'état IN. Mais dans le même temps, du fait de la requête pendante du nœud B_2, il diffuse une requête *intra* et passe dans l'état $WAIT\_FOR\_OUT$. Donc lorsque le nœud B_3 termine d'exécuter son code concurrent, il reste deux requêtes pendantes dans le cluster 3 : celle du nœud A_3 et celle du coordinateur C_3. L'ordre d'accès au jeton dans une même instance d'algorithme respectant celui des émissions des requêtes, B_3 envoie le jeton au nœud A_3. Ce n'est finalement qu'une fois la section critique de A_3 terminée que le coordinateur C_3 récupérera le jeton *intra*. Il envoie alors le jeton *inter* au coordinateur C_2 qui peut alors libérer le jeton *intra* du cluster 2, permettant ainsi au site B_2 d'accéder à la section critique.

Pour comprendre à partir de cet exemple comment notre mécanisme de composition permet de réduire le temps d'attente moyen de la section critique, il nous faut considérer le traitement de ces trois requêtes pendantes par un algorithme d'exclusion mutuelle global *à plat* (figure 6.3(f)).

Si cet algorithme est équitable (au sens équité forte), il satisfera ces dernières dans leur ordre d'émission. Après le nœud A_1, les nœuds B_3, B_2 et A_3 accéderont successivement à la section critique, comme le montre la figure 6.3(f). Ainsi le jeton passera du cluster 1 au cluster 3, puis du cluster 3 au cluster 2, pour finir par revenir dans le cluster 3. Ces allers-retours entre les clusters 2 et 3 augmentent considérablement la latence d'accès à la section critique. En effet, contrairement au temps de propagation des requêtes qui peut être partiellement (parfois même totalement) recouvert par des exécutions de sections critiques, pendant les transmissions de jeton aucun nœud ne peut accéder à la section critique. Minimiser le coût de ces transmissions a donc un impact direct sur la latence.

Dans le cas de notre algorithme de composition on peut observer sur la figure 6.3(e) un réordonnancement de l'accès à la section critique. Ce nouvel ordre d'accès à la section critique est induit naturellement par l'algorithme de composition qui favorise les accès locaux (dans un même cluster). Ce mécanisme permettant d'optimiser l'ordre d'accès à la section critique effectue donc un rôle d'ordonnanceur. Reprenant une dénomination système, nous y feront référence par la suite sous le nom de *préemption naturelle*. En effet, tout se passe comme si les requêtes locales préemptaient l'accès à la section critique.

6.3.5 Effets de la composition

Au travers de l'exemple présenté dans la section précédente, nous avons pu observer la mise en œuvre de plusieurs effets de la composition. Ces effets ne résultent pas de l'ajout de mécanismes spécifiques, mais résulte de la structure hiérarchique et de l'utilisation de coordinateurs. C'est pour cette raison qu'ils sont qualifiés de *naturels*. Cette dénomination permet, à mon sens, de mieux les distinguer des effets dus à des mécanismes complémentaires comme ceux présentés dans la section 7.5.

Certains de ces effets *naturels* apportent un gain en terme de nombre moyen de mes-

sages nécessaires à l'obtention d'une section critique :

- **filtre naturel :** les requêtes ne sont diffusées que dans le cluster où la section critique s'exécute (ou vient de s'exécuter).
- **agrégation naturelle** des requêtes *inter* cluster : il n'y a au plus une rediffusion au niveau *inter* pour l'ensemble des requêtes concurrentes d'un cluster.
- **agrégation naturelle** des requêtes *intra* cluster : dans le cluster où la section critique s'exécute, il n'y a au plus une émission de requête *intra* pour toutes les requêtes concurrentes émises dans les autres clusters.

De plus, le temps d'attente moyen d'une section critique pour un nœud applicatif se trouve réduit par :

- **préemption naturelle** : les requêtes pendantes d'un cluster lorsque le coordinateur reçoit le jeton *inter* sont toutes satisfaites avant que celui-ci ne le relâche.
- **effet structurel** : Les algorithmes qui utilisent une structure logique pour acheminer les requêtes et/ou le jeton, présentent un délai moyen d'obtention proportionnel au nombre de sites du système. Or la structure de la grille réduit naturellement le nombre de participants aux différentes instances des algorithmes. Le temps d'attente moyen s'en trouve réduit. Remarquons que cet effet structurel n'apparait pas sur l'exemple qui utilise un mécanisme de diffusion.

L'ensemble des effets présentés ici améliorent les performances de l'algorithme, mais il nous faut aussi considérer le surcoût de la composition. En effet, les coordinateurs ne sont pas présents dans le système de base. Les messages qu'ils génèrent ainsi que leur temps de traitement sont donc autant de surcharge pour l'exclusion mutuelle.

Nous présentons dans la section 7.2 une étude comparative entre l'approche classique (dite *à plat*) et notre approche hiérarchique pour mesurer les gains réels apportés par ces effets.

6.4 Compositions valides

Nous avons composé, Dans l'exemple précédent, une version simplifiée de l'algorithme proposé par Suzuki et Kasami [SK85]. Notre mécanisme de composition s'y est montré peu intrusif envers les algorithmes composés : les algorithmes composés ne sont pas modifiés. Est-il pour autant capable d'assembler n'importe quel algorithme d'exclusion mutuelle ? Cette section et la suivante tentent de répondre à cette question en listant les propriétés nécessaires à la bonne marche de la composition.

Notons tout d'abord que les propriétés d'une composition reposent entièrement sur les propriétés offertes par les algorithmes composés. Il est ainsi nécessaire de ne composer que des algorithmes permettant de résoudre localement l'exclusion mutuelle, c'est-à-dire vérifiant toutes les propriétés de sûreté et de vivacité de ce paradigme (voir section 2.2.2). À cette unique condition, toute composition vérifiera globalement ces propriétés de sûreté et de vivacité, réalisant donc une exclusion mutuelle pour l'ensemble du système. La preuve de cette assertion sera donnée dans la section 6.6.

Il est à noter qu'il n'est fait aucune hypothèse sur le type (famille, sous famille,...) des algorithmes ainsi composés. Algorithme à permissions et algorithme à jeton vérifiant tous deux ces mêmes propriétés, il est tout à fait possible de composer un (ou plusieurs)

algorithmes de chacune de ces deux familles. Mais puisque nous avons choisi dans cette thèse de s'intéresser aux algorithmes à jeton (voir section 2.6), nous ne parlerons dans la suite de ce document que des compositions de cette famille.

Il n'y a donc pas de restriction sur le choix de la composition pour obtenir une composition valide. Toutefois, même si elles demeurent valide algorithmiquement, toutes les compositions n'en sont pas pour le moins efficaces.

6.5 Compositions efficaces

Certaines caractéristiques des algorithmes composés peuvent rendre inefficace, voir pénaliser, notre mécanisme de composition. Les sous-sections suivantes s'intéressent à décrire ces caractéristiques afin d'identifier les classes d'algorithmes rendant inefficace la composition. Parmi les autres algorithmes restant, nous étudierons de façon approfondie à la section 7.3 les compositions optimales suivant le type d'applications.

Nous étudierons tour à tour les propriétés de :

quiescence : l'algorithme doit pouvoir s'arrêter lorsqu'il n'y a plus de requête en cours. Il est alors dans un état quiescent.

équité forte : l'algorithme doit respecter l'ordre d'émission des requêtes.

6.5.1 Composition quiescente

Comme nous l'avons vu à la section 6.2.1, les coordinateurs ne font pas de demande pour leur propre compte mais uniquement pour permettre la circulation des jetons. Dans ces conditions, si aucun des nœuds applicatifs n'exécute de code concurrent (i.e. tous sont dans l'état NO_REQ), les jetons ne devraient pas être amenés à circuler. Par conséquent, les coordinateurs ne devraient pas avoir à émettre de requêtes. La grille contient alors un unique coordinateur dans l'état IN, tous les autres étant dans l'état OUT. Cette situation doit perdurer tant qu'il n'y a pas de requête applicative. L'algorithme est alors au repos : il n'y a plus d'envoi de message. On dit d'un algorithme pouvant atteindre cet état qu'il est *quiescent*.

Ce mécanisme de "mise en veille" de l'algorithme permet d'éviter un grand nombre de messages inutiles. Même si cela ne remet pas en cause les propriétés de sûreté et de vivacité, il est donc important de ne composer que des algorithmes d'exclusion mutuelle permettant d'obtenir une composition *quiescente*.

Il faut tout d'abord exclure des compositions les algorithmes n'accédant pas à un état quiescent. Parmi ceux-ci on peut citer l'algorithme de Gérard Le Lann [Lan78]. En effet, dans ce dernier, le jeton circule en permanence autour de l'anneau virtuel ne s'arrêtant dans un site que pour la seule durée d'exécution du code concurrent. Quels que soient les autres algorithme composés, si deux clusters utilisent cet algorithme comme algorithme *intra*, leur coordinateurs ne pourront faire autrement que d'échanger le jeton *inter* à chaque tour d'anneau *intra*. Le mécanisme de composition devient alors extrêmement coûteux en nombre de messages envoyés.

Mais peut-on pour autant composer n'importe quel algorithme pouvant être quiescent ? Malheureusement, contrairement aux propriétés de sûreté et de vivacité, il ne suffit pas de composer des algorithmes *quiescent* pour que la composition la vérifie à son tour.

Considérons l'algorithme reposant sur un opencube dynamique proposé par Jean-Michel Helary et Achour Mostefaoui [HM94]. Dans ce dernier, certains sites sont amenés à rendre le jeton lorsqu'ils finissent d'exécuter leur section critique. Ainsi, lorsqu'il n'y a plus de requête, l'algorithme ne devient réellement quiescent qu'à l'instant où la racine de l'opencube récupère le jeton. Imaginons alors une composition utilisant cet algorithme comme algorithme *intra* dans deux clusters distincts (A et B). Si les coordinateurs de ces deux clusters font partie des sites devant rendre le jeton *intra* à la racine de leur opencube, la composition n'est pas quiescente. En effet, ces 2 coordinateurs ne peuvent rendre leur jeton *intra* qu'à la condition de détenir le jeton *inter*. En l'absence d'autre requête, on assiste alors au ballet suivant : racine A, coordinateur A, coordinateur B, racine B et inversement.

Au vu de cet exemple, on s'aperçoit qu'une composition ne pourra accéder à un état de "veille" qu'à la condition suivante : tous les coordinateurs doivent conserver leurs jetons (*intra* et/ou *inter*) en l'absence de requêtes applicatives.

On devra donc se limiter, pour le choix des algorithmes *intra* et l'algorithme *inter*, aux algorithmes d'exclusion mutuelle qui, en absence de requête applicative, se mettent immédiatement en état de "veille". Ces algorithmes vérifient une propriété plus forte que la quiescence, que nous appellerons :

quiescence immédiate : *tout site conserve le jeton dès lors qu'il n'y a plus de requête pendante.*

6.5.2 Équité forte

L'exemple présenté à la section 6.3 montre comment notre mécanisme de composition induit une préemption naturelle et comment celle-ci permet de diminuer fortement le délai moyen d'obtention de la section critique. Puisqu'elle représente l'un des points forts de notre proposition, il est important d'analyser l'impact du choix des algorithmes sur cette préemption naturelle.

Dans cet exemple, la préemption était caractérisée par l'envoi du jeton par le site B_3 au site A_3 alors que dans le cluster 2 le site B_2 avait émis une requête avant ce dernier. Ainsi, la préemption naturelle nécessite de composer des algorithmes d'exclusion mutuelle dont l'ordre de traitement des requêtes permet d'assurer la propriété suivante : toutes les requêtes pendantes dans un cluster, à l'instant où le coordinateur libère le jeton *intra*, seront satisfaites avant que celui-ci ne libère le jeton *inter*.

Ce comportement peut être obtenu simplement par la composition d'algorithme vérifiant la propriété d'*équité forte* (introduite à la section 2.2.3) de l'algorithme composé : les requêtes étant alors traitées par ordre d'émission.

Autrement dit, on choisira de composer des algorithmes mettant en jeux une "file d'attente". Cette dernière pourra être topologique comme dans l'algorithme de Le Lann ([Lan78]), où les traitements des requêtes suivent la topologie logique en anneau, mais aussi locale comme dans l'algorithme de Suzuki [SK85], dans lequel chacun des participants

maintient une vue des requêtes en attente, ou encore complètement distribuée comme dans les algorithmes de Naimi-Tréhel [NT96] ou de Raymond [Ray89].

Pour terminer, remarquons que contrairement au critère des sections 6.5 (vivacité et sûreté) et 6.5.1 (quiescence), celui de l'équité ne s'applique qu'au seul choix des algorithmes *intra*.

6.6 Preuve

Cette section présente une preuve de notre algorithme de composition. Prouver un tel algorithme revient à prouver une transitivité des propriétés locales vers les propriétés globales offertes par la composition.

Comme expliqué dans la section 6.5, on supposera ici que tous les algorithmes composés vérifient les propriétés de sûreté et de vivacité de l'exclusion mutuelle définies à la section 2.2.2. Parmi les différentes propriétés de vivacité on s'intéressera à celle dite "d'équité faible" introduite à la section 2.2.3.

6.6.1 Notations et formalismes

Pour formaliser cette preuve nous utiliserons les notations suivantes :

$\mathcal{C}$ **:** ensemble des identifiants des clusters.

$\bar{\mathcal{C}}_c = \{c' \in \mathcal{C} | c' \neq c\}$ **:** ensemble des identifiants des clusters de la grille, privé de l'identifiant du cluster c.

$\mathcal{A}_c$ **:** ensemble des identifiants des nœuds du cluster c. Dans chacun des clusters, les identifiants sont distincts et le coordinateur est noté par l'identifiant 0.

$\mathcal{A}_c^* = \{i \in \mathcal{A}_c | i \neq 0\}$ **:** ensemble des identifiants des nœuds applicatifs du cluster c.

$\mathcal{S} =$ **{NO_REQ, REQ, CS }** **:** ensemble des états accessibles par un nœud vis-à-vis d'un algorithme d'exclusion mutuelle.

$\mathcal{T} = [t_0, t_1, t_2, ...]$ **:** séquence discrète infinie. Chacun de ces termes, représente une date du système.

L'accès aux états des différents nœuds de la grille se fera au travers des fonctions temporisées suivantes :

$CoordState_{inter}$: $\mathcal{C} \times \mathcal{T} \rightarrow \mathcal{S}$. $CoordState_{inter}(c,t)$ retourne l'état du coordinateur du cluster c vis-à-vis de l'algorithme *inter* à l'instant t.

$CoordState_{intra}$: $\mathcal{C} \times \mathcal{T} \rightarrow \mathcal{S}$. $CoordState_{intra}(c,t)$ retourne l'état du coordinateur du cluster c vis-à-vis de l'instance de l'algorithme *intra* de ce cluster à l'instant t.

$Instruction$: $\mathcal{C} \times \mathcal{T} \rightarrow [108..122]$. $Instruction(c,t)$ retourne le numéro de la ligne du pseudo-code de l'algorithme 1 contenant l'instruction que le coordinateur du cluster c exécute à l'instant t.

$State_c$: $\mathcal{A}_c^* \times \mathcal{T} \rightarrow \mathcal{S}$. $State_c(a,t)$ retourne l'état du nœud applicatif a du cluster c vis à vis de l'instance de l'algorithme *intra* de ce cluster à l'instant t.

$PendReq_{inter}$: $\mathcal{C} \times \mathcal{T} \rightarrow Boolean$. $PendReq_{inter}(c,t)$ retourne TRUE si le coordinateur du cluster c a enregistré une requête non traitée dans l'algorithme *inter* à l'instant t et FALSE sinon.

$PendReq_{intra}$: $\mathcal{C} \times \mathcal{T} \rightarrow Boolean$. $PendReq_{intra}(c,t)$ retourne TRUE si le coordinateur du cluster c a enregistré une requête non traitée dans l'instance de l'algorithme *intra* de ce cluster à l'instant t et FALSE sinon.

6.6.2 Hypothèses

La preuve présentée ici suppose que les algorithmes composés respectent tous la propriété de quiescence immédiate présentée à la section 6.5.1. Dans le cas contraire la preuve de la vivacité se simplifie puisque les coordinateurs relâchent naturellement le jeton.

Avec cette hypothèse, les coordinateurs peuvent être amenés à conserver le jeton jusqu'à ce qu'une requête soit émise. Nous supposerons donc, pour les 2 algorithmes composés (*intra* et *inter*), qu'un coordinateur en état CS finit toujours par détecter la présence d'une requête pendante. Formellement on introduit donc les deux hypothèses suivantes :

Hypothèse 1.

$$\forall c \in \mathcal{C}, \forall a \in \mathcal{A}_c^*, \forall t \in \mathcal{T},$$
$$\left.\begin{array}{r} CoordState_{intra}(c,t) = CS \\ \wedge \;\; State_c(a,t) = REQ \end{array}\right\} \Longrightarrow \exists t' > t \in \mathcal{T}, PendReq_{intra}(c,t') = \text{TRUE}$$

Hypothèse 2.

$$\forall c \in \mathcal{C}, \forall c' \in \bar{\mathcal{C}}_c, \forall t \in \mathcal{T},$$
$$\left.\begin{array}{r} CoordState_{inter}(c,t) = CS \\ \wedge \;\; CoordState_{inter}(c',t) = REQ \end{array}\right\} \Longrightarrow \exists t' > t \in \mathcal{T}, PendReq_{inter}(c',t') = \text{TRUE}$$

Il nous faut aussi ici formaliser les hypothèses d'initialisation. À l'état initial, tous les coordinateurs accèdent à la section critique *intra* de leur cluster. On a donc l'hypothèse suivante.

Hypothèse 3. $\forall c \in \mathcal{C}, CoordState_{intra}(c,t_0) = CS$

6.6.3 Preuve de la propriété de sûreté

Dans la section 2.2.2 nous avions défini la propriété de sûreté de l'exclusion mutuelle. Traduite en termes de composition elle s'exprime de la manière suivante : *à tout moment il n'y a au plus qu'un site applicatif dans l'état CS*. Autrement dit :

Sûreté : $\forall t \in \mathcal{T}, \forall c \in \mathcal{C}, \forall a \in \mathcal{A}_c^*,$

$$State_c(a,t) = CS \Longrightarrow (\forall c' \in \mathcal{C}, \forall a' \in \mathcal{A}_{c'}^*, State_{c'}(a',t) = CS \Longrightarrow (c = c' \wedge a = a'))$$

D'autre part, dans la section 6.2.2 présentant notre algorithme de composition, nous avons vu que son fonctionnement repose sur le fait qu'au plus un coordinateur peut se trouver dans l'état IN ou $WAIT\_FOR\_OUT$ (Invariant 7). Avec les notations introduites pour cette preuve, nous pouvons formaliser cet invariant de la manière suivante :

Invariant 8. $\forall c \in \mathcal{C}, \forall t \in \mathcal{T}$,

$$CoordState_{intra}(c,t) \neq CS \Longrightarrow \forall c' \in \bar{\mathcal{C}}_c, CoordState_{intra}(c',t) = CS$$

Plan de la preuve : La preuve de la sûreté s'articule autour de cet invariant. On procède en deux temps :

1. Pour commencer nous montrons que l'invariant 8 est une condition suffisante à la propriété de sûreté (lemme 8).
2. Puis nous montrons que, quel que soit l'évolution du système, toutes les compositions vérifient cet invariant (lemme 11).

Lemme 8. *Si une composition d'algorithme d'exclusion mutuelle vérifie l'invariant 8 alors elle vérifie la propriété de sûreté de l'exclusion mutuelle.*

Démonstration. Passons à la contraposée et montrons que si la composition ne vérifie pas la propriété de sûreté de l'exclusion mutuelle elle ne respecte pas l'invariant 8.

Supposons qu'à l'instant $t \in \mathcal{T}$, il existe deux clusters $c_1 \in \mathcal{C}$ (resp. $c_2 \in \mathcal{C}$) tel qu'un de leurs nœuds applicatifs $a_1 \in \mathcal{A}^*_{c_1}$ (resp. $a_2 \in \mathcal{A}^*_{c_2}$) soit en section critique. Dans le cluster c_1 (resp. c_2), la propriété de sûreté de l'algorithme *intra* implique que le coordinateur de c_1 (resp. c_2) n'est pas en section critique.

Puisque $CoordState_{intra}(c_1,t) \neq CS$ et $CoordState_{intra}(c_2,t) \neq CS$ l'invariant 8 n'est pas respecté. □

Lemme 9. $\forall c \in \mathcal{C}, \forall t \in \mathcal{T}, CoordState_{intra}(c,t) \neq CS \Longrightarrow CoordState_{inter}(c,t) = CS$

Démonstration. Soit $t \in \mathcal{T}$ et un cluster $c \in \mathcal{C}$ tel que $CoordState_{intra}(c,t) \neq CS$. Puisqu'à l'état initial on avait $CoordState_{intra}(c,0) = CS$ (hypothèse 3), ce coordinateur a dû exécuter la ligne 115 du pseudo-code de la figure 1. Cette instruction est en effet le seul appel à la fonction $intra.CS\_Release()$ dans l'algorithme du coordinateur. D'autre part, il n'a pas pu exécuter (terminer l'appel bloquant) l'instruction $intra.CS\_Request()$ de la ligne 120. On a alors :

$$CoordState_{intra}(c,t) \neq CS \Longrightarrow 115 < Instruction(c,t) \leqslant 120$$

Puisque l'exécution de ce code est séquentielle, ce coordinateur a exécuté à l'instant $t' < t$ l'instruction $inter.CS\_Request()$ de la ligne 113, et puisqu'il n'y a pas d'instruction $inter.CS\_Release()$ entre les lignes 115 et 120, on peut conclure que $CoordState_{inter}(c,t) = CS$ □

Lemme 10. $\forall c \in \mathcal{C}, \forall t \in \mathcal{T}, CoordState_{inter}(c,t) \neq CS \Longrightarrow CoordState_{intra}(c,t) = CS$

Démonstration. Il s'agit de la contraposée du lemme 9. □

Lemme 11. *À tout moment, l'état d'une composition d'algorithme d'exclusion mutuelle vérifie l'invariant 8.*

Démonstration. Soit $t \in \mathcal{T}$ et un cluster $c \in \mathcal{C}$ on a :

$$\begin{aligned} CoordState_{intra}(c,t) \neq CS \quad &\underset{Lemme\ 9}{\Longrightarrow} \quad CoordState_{inter}(c,t) = CS \\ &\underset{\text{Sureté } inter}{\Longrightarrow} \quad \forall c' \in \bar{\mathcal{C}}_c, CoordState_{inter}(c',t) \neq CS \\ &\underset{Lemme\ 10}{\Longrightarrow} \quad \forall c' \in \bar{\mathcal{C}}_c, CoordState_{intra}(c',t) = CS \end{aligned}$$

□

Théorème 3. *Une composition d'algorithmes d'exclusion mutuelle vérifie la propriété de sûreté de l'exclusion mutuelle.*

Démonstration. Une composition d'algorithmes d'exclusion mutuelle vérifie la propriété de sûreté car elle vérifie toujours l'invariant 8 (lemme 11) qui est une condition suffisante à la sûreté (lemme 8). □

6.6.4 Preuve de la vivacité : équité faible

Nous allons maintenant prouver la propriété d'équité faible. Cette propriété de vivacité à été introduite dans la section 2.2.2. Traduite en termes de composition elle s'exprime de la manière suivante : *Si tous les accès applicatifs aux sections critiques intra sont finis, alors tout site applicatif qui le désire finira par accéder à la section critique intra de son cluster.* Formellement :

Équité faible : Si $\forall c \in \mathcal{C}, \forall a \in \mathcal{A}_c^*, \forall t \in \mathcal{T}, \exists t' > t \in \mathcal{T}, State_c(a,t') = NO\_REQ,$
$\forall t'' \in \mathcal{T}, \forall c' \in \mathcal{C}, \forall a' \in \mathcal{A}_c^*, State_c(a',t'') = REQ \Longrightarrow \exists t''' > t'' \in \mathcal{T}, State_c(a',t''') = CS$

Plan de la preuve : l'idée de la preuve de l'équité faible est d'utiliser cette même propriété dans les algorithmes composés. Pour ce faire, il faut prouver que les différents accès (*inter* comme *intra*) des coordinateurs à la section critique sont finis. On procédera en deux temps :

1. Pour commencer nous montrons que si les durées des sections applicatives sont finies, un coordinateur qui le désire finit toujours par accéder à la section critique *inter* (lemme 12).
2. Puis nous montrons qu'en présence de requêtes applicatives dans un cluster les accès de son coordinateur à la section critique *intra* sont finis (lemme 13).

Lemme 12. *Si les durées d'exécution des sections critiques des nœuds applicatifs sont finies, un coordinateur qui le désire finira toujours par accéder à la section critique inter.*

Démonstration. La propriété d'équité faible de l'algorithme *inter* garantit à un coordinateur l'accès à la section *inter* si et seulement si les durées d'exécution des sections critiques sont finies.

Seuls des coordinateurs participent à l'algorithme *inter*. Les durées des sections critiques sont donc définies par l'algorithme des coordinateurs. Puisque les coordinateurs ne possèdent pas la section critique *inter* à l'état initial, que le seul appel $inter.CS\_Request()$ se trouve ligne 113, que le seul appel $inter.CS\_Release()$ se trouve ligne 122 et que l'exécution est séquentielle, on peut dire que :

$$\forall c \in \mathcal{C}, \forall t \in \mathcal{T}, CoordState_{inter}(c,t) = CS \Longrightarrow 113 < Instruction(c,t) \leqslant 122$$

Or, ces lignes ne comportent que deux appels bloquants :
- Wait for $inter.PendingRequest()$ (ligne 118).
- $intra.CS\_Request()$ (ligne 120).

Tant qu'il y a des requêtes *inter* pendantes, l'hypothèse 2 garantit que le premier appel n'est pas bloquant. Pour le deuxième, il faut considérer la propriété d'équité faible de l'algorithme *intra* du coordinateur bloqué. Cette dernière nous assure que si les nœuds applicatifs de ce cluster ne restent pas un temps infini en section critique (hypothèse du lemme) alors le coordinateur sortira bien de l'appel bloquant $intra.CS\_Request()$. On peut donc conclure qu'avec les hypothèses du lemme :

$$\forall c \in \mathcal{C}, \forall c' \in \bar{\mathcal{C}_c}, \forall t \in \mathcal{T},$$
$$\left.\begin{array}{r} CoordState_{inter}(c,t) = REQ \\ \wedge\ CoordState_{inter}(c',t) = CS \end{array}\right\} \Longrightarrow \exists t' > t \in \mathcal{T}, CoordState_{inter}(c',t') \neq CS$$

□

Lemme 13. *En présence de requête(s) intra dans son cluster, un coordinateur ne peut rester indéfiniment en section critique intra.*

Démonstration. Dans la démonstration du lemme 9, on a vu que pour qu'un coordinateur ne soit pas en section critique *intra* il devait être en train d'exécuter une ligne l comprise entre $115 < l \leqslant 120$. Or, ces lignes appartiennent à une boucle séquentielle dont le reste des instructions ne comprend que deux appels bloquants :
- Wait for $intra.PendingRequest()$ (ligne 111).
- $inter.CS\_Request()$ (ligne 113).

Puisque nous nous intéressons au cas où il y a une requête *intra* pendante, l'hypothèse 1 nous garantit que le premier n'est pas bloquant. De plus, comme nous supposons les durées d'exécution des sections critiques des nœuds applicatifs finies, le lemme 12 nous garantit que l'appel $inter.CS\_Request()$ sera fini. On peut donc conclure :

$$\forall c \in \mathcal{C}, \forall a \in \mathcal{A}_c*, \forall t \in \mathcal{T},$$
$$\left.\begin{array}{r} State_c(a,t) = REQ \\ \wedge\ CoordState_{intra}(c,t) = CS \end{array}\right\} \Longrightarrow \exists t' > t \in \mathcal{T}, CoordState_{intra}(c,t') \neq CS$$

□

Théorème 4. *Une composition d'algorithmes d'exclusion mutuelle vérifiant la propriété d'équité faible vérifie elle aussi cette propriété.*

Démonstration. Avant de commencer la preuve, nous posons l'hypothèse que tous les accès à la section critique des nœuds applicatifs sont finis. En effet, dans le cas contraire le théorème est trivialement vérifié.

Supposons qu'un nœud applicatif a du cluster c désire accéder à la section critique. La propriété de l'équité faible de l'algorithme *intra* lui garantit cet accès si et seulement si les accès des autres participants sont finis. Or tant que ce nœud n'a pas accédé à la section critique, il y a au moins une requête pendante. Le lemme 13 nous assure alors que le coordinateur ne peut rester indéfiniment en section critique. De plus on a supposé les sections critiques applicatives finies. Le nœud a est donc certain de finir par accéder à la section critique.

□

6.6.5 Preuve de l'algorithme

Théorème 5. *Une composition d'algorithmes d'exclusion mutuelle est un algorithme d'exclusion mutuelle.*

Démonstration. Un algorithme résout l'exclusion mutuelle si et seulement s'il vérifie ses propriétés de sûreté et de vivacité. Puisque l'équité faible est l'une des propriétés de vivacité de l'exclusion mutuelle, les théorèmes 3 et 4 suffisent à prouver le théorème. □

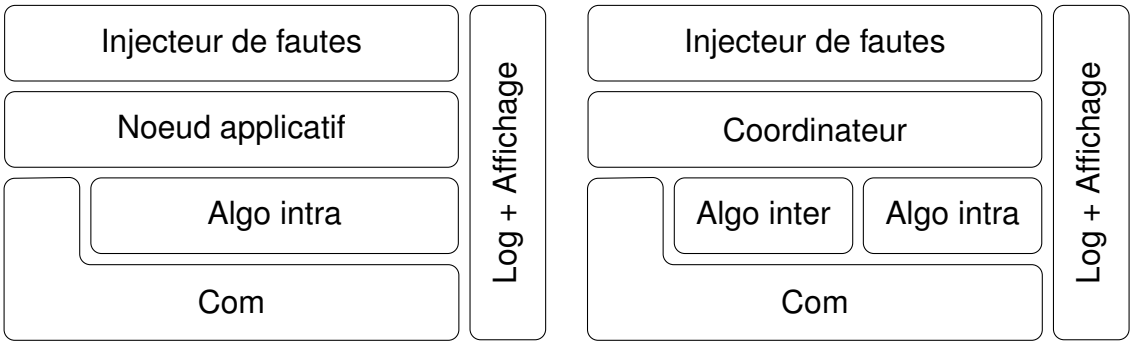
Chapitre 7

Évaluation de performance de notre approche générique

Sommaire

7.1 Implémentation dans la plate-forme de test

Dans le but d'en faire l'évaluation, nous avons implémenté notre algorithme de composition sur la plate-forme *G-Mutex* présenté dans le chapitre 5. Il s'agit aussi ici de vérifier l'aspect générique de notre approche ainsi que d'en mesurer l'intrusion dans les algorithmes déjà implémentés.



(a) Nœuds applicatifs (b) Coordinateurs

FIGURE 7.1 – Intégration dans l'architecture de la plate-forme de test *G-Mutex*

7.1.1 Implémentation des nœuds applicatifs

La figure 7.1(a) présente l'architecture utilisée pour implémenter la partie applicative de la composition. Cette architecture reprend composant par composant l'architecture présentée à la section 5.1 dans la figure 5.1.

L'algorithme d'exclusion d'exclusion mutuelle globale fait place ici à l'algorithme *intra*, mais vu de l'application, il s'agit d'un composant MUTEX comme les autres.

Pour conserver intact le code de cet algorithme et lui cacher la notion de groupe, on le connecte à librairie de communication au travers de l'interface requise (fonctions d'envoi, de réception et de diffusion). Ainsi, à son initialisation, l'algorithme *intra* reçoit des pointeurs vers les fonctions d'émission point à point et de diffusion. Durant toute la durée de l'expérience, algorithme et application ne manipulent que des identifiants locaux propres à chaque cluster.

Une telle implémentation montre bien que les mécanismes de compositions sont complètement transparents pour l'application. Leur mise en œuvre dans une application patrimoniale est donc parfaitement envisageable.

Après avoir implémenté la partie applicative de la composition, il reste à implémenter le coeur de la composition, à savoir les coordinateurs.

7.1.2 Implémentation des nœuds coordinateurs

Une fois implémenté, l'algorithme des coordinateurs s'intègre dans une architecture (figure 7.1(b)) proche de celle utilisée par les autres nœuds. Dans cette architecture, ce code tient lieu d'application, reflétant ainsi le caractère automatique des coordinateurs.

D'autre part, on retrouve ici les deux instances *intra* et *inter* sous la forme de deux composants MUTEX. Comme pour les nœuds applicatifs, ces instances ignorent tout de la topologie hiérarchique. L'instance *intra* est initialisée avec les fonctions de communication locale utilisé par les nœuds applicatifs. Quant à l'instance *inter*, elle reçoit des pointeurs vers des fonctions de communication limitées à l'ensemble des coordinateurs. Ainsi un algorithme à diffusion émettra tantôt aux membres de son cluster, tantôt aux autres coordinateurs suivant qu'il représente l'instance *intra* ou *inter*.

Les algorithmes d'exclusion mutuelle peuvent donc être utilisés comme algorithme *inter* ou *intra*, sans que la notion de groupe n'impose de modification de leur code. Pour autant, notre approche n'est pas complètement transparente pour les algorithmes composés. En effet, elle nécessite l'ajout de la fonction *pendingRequest* à l'interface classique d'un algorithme d'exclusion mutuelle. Cette fonction, introduite dans la figure 2 de la section 6.2.1, permet aux coordinateurs de tester la présence de requêtes pendantes dans tel ou tel algorithme.

L'implémentation d'une telle fonction, qui s'apparente à un tissage d'aspect, n'est pas automatisée. Elle demande au programmeur de réifier manuellement cette information. Du point de vue système, et pour éviter toute attente active, nous avons choisi d'implémenter cette fonction sous la forme de deux interfaces requises et non d'une interface *pendingRequest* fournie par l'algorithme. Ainsi l'algorithme utilise la fonction *noMoreRequest* dès qu'il satisfait à l'ensemble des requêtes pendantes et la fonction *newRequest* lorsqu'il reçoit une nouvelle requête. À charge pour cette dernière fonction, implémentée dans le coordinateur, de le réveiller s'il était en attente d'une telle requête.

Avec cette implémentation, notre algorithme conserve bien son caractère générique, son code étant indépendant du choix de la composition. D'autre part, son intrusion dans les algorithmes d'exclusion mutuelle se limite à une réification des requêtes pendantes.

7.2 Évaluation des gains apportés par la composition

Cette section présente une évaluation de performance de notre approche par composition. Pour en mesurer les gains et en analyser les limites, nous avons mené sur une grille académique une série d'expériences comparant l'utilisation classique (à "plat") d'un algorithme d'exclusion mutuelle à son utilisation hiérarchique.

Pour cette première étude nous avons choisi de considérer l'algorithme de Mohamed Naimi et Michel Tréhel [NT87b] déjà utilisé dans le chapitre précédent. Ce choix s'explique par les deux considérations suivantes. Tout d'abord, il fallait considérer un algorithme permettant de réaliser des expériences à "plat" sur un grand nombre de nœuds dans un environnement n'offrant pas de mécanisme de diffusion. Comme on l'a vu dans la première partie de cette thèse, l'algorithme de Naimi-Tréhel est particulièrement bien adapté à ces conditions. D'autre part, il nous est paru intéressant de confronter notre approche à un algorithme à "plat" utilisant une structure logique dynamique. En effet, cette dynamicité représente un atout pour s'adapter à un environnement très hétérogène. Pour ces raisons, l'algorithme de Naimi-Tréhel constitue un bon point de comparaison pour évaluer notre approche.

7.2.1 Grille utilisée pour les tests

Les résultats présentés dans cette section proviennent d'expériences menées sur la plate-forme nationale de tests Grid'5000 [16] [CDD+06]. Grid'5000 regroupe 17 clusters répartis dans 9 villes françaises (voir carte de la figure 7.2) . Dans cette étude, nous avons utilisé un cluster dans chacune de ces villes.

Les machines ayant servi aux tests sont toutes équipées de processeur Bi-Opteron et de 2Go de RAM. Les villes sont reliées entres elles par des liens dédiés en fibre optique. Des mesures répétées ont montré une latence de connexions *intra* cluster (i.e., entre les nœuds d'un même cluster) de l'ordre de 0,05 ms, ainsi qu'une latence de connexions *inter* cluster variant de 6 à 20 ms suivant les villes (voir figure 7.3). Ces mesures pratiquement constantes font ressortir un rapport de l'ordre de 10^2 entre les latences *inter*-cluster et *intra*-cluster.

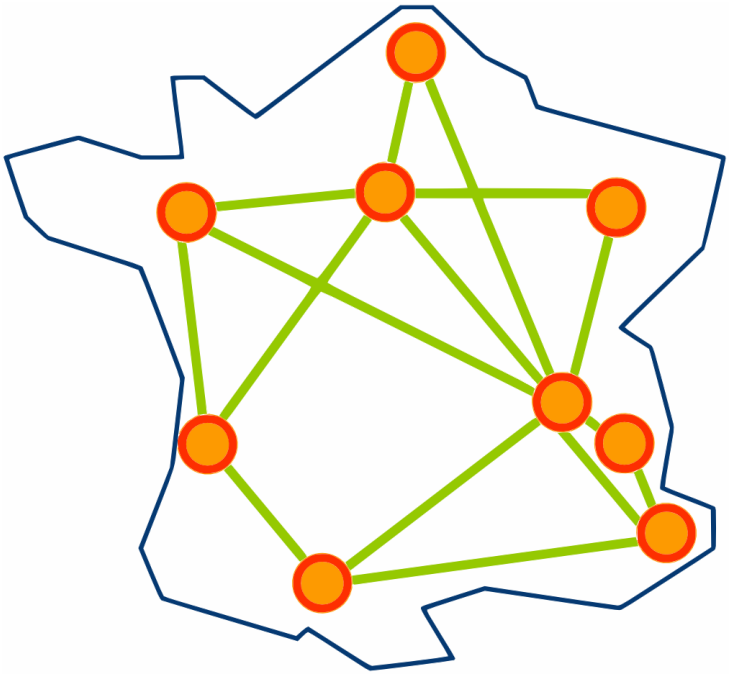


FIGURE 7.2 – Carte de la plate-forme nationale de test Grid'5000

Durant nos tests, le nombre de nœuds applicatifs N a été fixé à 180 répartis sur 20 processeurs distincts dans chacune des neuf villes de GRID'5000. Chaque nœud applicatif exécute un seul processus.

16. Grid'5000 est une plate-forme de test financée par le ministère de la recherche français, l'INRIA, le CNRS et les régions. Voir https://www.grid5000.fr

de \ à	orsay	grenoble	lyon	rennes	lille	nancy	toulouse	sophia	bordeaux
orsay	0.034	15.039	9.128	8.881	4.489	95.282	15.556	20.239	7.900
grenoble	14.976	0.066	3.293	15.269	12.954	13.246	10.582	9.904	16.288
lyon	9.136	3.309	0.026	12.672	10.377	10.634	7.956	7.289	10.078
rennes	8.913	15.258	12.617	0.059	11.269	11.654	19.911	19.224	8.114
lille	10.000	10.001	10.001	10.001	0.001	10.001	20.000	20.001	10.001
nancy	5.657	13.279	10.623	11.679	9.228	0.032	98.398	17.215	12.827
toulouse	15.547	10.586	7.934	19.888	19.102	17.886	0.043	14.540	3.131
sophia	20.332	9.889	7.254	19.215	16.811	17.238	14.529	0.051	10.629
bordeaux	7.925	16.338	10.043	8.129	10.845	12.795	3.150	10.640	0.045

FIGURE 7.3 – Latences moyenne de RTT sur Grid'5000

7.2.2 Spécification des expériences

7.2.2.1 Protocole

Pour chaque expérience, les processus applicatifs exécutent 100 demandes de section critique. Les mesures présentées ici correspondent au régime stationnaire : les 500 premiers accès et les 500 derniers ne sont pas mesurés. Ces sections critiques ont une durée moyenne de 10 ms (ce qui est du même ordre de grandeur que le délai d'un envoi de paquet entre deux clusters). Chaque expérience a été répétée 25 fois, et nous présentons ici une moyenne des résultats.

7.2.2.2 Paramètres

L'étude de performance présentée au chapitre 5 ayant montré l'énorme impact du type d'application sur les performances de l'algorithme d'exclusion mutuelle qu'elle utilise, nous avons ici choisi de paramétrer les expériences par le ratio $\rho = \beta/\alpha$ introduit dans la la section5.2.1.3. Rappelons que ce paramètre ρ exprime la fréquence d'accès à la section critique par un nœud et que plus il est petit et plus il y a de requêtes concurrentes.

Nous avons ainsi renouvelé les expériences avec différentes valeurs du paramètre ρ : $N/2$, N, $2N$, $3N$, $4N$ et $5N$ (où N est le nombre total des nœuds applicatifs de la grille). Comme on le verra dans les sections suivantes, les résultats de ces expériences caractérisent trois catégories distinctes d'applications suivant la charge qu'elles engendrent pour les coordinateurs. Pour simplifier l'étude, on distingue les trois types suivants :

- **Parallélisme faible** : $\rho \leq N$: L'exclusion mutuelle est fortement sollicitée. Un grand nombre de nœuds applicatifs demande la section critique. Presque tous les coordinateurs sont en attente du jeton *inter*.
- **Parallélisme intermédiaire** : $N < \rho \leq 3N$: Seulement quelques nœuds applicatifs sont en compétition pour obtenir la section critique. Par conséquent, seule une partie des coordinateurs ont fait une demande pour le jeton *inter*.
- **Parallélisme fort** : $3N \leq \rho$: Le parallélisme est important dans l'application, les demandes de section critique sont éparses. L'exclusion mutuelle est peu sollicitée. Le nombre total de requêtes sur l'ensemble des nœuds applicatifs est peu conséquent et le nombre de coordinateurs demandant le jeton *inter* est donc faible.

Il est à remarquer que la classification présentée ici ne recoupe pas exactement celle présentée à la section 5.2.1.3 lors de l'étude de performance de notre algorithme tolérant aux fautes. En effet, la composition joue ici un rôle concentrateur au niveau des coordinateurs. Les phénomènes observés dans la couche *inter* sont donc accentués par la composition.

7.2.2.3 Métriques

Dans chaque étude nous considérerons les trois métriques suivantes :
- **le nombre de messages *intra*-cluster :** Nombre total de messages émis dans les réseaux locaux pour l'ensemble des clusters.
- **le nombre de messages *inter*-cluster :** Nombre total de messages émis sur le réseau inter-connectant les clusters.
- **le temps moyen d'attente du jeton :** Délais moyen s'écoulant entre l'envoi d'une requête d'un nœud applicatif et l'accès à la section critique.

Notons, que contrairement à ce qui était fait dans le chapitre 5, on ne distinguera pas ici les diffusions (*broadcast*) des émissions point à point. En effet, le réseau WAN interconnectant les clusters de GRID'5000, tout comme dans la plupart des autres grilles de calcul, n'est pas équipé d'un service de diffusion. Diffuser un message à l'ensemble des nœuds de la grille revient alors à émettre N messages point à point : il n'y a donc pas de différence entre le nombre de messages émis et le nombre de messages reçus.

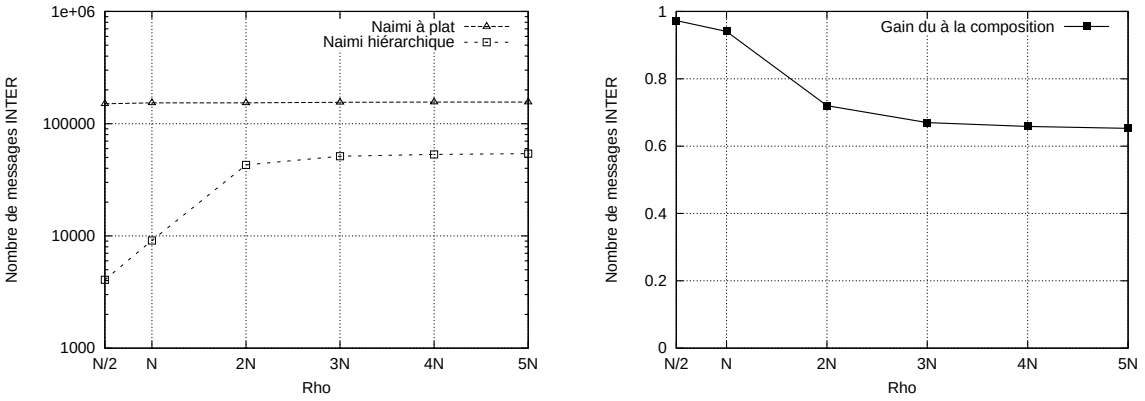
7.2.3 Étude de la complexité en nombre de messages *inter*

La figure 7.4(a) présente pour chacune des deux approches l'évolution du nombre total de messages *inter* émis au cours de l'expérience lorsque le degré de parallélisme augmente. Pour la composition, cette mesure comptabilise les échanges entre les différents coordinateurs. Tandis que pour l'approche à plat elle représente le nombre de messages émis entre deux nœuds applicatifs appartenant à deux clusters différents. Sur la figure 7.4(b) on trouvera la mesure relative aux économies de messages *inter* réalisées avec l'utilisation de la composition.

Sur la figure 7.4(a), nous pouvons tout d'abord constater que le nombre de messages *inter* générés par l'algorithme Naimi à *plat* reste identique quelle que soit la valeur du paramètre ρ. En effet, le comportement des mécanismes mis en jeux dans cet algorithme est complètement indépendant de la charge du système.

On observe aussi que cet algorithme génère un volume important de messages *inter* cluster. Ce fort trafic est induit par la non correspondance entre les topologies logiques imposées par l'algorithme de Naimi et la topologie physique du réseau. Ainsi, même si un site désirant entrer en section critique se trouve dans le même cluster que la racine de l'arbre des *LAST*, sa requête pourra être retransmise à l'extérieur de ce cluster avant de l'atteindre. Il est même fort probable qu'elle revienne et ressorte plusieurs fois en remontant dans l'arbre. On mesure bien sur la figure 7.4(a) le surcoût de messages *inter* induit par cette non correspondance physique/logique.

Comparons maintenant ces résultats avec les mesures de l'algorithme hiérarchique. Pour commencer on peut remarquer que quel que soit le type d'application, le nombre



(a) Nombre total de messages *inter* (b) Gain relatif sur le nombre de messages *inter*

FIGURE 7.4 – Étude comparative du nombre de message *inter*

de messages *inter* émis est très largement inférieur à celui de l'algorithme à "plat". La figure 7.4(b) nous montre des différences allant de 60% à plus 90% suivant les valeurs de ρ.

Pour comprendre cette variation dans les écarts, étudions l'évolution de la complexité en messages *inter* pour l'approche par composition. La courbe montre deux allures distinctes : dans un premier temps entre 0 et $2N$ la complexité augmente linéairement puis elle se stabilise et reste presque constante pour les valeurs de ρ supérieurs à $3N$. Cette évolution s'explique par deux phénomènes complémentaires.

Le premier est indépendant de ρ et se mesure sur les performances de la composition pour les applications **fortement parallèles**. Il résulte d'un effet "*structurel*" de la composition. En effet, notre approche partitionne le système suivant la topologie physique. Ce faisant, elle réduit le nombre de participants à chacun des algorithmes. Ceci a pour effet direct de réduire la taille des arbres parcourus par les requêtes. Le nombre moyen de messages nécessaire à l'obtention d'une section critique s'en trouve réduit d'autant. La charge du système n'ayant pas d'impact sur cet effet "*structurel*", les gains qu'il engendre sont constant et donc à prendre en compte pour toutes les valeurs de ρ. En revanche, puisque notre algorithme rapproche structure logique et topologie physique, ces gains seront très fortement liés à la topologie de la grille. Une étude exhaustive de l'influence de la topologie sur la hiérarchisation sera présentée à la section 7.4.

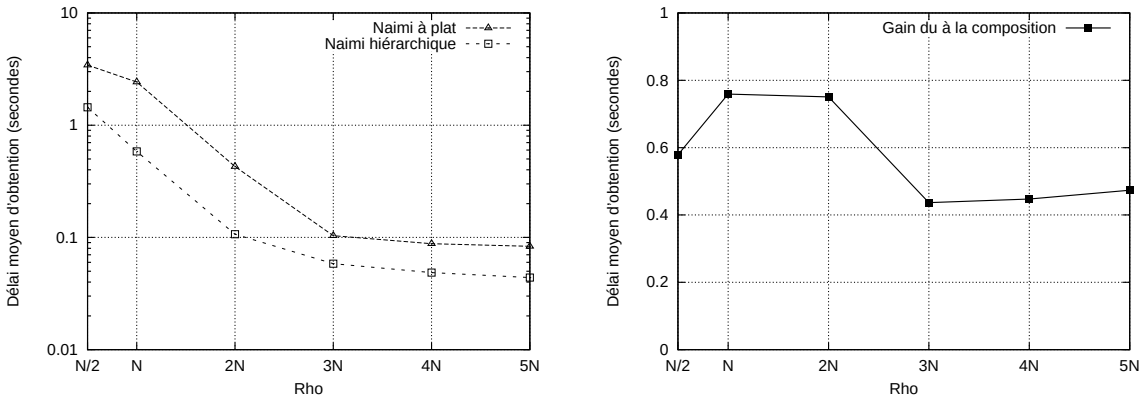
À ce premier "effet" de la composition viennent s'ajouter des gains dûs à l'effet d'*agrégation naturelle* des messages *inter*. Cet effet, que nous avions déjà observé dans l'exemple de la figure 6.3, explique ici l'écart conséquent entre les deux approches pour des applications **faiblement et moyennement parallèles**. En effet, de petites valeurs de ρ impliquent une grande concurrence d'accès à la section critique de la part des nœuds applicatifs. L'*agrégation naturelle* est alors à son maximum : une requête de coordinateur correspond alors à de nombreuses requêtes applicatives. Dans ce cas notre approche se montre donc très économe sur l'envoi des message *inter* en comparaison des algorithmes à plat.

Lorsque la concurrence d'accès à la section critique diminue tandis que ρ augmente,

cet effet d'agrégation des requêtes *intra* au niveau du coordinateur diminue. Les courbes nous montrent qu'il reste efficace jusqu'à $2N$ et n'est plus significatif lorsque ρ devient supérieur à $3N$.

7.2.4 Étude de la latence moyenne d'obtention de la section critique

Sur la figure 7.5(a), on retrouve pour chacune des deux approches le délai moyen d'attente sur l'ensemble des accès à la section critique de l'expérience tandis que la figure 7.5(b) montre les gains relatifs qui résultent de l'utilisation de notre algorithme de composition.



(a) Délai moyen d'obtention du jeton

(b) Gain relatif sur le délai moyen d'obtention

FIGURE 7.5 – Étude comparative du délai moyen d'obtention du jeton

Pour commencer l'étude de ces graphiques, remarquons que quelle que soit l'approche considérée, le temps d'attente moyen augmente lorsque ρ diminue. En effet, tout comme dans les expériences du chapitre précédent, la file d'attente constitue là encore un goulot d'étranglement pour les applications *faiblement et moyennement parallèles*.

La comparaison des deux courbes sur la figure 7.5(a) montre qu'à l'instar de la complexité, notre approche par composition est toujours plus efficace en terme de latence qu'un algorithme à plat et ce quel que soit le type d'application. Mais contrairement à ce que nous avons observé dans la section précédente, la figure 7.5(b) montre deux seuils d'écart : pour les applications *fortement parallèles* l'écart entre les deux approches se maintient à un facteur de 45% tandis que pour les applications *faiblement et moyennement parallèles* notre approche se montre environ 4 fois plus rapide. La différence d'allure entre les courbes des figures 7.5(b) et 7.4(b) s'explique par l'impact de la taille de file d'attente sur le délai moyen d'accès à la section critique.

On retrouve ici les effets *structurels* de la composition. En effet, en réduisant la taille des arbres, on réduit aussi le temps nécessaire à l'acheminement d'une requête. Ces gains sont là encore indépendants du type d'application. Pour observer leur impact, il nous faut considérer les mesures pour les applications *fortement parallèles*. En effet, comme nous

allons le voir, les autres effets sont quasiment inexistants pour ce type d'application. Si l'on observe la figure 7.5(b), on peut donc dire que les gains *structurels* permettent de réduire d'environ 45% le temps moyen d'attente du jeton.

Le deuxième effet entrant en jeu pour déterminer le temps moyen d'attente de la section critique est celui de la *préemption naturelle*. Comme nous l'avions vu à la section 6.3, cet effet réordonne les requêtes pour éviter au jeton des allers-retours inutiles sur le réseau WAN inter-connectant les clusters. Plus le nombre de sites en attente est élevé, plus ce réordonnancement sera efficace.

Mais il nous faut prendre en compte ici l'impact de la file d'attente. En effet, la taille de la file grandissant tandis que ρ diminue, le temps passé dans la file d'attente augmente en même temps que l'efficacité de la préemption. Si bien que proportionnellement, les gains de la préemption restent constants entre N et $2N$.

Enfin, dans la figure 7.5(b), on remarque que la différence de latence entre l'algorithme à plat et notre algorithme de composition se réduit lorsque ρ diminue de N à $N/2$. Pourtant, le nombre de sites en attente augmentant, le nombre de préemptions devrait lui aussi augmenter. Il s'agit en fait ici d'un effet de bord dû au décalage dans les tailles des files d'attente. En effet, pour l'algorithme à plat le nombre de sites en attente est déjà très important pour $\rho = N$. Il commence même à atteindre son maximum. Ceci explique pourquoi l'on observe un ralentissement dans l'augmentation de la taille de la file lorsque ρ diminue pour passer à $N/2$. À l'inverse, pour $\rho = N$, la préemption a permis à la composition de réduire le temps d'attente des sites et donc la taille de la file. Ce faisant, elle reste loin d'atteindre la latence maximum et reste donc plus sensible à la diminution du paramètre ρ. Ceci explique que l'on observe pas pour la composition la diminution de la croissance des délais d'attentes précédemment observée pour l'algorithme à plat. On comprend alors pourquoi le gain relatif diminue entre $\rho = N$ et $\rho = N/2$ alors que la "préemption naturelle" continue d'augmenter.

Cette étude de la latence moyenne d'obtention de la section critique permis entre autres de mettre en évidence l'impact de file d'attente sur les performances de la composition. Il nous faudra donc le garder à l'esprit lors du choix des compositions efficaces dans la section 7.3

7.2.5 Étude de la complexité en nombre de messages *intra*

La figure 7.6 est le pendant de la figure 7.4(a) pour les messages *intra*. Avant d'en faire l'étude, précisons ce que l'on comptabilise ici. Pour la composition, il s'agit des échanges entre les nœuds applicatifs d'un même cluster ainsi que les messages émis entre un coordinateur et les nœuds applicatifs de son cluster. Autrement dit, il s'agit du nombre de messages émis par l'ensemble des instances de l'algorithme *intra*. Pour l'algorithme classique on retrouve les messages émis entre nœuds d'un même cluster, auquel on ajoute les émissions entre des nœuds de clusters différents. En effet, dans une topologie de type grille, les machines ne sont pas toutes reliées directement au réseau WAN. Pour que l'information traverse la grille, elle doit traverser le réseau local de l'émetteur, puis la couche WAN, pour terminer par être envoyée au destinataire sur son réseau local. Ce type de message sera donc comptabilisé ici comme deux messages *intra*.

Dans leur forme, les courbes de la figure 7.6 sont similaires à celle des mesures des

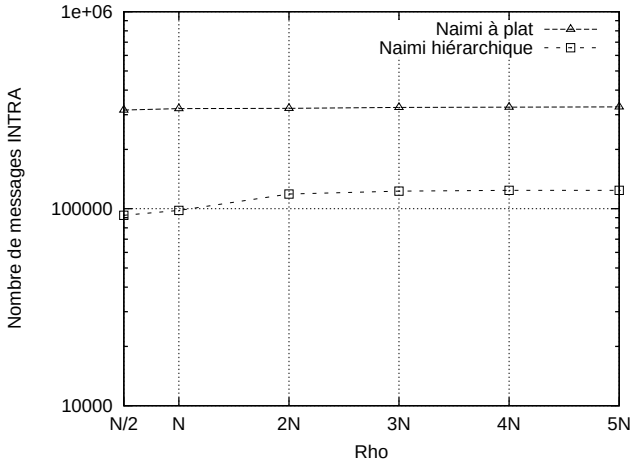


FIGURE 7.6 – Étude comparative du nombre de messages *intra*

messages *inter* (figure 7.4(a)). On retrouve le gain structurel indépendant du type d'application. Mais la différence d'échelle dans le nombre des participants aux algorithmes n'explique pas entièrement le très grand écart entre les deux approches. En effet, vient s'ajouter ici l'économie de messages *intra* due à l'effet de *filtre naturel*. Ainsi, une requête engendre des messages dans le cluster d'où elle est émise et, s'il est différent, dans le cluster où se trouve le jeton. Tout comme pour le gain structurel, cette économie est indépendante de la charge du système.

Enfin, cette figure permet de mesurer l'effet d'*agrégation naturelle* des requêtes *intra*. Comme on l'a vu dans l'exemple de la section 6.3.4, un coordinateur dans l'état $WAIT\_FOR\_OUT$ ne ré-émet pas de messages *intra* s'il reçoit une requête d'un autre coordinateur. L'efficacité de ce mécanisme dépend du nombre de requêtes concurrentes et donc du degré de parallélisme de l'application. Les mesures montrent que les économies dues à cet effet ne sont pas très importantes relativement au nombre total de messages *intra* émis. En effet, en terme de messages *intra*, les économies réalisées sont proportionnelles à la taille des clusters. Elles souffrent donc de l'effet *structurel* dû à la composition.

7.2.6 Étude de la complexité globale en nombre de messages

Pour terminer cette étude comparative entre un algorithme à plat et sa version hiérarchique, nous nous intéressons à leur complexité globale, *i.e.*, au nombre total de messages émis. Dans ce calcul on ne distingue pas le type des messages, pas plus qu'on ne s'intéresse à leur mode d'émission. Ainsi contrairement à ce que nous prenions en compte dans la section précédente, un message dont l'émetteur et le destinataire ne se trouvent pas dans le même cluster ne comptera pas comme deux messages *intra* et un message *inter* mais bien comme un unique message.

Additionner messages *intra* et messages *inter* sans discernement peut sembler aussi étrange que d'"*comparer des bicyclettes avec des laitues*" [17]. Le but de cette mesure est

17. Traduction littérale de l'idiomatique brésilien "comparar alfoce com bicicleta" très proche de l'ex-

double. D'une part, elle va permettre de mesurer le surcoût en messages dû à l'ajout des coordinateurs. D'autre part, elle nous permet d'évaluer notre approche dans un autre type d'environnement. En effet, on pourrait imaginer utiliser notre mécanisme de composition sur un réseau local regroupant un très grand nombre de machines. Les clusters deviennent alors des groupes logiques de machines. Dans un tel environnement il n'y a plus de différence entre messages *intra* et messages *inter* : tous sont des messages locaux.

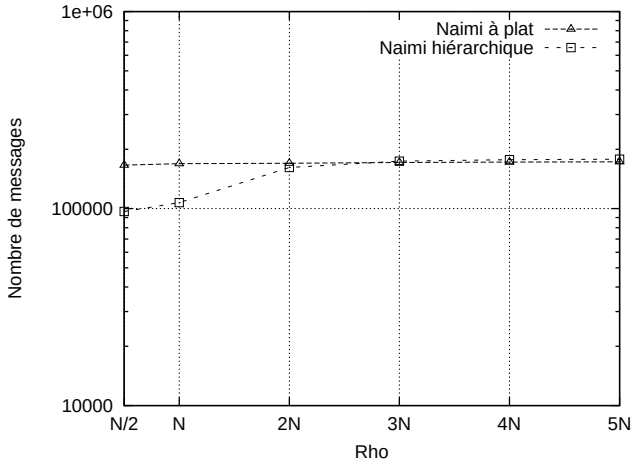


FIGURE 7.7 – Étude comparative du nombre de messages *intra*

Les résultats présentés sur la figure 7.7, révèlent que les deux approches sont équivalentes en terme de messages pour une application *fortement parallèle*. Ceci montre que le surcoût engendré par les requêtes des différents coordinateurs est couvert par le gain structurel présenté dans les sections précédentes. Ce résultat s'explique surtout par l'effet de *filtre naturel*. En limitant le nombre de coordinateurs ré-émettant localement les requêtes, il limite d'autant le surcoût.

Pour les valeurs de $\rho < 2N$, notre approche se montre un peu plus économe en messages. Pour les applications *faiblement parallèles* on observe un gain de 30%. Ces économies sont la somme des gains en messages *intra* et *inter* faites par les mécanismes d'*agrégation naturelle*.

Cette étude de la complexité globale montre qu'il pourrait y avoir une amélioration des performances avec l'utilisation de la composition au niveau d'un cluster. Mais il ne s'agit là que d'une première évaluation qui devra être confirmée par de nouvelles mesures.

7.3 Évaluation de différentes compositions

L'aspect générique de notre approche présente l'avantage de pouvoir composer la plupart des algorithmes de la littérature. Les quelques restrictions ont été étudiées dans la section 6.5. Il est donc naturel de se demander si certaines compositions se montrent à l'usage plus performantes que d'autres.

pression française "additionner des pommes et des poires"

Cette section présente une étude comparative des gains en terme de délai d'obtention du jeton et de complexité en nombre de messages entre différentes compositions. De cette étude nous essaierons de dégager les bons choix de composition en fonction du type d'application.

7.3.1 Compositions évaluées

Cette étude concerne le choix des algorithmes *inter* et *intra* permettant d'obtenir une composition optimal. Comme dans le reste de cette thèse, nous ne considérons ici que des algorithmes d'exclusion mutuelle basés sur la circulation d'un jeton.

Parmi ces algorithmes, nous avons choisi un représentant dans les sous-familles décrites dans la taxonomie de la section 2.5 : non structuré, structure statique et structure dynamique. Les algorithmes sélectionnés utilisent trois structures logiques différentes pour acheminer les requêtes :

- **Arbre logique :** Algorithme de Naimi et Tréhel.
- **Anneau logique :** Algorithme de Martin.
- **Graphe logique complet :** Algorithme de Suzuki et Kasami.

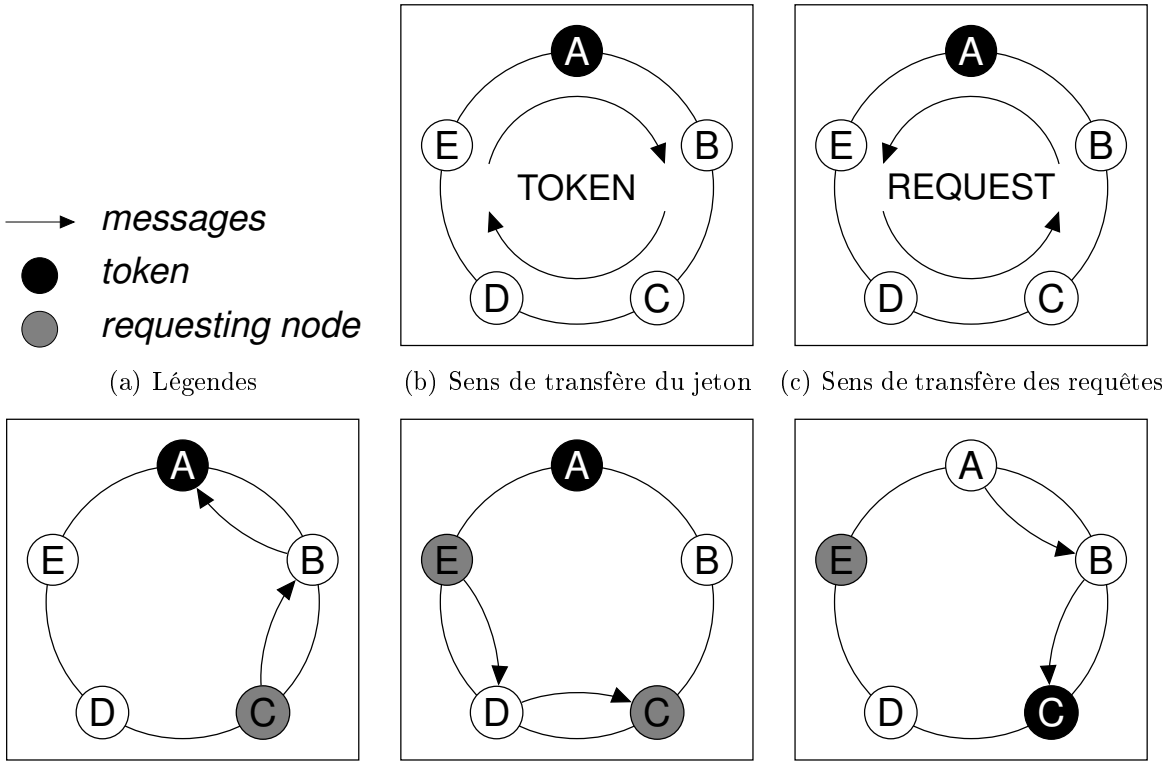
7.3.1.1 Algorithme de Nami et Tréhel

L'algorithme de Naimi-Tréhel [NT87b] a déjà été décrit dans le chapitre 4 à la section 4.1. Rappelons qu'il utilise un **arbre dynamique** pour acheminer les requêtes. Chaque requête étant acheminée jusqu'à la racine de l'arbre, le nombre moyen de requêtes transmises par section critique est logarithmique. À la fin d'une section critique, le jeton est directement transmis au premier nœud ayant fait une requête.

7.3.1.2 Algorithme de Martin

L'algorithme de Martin [Mar85] est une optimisation de l'algorithme à **anneau** de Gérard Le Lann [Lan78]. Comme lui, il utilise le principe du jeton tournant, mais il y ajoute des requêtes. En effet, comme on l'a vu dans la section 6.5.1, il ne peut y avoir d'état quiescent dans l'algorithme de Le Lann. C'est pour éviter cette circulation perpétuelle du jeton que Martin ajoute un mécanisme de requête. Grâce à ce dernier, s'il n'y a plus de requête pendante, un site peut conserver le jeton au sortir de sa section critique. Il ne le libérera qu'à la réception d'une requête.

L'idée principale de cet algorithme résulte dans le sens de circulation de ses requêtes. En effet, comme on peut le voir sur les figures 7.8(b) et 7.8(c), le jeton circule dans un sens tandis que les requêtes sont retransmises de site en site dans le sens contraire. Cette circulation à double sens permet d'accélérer la rencontre entre les requêtes et le jeton. De plus, elle permet au site ayant retransmis la requête de savoir s'il reste des requêtes pendantes ou si toutes ont été satisfaites. En effet, un site ayant retransmis une requête est sûr de voir passer le jeton. La demande étant alors pour lui satisfaite, la requête n'est plus pendante. Cette connaissance fine des requêtes pendantes permet à l'algorithme de réaliser la propriété de *quiescence immédiate* définie à la section 6.5.1.



(a) Légendes (b) Sens de transfère du jeton (c) Sens de transfère des requêtes

(d) Le site C émet une requête (e) Le site E émet une requête (f) Le site A libère le jeton

FIGURE 7.8 – Exemple d'exécution de l'algorithme Martin

Les figures 7.8(d), 7.8(e) et 7.8(f) déroulent une séquence de cet algorithme dans un anneau composé des 5 nœuds : A, B, C, D et E. À l'état initial, le nœud A possède le jeton. Le nœud C désire accéder à la section critique et émet une requête vers B (figure 7.8(d)). Lorsque le site B la reçoit, il la retransmet au site A et il enregistre qu'une requête est en attente. À partir de cet instant, et tant qu'il n'aura pas vu le jeton, le site B ne retransmet plus aucune requête et s'il désire entrer en section critique il se contentera d'attendre le passage du jeton.

La figure 7.8(d) illustre les retransmissions successives d'une requête du site E. Dans cet exemple, cette demande n'est pas ré-émise par le site C. En effet, tout comme le site B, le site C est déjà sûr de voir passer le jeton puisqu'il a enregistré une requête (la sienne).

Lorsque le site A sort de section critique, le jeton circule en sens inverse jusqu'à C (figure 7.8(f)). Puis, au sortir de la section critique, le site C libère le jeton qui finit par parvenir au site E en transitant par D.

Sur cet exemple, on voit bien que la complexité de l'algorithme de Martin dépend fortement du degré de parallélisme de l'application. Ce point constituera un des critères principaux pour le choix de cet algorithme.

7.3.1.3 Algorithme de Suzuki-Kasami

L'algorithme de Ichiro Suzuki et Tadao Kasami [SK85] est l'un des algorithmes d'exclusion mutuelle à diffusion les plus connus et peut-être l'un des plus simples à mettre en œuvre. Son fonctionnement ressemble beaucoup à celui de l'algorithme utilisé dans l'exemple figure 6.3.

Cet algorithme utilise deux vecteurs de données et une file d'attente. Le premier vecteur, dénommé LR, est lié au jeton et est transmis avec lui. Ce vecteur contient, pour chaque site, le nombre de fois où il a pu accéder à la section critique. Le deuxième vecteur, noté RN, est présent sur chaque site et contient pour chaque site un compteur du nombre requêtes. À un instant donné, les vecteurs RN peuvent différer suivant les sites. Une différence entre deux valeurs locales correspond à un requête pendante. Lorsque plus aucune requête n'est envoyée, chacun des sites finit par posséder le même vecteur RN. La troisième structure utilisée par l'algorithme de Suzuki-Kasami est une file d'attente attachée au jeton. Cette file assure la vivacité (et même l'équité faible) de l'algorithme. Elle est mise à jour à la fin de chaque section critique.

Pour mieux comprendre son fonctionnement, nous allons suivre l'exemple dont l'exécution est présentée sur la figure 7.9. Celui-ci commence alors que le site A exécute une section critique (figure 7.9(b)). Il s'agit de sa $7^{\text{ème}}$ section critique ($RN[A] = 7$) et l'on peut vérifier qu'il en a déjà exécuté 6 autres ($LR = [A] = 6$).

Le site C émet alors une requête (figure 7.9(c)) : il incrémente son propre compteur dans son vecteur RN puis diffuse cette valeur à l'ensemble des sites du système. À la réception de cette dernière, chaque site du système met à jour $RN[C] = 9$. Sur la figure 7.9(d) le site D émet à son tour une requête. Elle est alors progressivement enregistrée dans tous les vecteurs RN ($RN[D] = 3$).

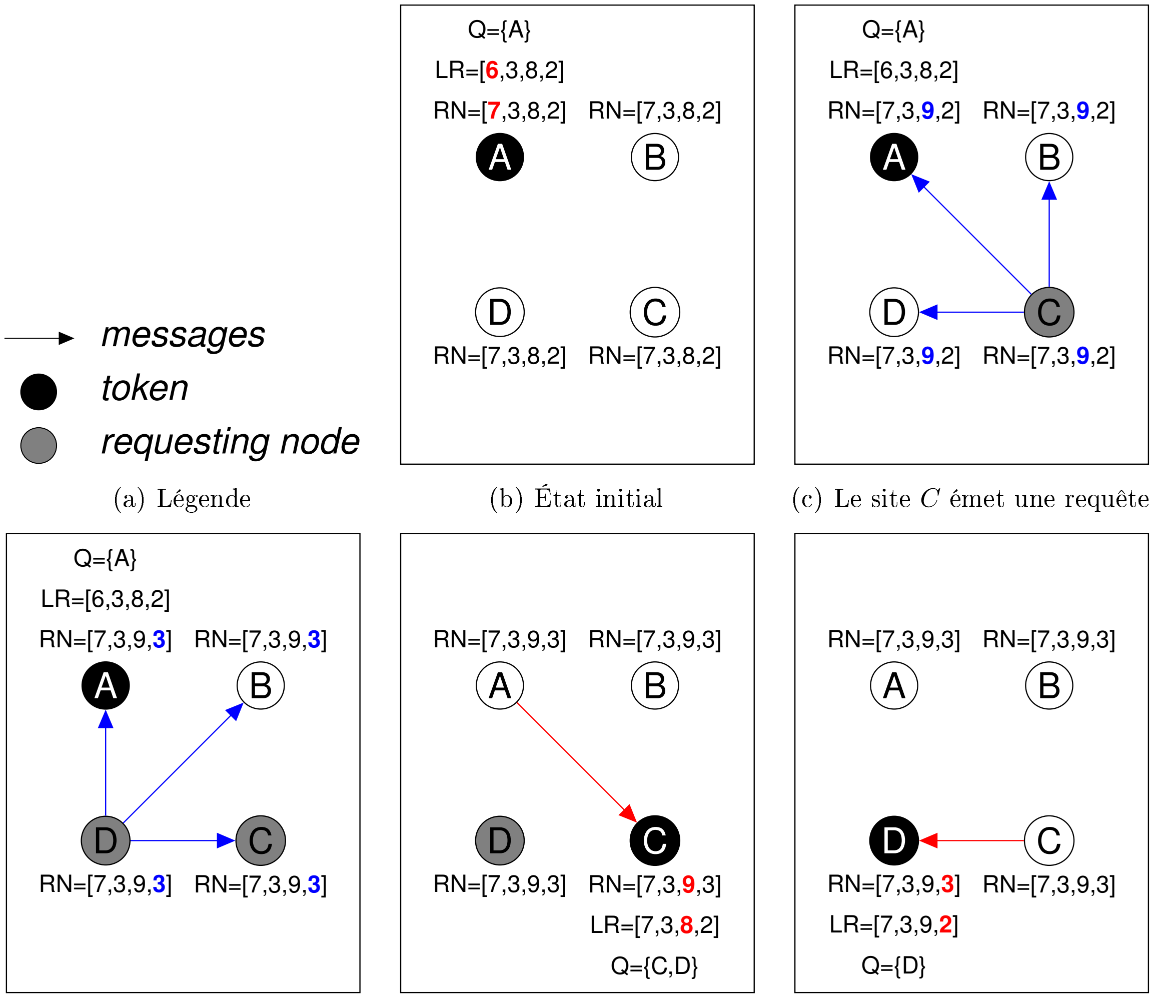
Lorsque le site A termine sa section critique. Il commence par enregistrer cet accès dans le jeton : $LR = [A] = 7$. Puis il compare son vecteur RN avec le vecteur LR du jeton. Toute différence correspond à une requête pendante et est ajoutée dans la file d'attente Q si elle n'y est pas déjà présente. Dans cet exemple il y a deux différences entre les vecteurs et donc deux sites en attente. Le site A ajoute donc C et D dans Q. Il envoie alors le jeton au premier site de la file d'attente : C.

Lorsque le site C sort à son tour de section critique, il met à jour son nombre d'accès puis compare les vecteurs. Il y a toujours une différence pour le site D mais celui-ci est déjà enregistré dans Q. Le site C se contente de se retirer de la file puis envoie le jeton au site D.

Plusieurs optimisations dans la gestion de l'information ont été proposées dans la littérature. Certaines, par exemple, se passent de la file d'attente en se contentant de comparer les vecteurs, un système de tourniquet assurant alors la vivacité [RA83]. Ces optimisations conservent cependant la mécanique générale des diffusions.

7.3.1.4 Compositions et notations

Les expériences présentées dans cette section étudient donc différentes compositions utilisant chacun de ces trois algorithmes. Pour simplifier cette analyse, nous avons choisi de ne considérer que des compositions utilisant le même algorithme (*intra*) dans tous les



(a) Légende (b) État initial (c) Le site C émet une requête

(d) Le site D émet une requête (e) Le site A libère le jeton (f) Le site C libère le jeton

FIGURE 7.9 – Exemple d'exécution de l'algorithme Suzuki

clusters de la grille.

Dans la suite de cette thèse nous noterons Naimi pour l'algorithme Naimi-Tréhel et Suzuki pour l'algorithme Suzuki-Kasami. Nous avons aussi choisi de noter les différentes compositions sous la forme "*Algorithme Intra-Algorithme Inter*". Ainsi une composition utilisant Suzuki comme algorithme *inter* et où chacun des clusters instancient l'algorithme de Naimi sera noté : Naimi-Suzuki.

Afin de bien séparer l'impact du choix de l'algorithme *inter* de celui des algorithmes *intra*, nous fixerons successivement chacune de ces deux couches. Comme dans l'étude précédente et pour les mêmes raisons (voir section 7.2), nous gardons ici l'algorithme Naimi-Trehel comme point de comparaison. Ainsi la section 7.3.3 présente une étude comparative des compositions : Naimi-Marti, Naimi-Naimi et Naimi-Suzuki. De même la section 7.3.4 étudie l'algorithme *intra* en comparant les compositions : Martin-Naimi, Naimi-Naimi et Suzuki-Naimi.

7.3.2 Paramètre des mesures et protocole d'expérimentation

Nous avons choisi de comparer les différentes compositions d'algorithmes suivant les deux mesures suivants :

- le temps d'attente moyen pour obtenir le jeton - *i.e.*,, le temps entre le moment de la demande par un nœud applicatif et le moment où ce nœud reçoit effectivement le jeton,
- le nombre de messages *inter* échangés.

Parmi ces deux critères, on cherchera d'abord à optimiser la latence d'accès à la section critique. À latence équivalente, le choix de la composition se portant alors sur celle minimisant la charge réseau au niveau *inter*.

Le but de cette étude étant de permettre l'optimisation du choix de la composition en fonction du type d'application, nous reprenons ici les différentes valeurs de ρ présentées dans la section 7.2.2.2. La grille et le protocole d'expérience restent les mêmes que ceux définis à la section 7.2.1.

7.3.3 Choix de l'algorithme *inter*-cluster

7.3.3.1 Étude des délais moyens d'obtention

Pour expliquer les différences de *délai d'obtention* du jeton, nous utiliserons les notations suivantes :

- T : le délai moyen de transmission d'un message entre deux coordinateurs.
- T_{req} : le délai moyen d'acheminement d'une requête *inter*.
- T_{token} : le délai moyen pour acheminer le **jeton** *inter* d'un coordinateur au coordinateur suivant. Ce dernier pouvant alors libérer son jeton *intra*.
- T_{pendCS} : le délai moyen pour satisfaire toutes les demandes de jeton *inter*, pendantes au moment de l'émission d'une requête *inter*. Ce délai correspond au temps passé dans la file d'attente.

La figure 7.10 présente des mesures du temps d'attente moyen pour obtenir une section critique en fonction du choix de l'algorithme *inter*. On y trouve trois compositions : Naimi-Martin, Naimi-Naimi et Naimi-Suzuki.

Avec une application **faiblement parallèle** ($\rho \leq N$), il y a de nombreux accès à la section critique. Un coordinateur qui réclame le jeton doit attendre que toutes les requêtes *inter* pendantes soient satisfaites. Le délai d'attente (T_{pendCS}) est alors plus grand que celui nécessaire aux transmissions de sa requête (T_{req}) et recouvre complètement ce dernier. Par conséquent le *délai d'obtention* est égal à T_{pendCS}+T_{token}.

D'autre part, le recouvrement du temps de transmission des requêtes (T_{req}) explique les faibles différences entre les temps moyens d'obtention des différentes compositions. En effet, dans ce cas, seul le temps d'envoi du jeton (T_{token}) pourrait faire la différence. Or les 3 algorithmes n'utilisent qu'un message (T) pour transférer le jeton : Suzuki et Naimi par construction et Martin grâce à la concurrence. La probabilité que le prédécesseur du coordinateur possédant le jeton soit à l'état *REQ* étant très proche de 1, ce prédécesseur pourra préempter le jeton avant de le faire suivre.

Concernant les applications avec un degré de **parallélisme moyen**, nous pouvons remarquer un comportement presque identique pour les compositions Naimi-Naimi et

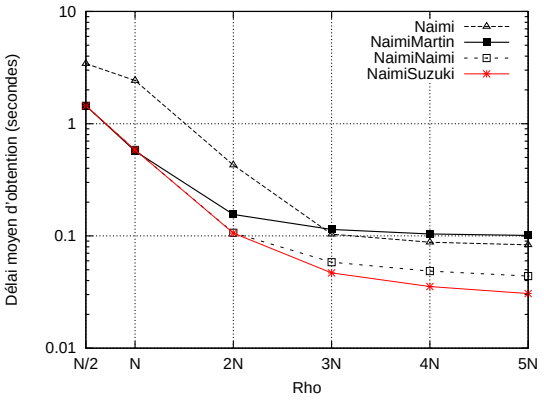


FIGURE 7.10 – Délai moyen d'obtention du jeton suivant les différentes compositions

Naimi-Suzuki et un coût plus important pour Naimi-Martin. Avec ce niveau de parallélisme, il y a toujours recouvrement du temps de transmission mais seuls certains coordinateurs sont à l'état *REQ*. Pour Naimi-Martin il faut donc plusieurs transmissions successives dans l'anneau pour acheminer le jeton à un coordinateur intéressé par la section critique ($T_{token} > T$).

Enfin, pour $\rho \geq 3N$ l'application a un degré de **parallélisme fort**. Il n'y a donc que peu de concurrence. La file de requêtes *inter* étant petite, le temps passé par une requête dans la file (T_{pendCS}) ne couvre plus le temps de transmission des requêtes (T_{req}). Le *délai d'obtention* du jeton *inter* comprend alors le délai pour acheminer une demande de section critique (T_{req}) et le délai pour récupérer le jeton (T_{token}).

Suzuki utilisant une diffusion, son délai T_{req} est égal à T. Celui de Naimi est en moyenne de $log(N) * T$. Tandis qu'avec Martin qui est le moins efficace, il faut un délai moyen de $N/2 * T$ pour acheminer une requête.

Le délai d'acheminement du jeton (T_{token}) reste lui égal T pour Naimi et comme pour Suzuki. En revanche, pour Martin, il faut encore un délai moyen de $(N/2) * T$ pour atteindre le premier (et souvent seul) coordinateur demandeur du jeton.

En conclusion, si l'on veut minimiser le *délai moyen d'obtention* de la section critique pour une application fortement parallélisée ($\rho \geq 3N$), le choix de Suzuki comme algorithme *inter* s'impose. En effet, il ne consomme qu'une diffusion (T) et qu'un envoi de message (T).

On peut résumer cette étude de la latence d'obtention du jeton pour les différentes compositions dans le tableau suivant (figure 7.11) avec $1 < K < N$:

Composition \ Parallélisme	Faible	Moyen	Fort
Naimi-Suzuki	$T_{pendCS} + T$	$T_{pendCS} + T$	$T + T$
Naimi-Martin	$T_{pendCS} + T$	$T_{pendCS} + K * T$	$(N/2) * T + (N/2) * T$
Naimi-Naimi	$T_{pendCS} + T$	$T_{pendCS} + T$	$log(N) * T + T$

FIGURE 7.11 – Délai moyen d'obtention du jeton pour chaque composition

7.3.3.2 Étude du nombre de messages *inter*-cluster

Intéressons nous maintenant au nombre de messages *inter* émis en analysant les résultats de la figure 7.12.

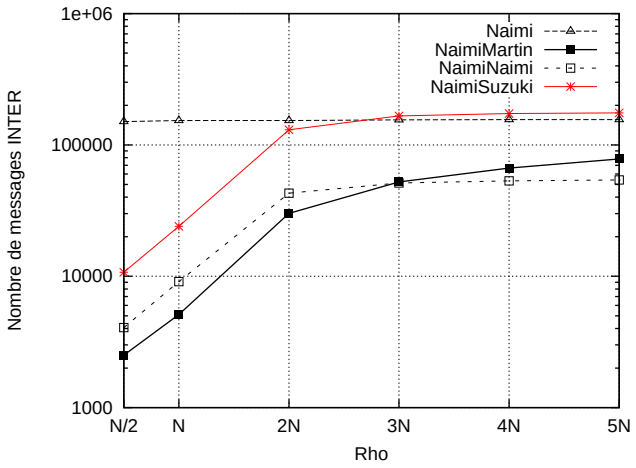


FIGURE 7.12 – Nombre total de messages suivant les différentes compositions

Pour les compositions Naimi-Suzuki et Naimi-Naimi, le nombre moyen de messages *inter* échangés entre les coordinateurs sont respectivement N et $log(N)$ messages pour effectuer une requête, auxquels s'ajoute un message pour acheminer le jeton.Ce qui explique que Naimi-Naimi reste le plus intéressant des deux quel que soit ρ.

Pour la composition Naimi-Martin, le nombre de messages dépend du nombre de coordinateurs de l'anneau ayant fait une requête *inter*. La complexité dépend donc du nombre de requêtes *intra* et par conséquent de ρ.

Pour des applications à **faible parallélisme** ($\rho \leq N$), la probabilité que tous les coordinateurs soient en état *inter REQ* est forte. Le nombre de messages *inter* échangés est alors minimum (2 messages par accès). La composition Naimi-Martin est alors la moins coûteuse en messages *inter*.

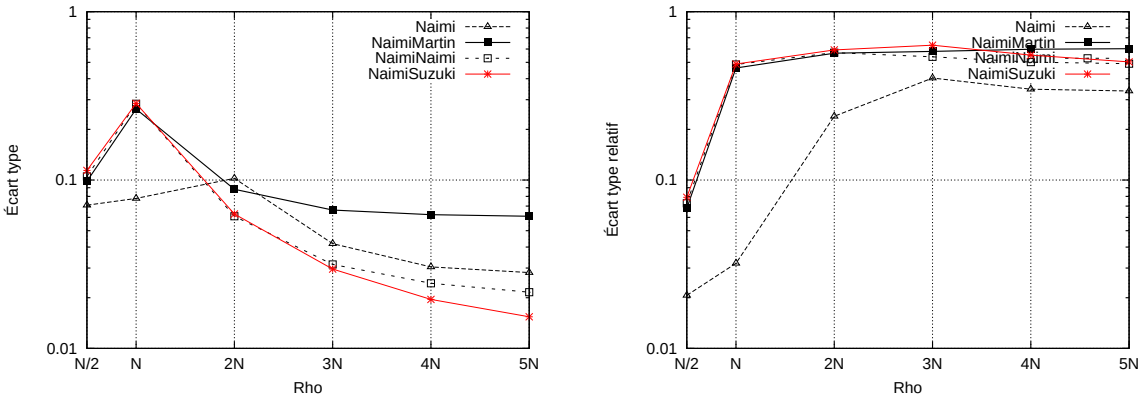
Lorsque ρ augmente, la probabilité que le successeur (resp. prédécesseur) attende le jeton se réduit, augmentant d'autant le nombre de messages *inter* nécessaire pour transmettre les requêtes (resp. le jeton). Le nombre total de messages passe donc, en moyenne, de 2 à N. Ainsi pour une application **fortement parallélisée** ($\rho \geq 3N$), la composition Naimi-Naimi devient moins coûteuse que la composition Naimi-Martin.

On peut résumer cette étude de la complexité en nombre de messages *inter* des différentes compositions par le tableau suivant (figure 7.13) :

Composition \ Parallélisme	Faible	Moyen	Fort
Naimi-Suzuki	$N+1$	$N+1$	$N+1$
Naimi-Martin	2	$2 < K < N$	$(N/2)+(N/2)$
Naimi-Naimi	$log(N)+1$	$log(N)+1$	$log(N)+1$

FIGURE 7.13 – Complexité moyenne en nombre de messages *inter* pour chaque composition

7.3.3.3 Étude de l'écart type des temps d'obtention



(a) Écarts types des délais moyen d'obtention (σ) (b) Écarts types relatifs des délais moyen d'obtention ($\sigma/\bar{x}$)

FIGURE 7.14 – Écart-type des délais moyen d'obtention de la section critique

Afin d'étudier plus précisément l'effet de la composition sur les latences d'obtention, nous avons mesuré l'écart type du délai d'attente pour tous les accès à la section critique dans l'ensemble de la grille. Ces résultats pour chacune des compositions sont présentés dans la figure 7.14(a).

On peut tout d'abord remarquer que l'écart type reste significatif quelque soit l'algorithme considéré. Cette forte dispersion des mesures traduit bien l'impact de l'hétérogénéité des réseaux sur l'exclusion mutuelle. En effet, quelle que soit la méthode employée pour acheminer requêtes et jeton, il y aura toujours une énorme différence de latence suivant que l'algorithme utilise ou non le réseau "*WAN*" inter-connectant les clusters.

Pour étudier plus avant les variations des temps d'accès à la section critique suivant le degré de parallélisme des applications, nous avons choisi de calculer les écarts types relatifs $\sigma_r = (\sigma/\bar{x})$, c'est-à-dire les ratios des écarts types σ sur les moyennes $\bar{x}$. En effet, la section précédente ayant montré le fort impact du paramètre ρ sur la latence moyenne, il apparaît difficile de comparer les valeurs brutes des écarts types sans les pondérer par le temps d'accès moyen. Les courbes de la figure 7.14(b) présentent l'évolution de ces valeurs suivant ρ.

D'une manière générale on remarque que l'écart type relatif σ_r pour l'algorithme de Naimi à plat est toujours inférieur à celui des compositions. Cela s'explique par le fait

que l'envoi de messages dans l'algorithme à plat est complètement indépendant de la localisation physique. Chaque message a donc autant de chance d'être émis sur le réseau local ou sur le "*WAN*". Nous reviendrons sur cette assertion lors de l'étude des effets du "*clustering*" section 7.4.3. À l'inverse, l'effet de préemption naturelle de la composition induit deux cas bien distincts : des délais très courts si le jeton est déjà présent dans le cluster ou une forte latence si l'on utilise l'algorithme *inter* et donc le réseau "*WAN*".

Si l'on étudie maintenant l'évolution des écarts types relatifs σ_r suivant le paramètre ρ, on remarque que l'ensemble des courbes présente la même allure. Dans un premier temps σ_r augmente avec ρ puis il se stabilise autour d'une valeur seuil. Pour des applications faiblement parallèle, les faibles valeurs de σ_r peuvent s'expliquer par la conjonction de deux phénomènes différents. Le premier résulte du recouvrement du temps d'acheminement des requêtes (T_{req}) par celui passé en file d'attente qui a été étudié dans les sections précédentes. Ce recouvrement a ici pour effet de réduire le nombre des messages induisant de la latence : seul(s) le ou les messages nécessaires à l'envoi du jeton rentrent dans le calcul de la latence. Puisqu'il y a moins de messages pris en compte, les écarts de latence se trouvent réduits. L'autre phénomène agit surtout pour les plus petites valeur de ρ. Comme on l'a vu précédemment, lorsque $\rho = n/2$ un très grand nombre de sites sont en attente de section critique. La conséquence de cette concurrence est une sérialisation des accès à la section critique. En réduisant fortement l'entropie du système, cette sérialisation diminue d'autant l'écart type.

Pour terminer cette étude des écarts types, comparons les résultats pour les différentes compositions. Globalement, pour un degré de parallélisme donné, les écarts types relatifs σ_r restent très similaires quelles que soient les compositions considérées. Ainsi, si l'on fixe ρ, la composition la plus efficace sera aussi la plus équitable.

7.3.4 Choix de l'algorithme *intra*-cluster

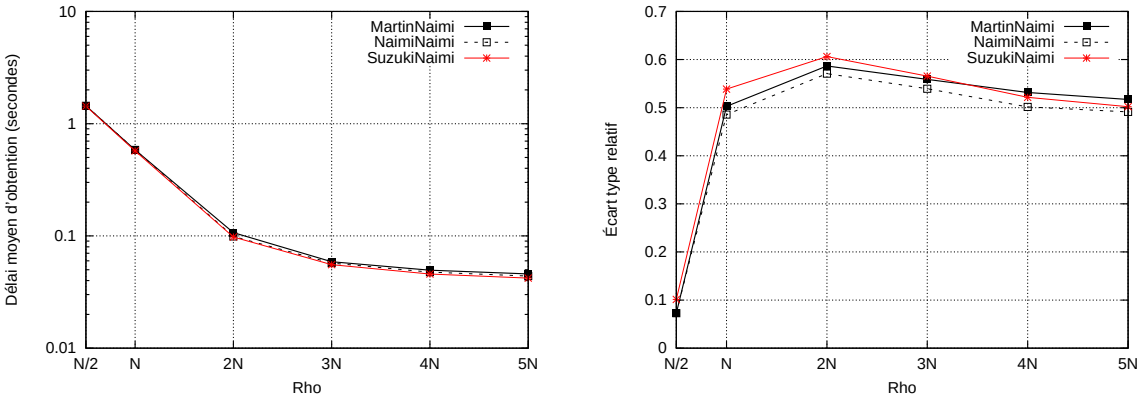
Après avoir étudié l'impact du choix de l'algorithme *inter* sur les performances de la composition, nous allons étudier maintenant celui du choix de l'algorithme *intra*. Pour ce faire, nous avons mené une nouvelle série d'expériences. Ces expériences reprennent le même protocole et la même topologie que dans les sections précédentes. Toutes les compositions testées utilisent Naimi comme algorithme *inter*.

La figure 7.15(a) présente les mesures pour le temps moyen d'obtention de la section critique. Contrairement à ce que nous avions observé pour l'algorithme *inter*, on ne voit ici aucune incidence du choix de l'algorithme *intra* sur la latence. Ceci se comprend facilement si l'on met en rapport les temps de propagation des messages sur le réseau local avec ceux mesurés sur le réseaux WAN inter-connectant les clusters (voir le tableau de la figure 7.3). Les gains dûs à tel ou tel algorithme *intra* deviennent alors négligeables devant les performances de l'algorithme *inter* choisi.

Concernant l'équité, on observe quelques petites différences sur les écarts types présentés dans la figure 7.15(b). L'algorithme de Naimi-Tréhel se montre plus régulier que les deux autres. Ce résultat ne dépend d'ailleurs pas du degré de parallélisme de l'application.

Pour étudier les raisons de ce phénomène, il nous faut considérer le mécanisme de création de la file d'attente dans chacune des compositions.

Dans l'algorithme de Martin, la file est induite par la topologie en anneau. Le jeton



(a) Délai moyen d'obtention

(b) Écarts types relatifs des délais moyen d'obtention ($\sigma/\bar{x}$)

FIGURE 7.15 – Choix de l'algorithme *intra*

parcoure l'ensemble des sites du cluster avant d'en ressortir, assurant ainsi à tous les nœuds applicatifs de pouvoir accéder à la section critique. Mais puisque ce parcours est fixe, il favorise les sites proches du coordinateur (dans le sens du parcours du jeton), diminuant ainsi l'équité de l'algorithme.

L'algorithme de Suzuki-Kasami, lui utilise une queue véhiculée par le jeton. Celle-ci est mise à jour lorsqu'un site, ayant terminé sa section critique, s'apprête à libérer le jeton. Comme expliqué à la section 7.3.1.3, il regarde, séquentiellement pour chacun des autres sites du système, s'il n'a pas enregistré une requête qui n'aurait pas déjà été ajoutée à la file d'attente. Supposons qu'il ait reçu une requête du site 12 puis une autre émise par le site 3. Lors du parcours de la liste des identifiants, la requête 3 sera enregistrée avant celle du site 12. Les sites ayant un petit identifiant sont donc avantagés. Ce phénomène est accentué ici par le temps que passe le coordinateur en section critique. En effet, vu de l'algorithme *intra*, un coordinateur exécute des sections critiques dont la durée correspond à toutes les sections critiques des autres clusters. La probabilité qu'il reçoive des requêtes concurrentes pendant sa section critique est donc beaucoup plus grande que pour les nœuds applicatifs. Notons, pour terminer, que cette iniquité résulte de la concurrence des nœuds applicatifs. Elle ne s'applique donc pas pour des applications **fortement parallélisée**. On observe d'ailleurs que l'écart type mesuré pour cet algorithme rejoint celui de Naimi lorsque $\rho = 5N$.

L'algorithme de Naimi-Tréhel est plus équitable car la file d'attente distribuée, que constitue la chaîne des *NEXT*, se forme dynamiquement dès qu'une requête atteint la racine de l'arbre des *LAST*. L'ordre d'accès à la section critique correspond donc exactement à l'ordre d'émission des requêtes, faisant de Naimi-Tréhel un algorithme extrêmement équitable.

On peut conclure cette étude de la régularité par la remarque suivante. Un bon algorithme *intra* doit s'abstraire de l'asymétrie que produit la présence du coordinateur, ce dernier ayant des sections critiques beaucoup plus longues que celles des autres participants.

7.3.5 Choix des couples de composition

Tout d'abord, nous avons montré que le choix de l'algorithme *intra* a une influence très faible sur les performances globales. En revanche, le choix des algorithmes *inter* change significativement le comportement global. De plus, les expériences nous ont permis de mettre en évidence l'importance du choix d'une bonne composition d'exclusion mutuelle suivant le type du comportement de l'application utilisant le verrou.

Trouver la composition optimal revient alors à faire un compromis entre le *délai d'obtention* du jeton et le *nombre de messages inter-cluster*. Or dans cette étude nous avons considéré un nombre de cluster raisonnable par rapport au nombre de nœuds applicatifs.

Pour des applications **faiblement parallèles** ($\rho < N$), les algorithmes Martin, Naimi-Tréhel et Suzuki-Kasami ont les mêmes *délais d'obtention du jeton*. Cependant, Martin génère beaucoup moins de messages *inter* ce qui le rend le plus intéressant.

Pour des applications **moyennement parallèles** ($N \leq \rho < 3N$), les algorithmes Naimi et Suzuki sont équivalents en termes de *délai d'obtention du jeton*, mais Suzuki génère encore un nombre plus important de messages. D'où notre choix de Naimi comme meilleur algorithme.

Finalement, pour des applications **hautement parallèles** ($\rho \geq 3N$), Suzuki génère beaucoup plus de messages *inter* que les deux autres algorithmes, mais comme il reste le plus rapide pour délivrer le jeton ($2 * T$), il devient le meilleur choix grâce à sa "vivacité".

Remarquons pour finir que ce dernier choix, même s'il n'est pas le plus économe en messages, nous est permis ici par le gain en complexité obtenu grâce à la composition. En effet, il serait difficilement envisageable d'utiliser l'algorithme de Suzuki sur l'ensemble de la grille. Son utilisation dans le cadre de la composition ne pose par contre aucun problème. Comme on peut le voir sur la figure 7.12, la complexité d'un Suzuki-Suzuki est identique à celle d'un Naimi à plat.

On peut résumer ce résultat sur le choix des composition dans le tableau 7.16 suivant :

Parallélisme	Faible	Moyen	Fort
Composition à privilégier	X-Martin	X-Naimi	X-Suzuki

FIGURE 7.16 – Choix des compositions

7.4 Impact de la structure de la grille

Après avoir étudié le choix de la composition en fonction du type d'applications, nous nous intéressons maintenant à l'impact du partitionnement (nombre de clusters) d'une grille sur les algorithmes d'exclusion mutuelle en général et sur notre algorithme de composition en particulier. Pour ce faire, nous avons réalisé des expériences en faisant varier le nombre de clusters de la grille tout en conservant le même nombre de processeurs. Nous avons fixé le nombre de processeurs à 120 et étudions des configurations de 2, 3, 4, 6, 8, 12, 20, 30, 40, 60 et 120 clusters.

7.4.1 Grille de test et protocole d'expérimentation

La grille expérimentale Grid'5000 utilisée lors de l'étude de la composition étant limitée à 9 sites, nous avons émulé les différentes topologies réseaux et la structure de nos grilles sur un cluster possédant un nœud routeur "dummynet" ([Riz97]) utilisé pour injecter des latences entre les clusters virtuels. Le cluster est composé de 24 machines Bi-Xeon 2,8Ghz avec 2Go de RAM et d'une machine BSD exécutant un module dummynet qui est une machine P4 2Ghz spécialement dédiée à l'émulation des latences de communications. L'ensemble de ces machines est connecté par ethernet gigabit via un switch d'une capacité de 148 Gbps. Afin de valider la méthode, nous avons reproduit certaines expériences de la section 7.3. Les résultats obtenus se sont montrés quasiment identiques à ceux mesurés sur Grid'5000.

Comme précédemment et pour les mêmes raisons, les expériences présentées ici portent sur l'algorithme de Naimi-Tréhel (noté "*A plat*" sur les graphiques). Ses performances sont comparées à celles de la composition Naimi-Naimi (notée "*composition*" sur les graphiques). Cette dernière s'est en effet montrée la plus polyvalente, avec un délai moyen d'obtention optimum pour les applications faiblement et moyennement parallèles, ainsi qu'une bonne latence pour les applications ayant un fort degré de parallélisme (voir section 7.3.5). De plus, sa faible complexité en messages permettra d'étudier le cas extrême où la grille est composée de 120 clusters d'une seule machine.

La section 7.3 ayant montré le fort impact du niveau de parallélisme sur les performances, nous avons choisi d'intégrer ce paramètre dans notre étude des effets du clustering. Les figures 7.17(a) et 7.17(d) correspondent à une application dont le niveau de parallélisme est relativement faible ($\rho = N/2$). Les figures 7.17(b) et 7.17(e) représentent les résultats pour une application moyennement parallèle ($\rho = 2N$) tandis que les figures 7.17(c) et 7.17(f) ont été réalisées avec une application hautement parallèle ($\rho = 5N$).

Pour chaque expérience, nous avons mesuré : le temps moyen d'attente pour obtenir une section critique en secondes (figures 7.17(a), 7.17(b) et 7.17(c)) et le nombre total de messages *inter* (figures 7.17(d), 7.17(e) et 7.17(f)).

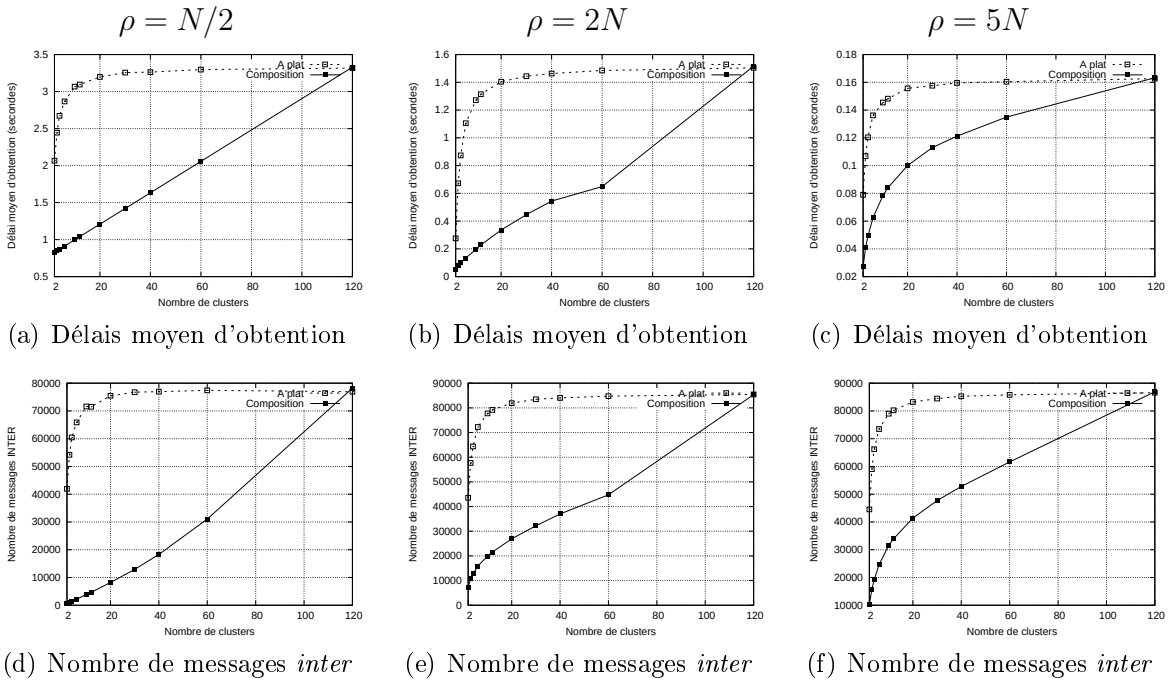
Dans cette série d'expériences, chaque nœud exécute 100 sections critiques. Nous effectuons les mesures en régime stationnaire. Ainsi les 500 premiers accès et les 500 derniers ne sont pas mesurés. Les durées des sections critiques suivent une distribution de Poisson dont la moyenne est fixée $10ms$. De même, les délais entre deux requêtes suivent une distribution de Poisson dont la moyenne est fixée par le paramètre ρ.

7.4.2 Impact sur l'algorithme à plat

Commençons par étudier l'impact de la structure de la grille sur l'algorithme de Naimi-Tréhel à plat.

Si l'on compare globalement pour cet algorithme, dans les figures 7.17(a), 7.17(b) et 7.17(c), le délai moyen d'obtention en fonction du nombre de clusters, on remarque une similarité dans l'allure de ces courbes. On remarque de même dans les figures 7.17(d), 7.17(e) et 7.17(f) un comportement très similaire pour l'ensemble des courbes mesurant le nombre de messages *inter* émis par l'algorithme de Naimi-Tréhel global.

Ainsi, pour toutes ces mesures, on observe une très rapide augmentation lorsque le



(a) Délais moyen d'obtention (b) Délais moyen d'obtention (c) Délais moyen d'obtention

(d) Nombre de messages *inter* (e) Nombre de messages *inter* (f) Nombre de messages *inter*

FIGURE 7.17 – Impact du nombre de clusters sur la composition

nombre de clusters varie de 2 à 12, et ce, quelle que soit la valeur de ρ. Lorsque le partitionnement s'accroît davantage, la croissance des mesures diminue fortement pour devenir négligeable passé 40 clusters.

Pour expliquer cette allure hyperbolique des courbes, il nous faut étudier la fréquence d'utilisation du réseau WAN par un algorithme d'exclusion mutuelle global et donc la probabilité $\mathcal{P}$ que le destinataire d'un message n'appartienne pas au même cluster que l'émetteur. Considérons pour cela une grille de n nœuds se répartissant uniformément dans c clusters. Pour simplifier les calculs, on supposera qu'un site peut s'envoyer un message à lui même. Cette hypothèse modélise 2 accès successifs à la section critique par le même site. On obtient alors la probabilité suivante :

$$\mathcal{P}_{plat} = \frac{n - \frac{n}{c}}{n} = 1 - \frac{1}{c}$$

Cette formule explique parfaitement l'allure des courbes de la figure 7.17. Elle permet aussi de montrer que ce résultat ne dépend pas du nombre de machines si elles sont uniformément réparties sur la grille.

Enfin, de façon plus fine, ce calcul montre que l'effet de bord dû à l'hétérogénéité des latences de communication sur l'algorithme de Naimi-Tréhel à plat, s'il existe, est marginal. En effet, même si l'algorithme est déployé globalement sans tenir compte de la topologie, on était en droit de se demander si la différence entre les latences ne favorisait pas les sites proches. On aurait pu voir un ré-ordonnancement de l'arbre des *LAST* suivant la topologie physique. Or dans le calcul précédent, le choix du destinataire parmi l'ensemble des sites se fait de façon équiprobable (sans tenir compte de la topologie) et puisque l'allure de la courbe théorique ainsi obtenue est identique à celle mesurée, on

peut déduire que l'hypothèse d'équiprobabilité est raisonnable et donc que l'impact d'une "composition naturelle" dans l'algorithme global est négligeable.

Étudions maintenant l'impact de la topologie sur l'algorithme de Naimi-Tréhel en fonction du type d'application. En comparant les mesures de l'algorithme de Naimi-Tréhel dans les figures 7.17(a), 7.17(b) et 7.17(c), on retrouve ce que l'on avait déjà observé sur GRID'5000 et expliqué à la section 7.2.4, à savoir que le degré de parallélisme des applications a une incidence sur le délai moyen d'obtention de la section critique. Mais puisque ces courbes gardent la même allure dans les trois figures, cet impact reste constant quel que soit le niveau de partitionnement dans la grille.

À l'inverse, les courbes des figures 7.17(d), 7.17(e) et 7.17(f) mesurant le nombre de messages *inter* émis montrent que le parallélisme n'a presque pas d'impact sur la complexité. On observe néanmoins une légère diminution du nombre de messages dans le cas d'un algorithme faiblement parallélisé.

Après avoir étudié le surcoût engendré par le partitionnement de la grille dans un algorithme à plat, étudions maintenant l'impact de la topologie sur notre composition.

D'une manière générale, on peut voir sur l'ensemble des courbes de la figure 7.17 que le partitionnement a aussi un impact sur l'algorithme de composition : la latence et le nombre de messages augmentent avec le nombre de clusters.

Mais si l'on exclut le cas particulier d'une grille composée de 120 clusters constitués d'un seul nœud applicatif, où il n'y a pas vraiment de composition, notre approche reste toujours plus rapide et moins coûteuse en messages *inter* que l'algorithme de Naimi-Tréhel à plat. Du reste, ces gains demeurent encore très significatifs pour une grille composée de 60 clusters de deux nœuds applicatifs.

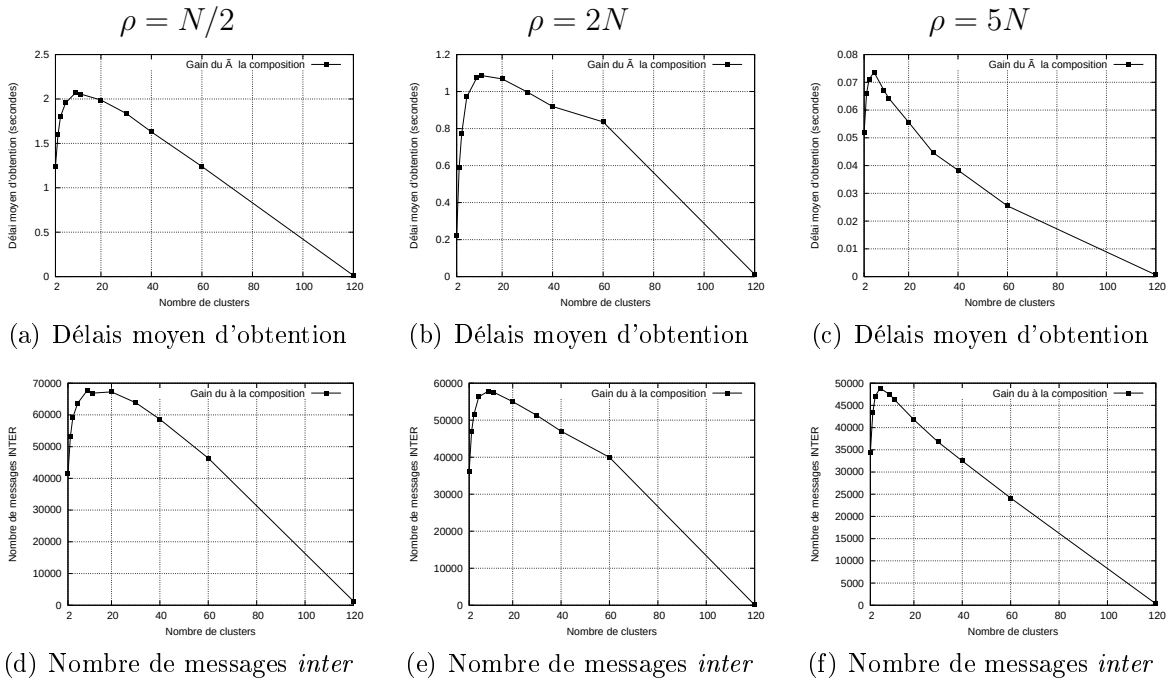
D'autre part, ces augmentations sont beaucoup moins brutales que pour "Naimi" : l'approche par composition permet de résister bien plus longtemps au partitionnement. Ainsi l'écart de croissance est très marquée pour les premières valeurs des courbes.

7.4.3 Impact sur l'approche par composition

Notre approche ne réagissant pas de la même manière au partitionnement que l'algorithme global, il peut être intéressant d'étudier la variation des gains, tant en termes de latence qu'en nombre de messages. Pour ce faire, étudions les courbes des figures 7.18(a), 7.18(b) et 7.18(c) (resp. 7.18(d), 7.18(e) et 7.18(f)) qui mesurent pour chaque topologie l'écart du délai d'obtention (resp. du nombre de messages) entre les deux approches.

Sur ces courbes, on remarque que globalement le gain dû à la composition augmente lorsque le partitionnement varie entre 2 et 12 clusters. Comme on l'a vu dans la sous-section précédente, pour ces petites valeurs le surcoût dû à la topologie augmente très rapidement si l'on utilise un l'algorithme de Naimi-Tréhel à plat. Or, sur la même plage de valeurs, notre approche résiste beaucoup mieux. Ceci explique qu'un écart maximal entre les deux courbes soit atteint lorsque les nœuds du système se répartissent en 12 clusters. Une fois passé ce *seuil*, le partitionnement n'engendre plus de surcoût pour une utilisation à plat. L'écart se réduit alors progressivement, pour finir par être nul lorsque chaque site représente un cluster. Il n'y a alors plus vraiment de composition.

Essayons de calculer ce partitionnement *seuil*, noté c_{seuil}, pour lequel la composition apporte un gain maximum. Nous considérons un système regroupant n sites applicatifs.



(a) Délais moyen d'obtention (b) Délais moyen d'obtention (c) Délais moyen d'obtention

(d) Nombre de messages *inter* (e) Nombre de messages *inter* (f) Nombre de messages *inter*

FIGURE 7.18 – Impact du partitionnement de la grille sur la composition

Nous y re-calculons, pour notre approche par composition, la probabilité $\mathcal{P}$ d'utilisation du réseau WAN précédemment calculée pour l'algorithme à plat. Afin de simplifier les calculs, on se place dans le cas d'une préemption maximum, c'est à dire qu'à chaque réception du jeton *inter* par un coordinateur, tous les sites de son cluster effectuent une section critique. $\mathcal{P}_{compo}$ est donc égale à la probabilité de considérer la dernière section critique avant que le jeton ne ressorte du cluster :

$$\mathcal{P}_{compo} = \frac{1}{\frac{n}{c}} = \frac{c}{n}$$

On peut maintenant calculer l'écart $E(c)$ entre les deux approches en fonction du partitionnement :

$$E(c) = \mathcal{P}_{plat} - \mathcal{P}_{compo} = 1 - \frac{1}{c} - \frac{c}{n}$$

On en déduit la valeur c_{seuil} du partitionnement seuil :

$$E'(c) = \frac{1}{c^2} - \frac{1}{n} \Rightarrow c_{seuil} = \sqrt{n}$$

Le gain engendré par l'utilisation de notre approche par composition est donc maximum pour une grille comprenant $\sqrt{n}$ clusters. Ceci se vérifie bien sur les courbes puisque $\sqrt{120} = 10,95$.

Comparons maintenant les résultats obtenus pour les différents types d'applications. Contrairement à ce que nous avions observé pour l'algorithme de Naimi-Tréhel à plat, on voit ici une nette différence entre les allures des courbes suivant le niveau du parallélisme.

En effet, la structure hiérarchique de composition a un effet concentrateur. Cet effet, lié au nombre de nœuds applicatifs par cluster, se démultiplie avec l'augmentation de la concurrence d'accès à la section critique.

Ainsi, pour une application faiblement ou moyennement parallèle ($\rho = N/2$ et $\rho = 2N$), l'effet concentrateur de la composition est à son maximum, et ce quel que soit le nombre de clusters. La latence et le nombre de messages deviennent alors proportionnels au taux de partitionnement, ce qui explique l'allure quasi linéaire des courbes des figures 7.17(a) et 7.17(d).

À contrario, pour une application fortement paralléle ($\rho = 5N$), l'effet de concentration de la composition est plus faible et dépend directement du taux de partitionnement : plus le nombre de nœuds applicatifs est petit, plus la probabilité d'avoir une autre requête dans son cluster local est faible.

7.4.4 Généralisation pour une répartition hétérogène des machines

L'étude précédente faisait l'hypothèse d'une répartition homogène des machines dans chacun des clusters de la grille. Elle a, entre autres, montré que dans ces conditions le surcoût dû au partitionnement des machines est très important et qu'il reste significatif pour un nombre restreint de clusters. On peut alors se demander ce qu'il advient de ces résultats avec une répartition hétérogène des nœuds.

Devant la durée des expériences et le nombre de topologies à tester, il paraît difficile d'étudier expérimentalement l'impact d'une grille hétérogène. Nous allons donc étudier théoriquement le surcoût qu'elle engendre. Pour ce faire, on considère une grille de n nœuds répartis dans c clusters, de telle manière qu'il y ait m_i machine(s) dans le cluster i. La probabilité d'utiliser le réseau WAN, calculée précédemment, ne dépend plus directement du nombre de clusters (c), mais est fonction de la répartition (noté M) des nœuds dans ces c clusters. On notera $\mathcal{M}_c$ l'ensemble de ces répartitions :

$$\mathcal{P}(M) = \sum_{i=1}^{c} \left(\frac{m_i}{n}\right)\left(\frac{n-m_i}{n}\right) = \frac{1}{n^2}\sum_{i=1}^{c}(m_i n) - \frac{1}{n^2}\sum_{i=1}^{c} m_i^2 = 1 - \frac{\sum_{i=1}^{c} m_i^2}{(\sum_{i=1}^{c} m_i)^2}$$

La probabilité qu'un message soit émis sur le réseau inter-connectant les clusters est donc proportionnelle à une somme de carrés sur le carré de la somme. Cette probabilité

peut alors être majorée grâce à l'inégalité de Cauchy-Schwarz ([Sch88]) :

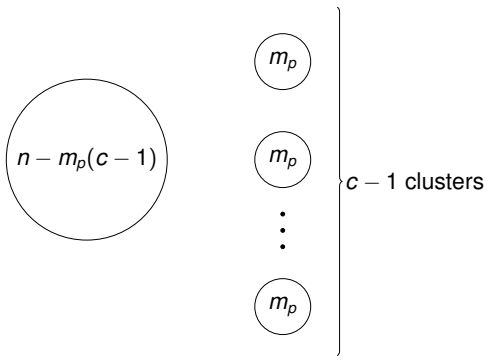
$$\begin{aligned}
\text{Inégalité de Cauchy-Schwarz} \quad &\Rightarrow \quad \left(\sum_{i=1}^{c} m_i \times 1\right)^2 \leq \left(\sum_{i=1}^{c} m_i^2\right)\left(\sum_{i=1}^{c} 1^2\right) \\
&\Rightarrow \quad \left(\sum_{i=1}^{c} m_i\right)^2 \leq c\sum_{i=1}^{c} m_i^2 \\
&\Rightarrow \quad \frac{1}{c} \leq \frac{\sum_{i=1}^{c} m_i^2}{(\sum_{i=1}^{c} m_i)^2} \\
&\Rightarrow \quad \forall c,\ \forall M \in \mathcal{M}_c,\ \mathcal{P}(M) \leq 1 - \frac{1}{c}
\end{aligned}$$

On peut donc, pour l'ensemble des configurations $\mathcal{M}_c$ contenant c clusters, majorer la probabilité d'émission sur le réseau WAN par la constante $1 - 1/c$. Or cette valeur est atteinte lorsque l'on considère une configuration homogène $H_c \in \mathcal{M}_c$ (voir section 7.4.3). On a donc la borne suivante :

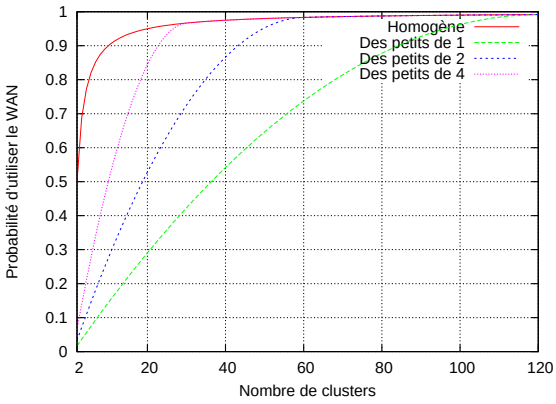
$$\forall c,\ \forall M \in \mathcal{M}_c,\ \mathcal{P}(M) \leq \mathcal{P}\left(H_c\right)$$

Les configurations homogènes étudiées dans les expériences de la section 7.4 constituaient donc le pire cas. Il est alors intéressant d'étudier les variations de cette probabilité $\mathcal{P}(M)$ suivant différentes répartitions hétérogènes (pour un nombre de clusters donné). Le but de cette étude est de faire ressortir la vitesse de convergence des surcoûts dû à l'utilisation d'une grille vers leur maximum (répartition homogène). Autrement dit, d'étudier l'exemplarité des résultats obtenus sur une répartition homogène. Dans un premier temps nous considérons des grilles composées d'un unique grand cluster accompagné de k petits clusters de taille équivalente m_p. Le nombre de machines restant fixe, la taille m_g du gros cluster dépend donc du nombre de machines restantes après avoir placé m_p machines dans chacun des $c - 1$ petits clusters : $m_g = n - m_p(c - 1)$. La figure 7.19(b) présente la probabilité d'émettre sur le réseau WAN pour quatre de ces répartitions avec : $m_p = 1$, $m_p = 2$, $m_p = 4$ et $m_p = n/c$ (répartition homogène). Ces courbes caractérisent l'influence de la taille minimum des clusters sur les performances des algorithmes. On y remarque que dans le cas extrême de $n - 1$ clusters d'une seule machine, le surcoût augmente presque linéairement en fonction du nombre de clusters. En revanche, si l'on augmente un tout petit peu la taille des petits clusters cette augmentation est très rapide.

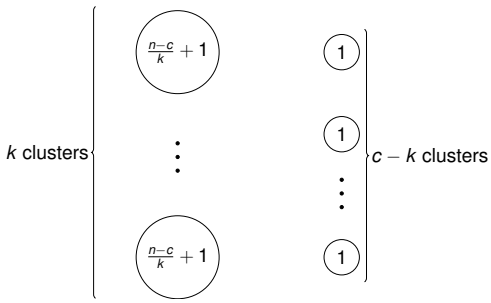
Dans les topologies que nous venons de considérer, il n'y avait qu'un gros cluster. On peut alors se demander ce qu'il advient du surcoût en présence de plusieurs gros clusters. On considère donc maintenant des grilles composées de k gros clusters de taille équivalente, tous les autres clusters étant constitués d'une unique machine ($m_p = 1$). La taille m_g de ces gros clusters diminue naturellement avec leur nombre : $m_g = (n-c)/k+1$. La figure 7.19(d) présente la probabilité d'émettre sur le réseau WAN pour quatre de ces répartitions : $k = 1$, $k = 2$, $k = 4$ et $k = c$ (répartition homogène). Là encore, ces courbes nous montrent qu'en augmentant la proportion des gros clusters on tend très rapidement vers les valeurs obtenues avec une répartition homogène. On voit par exemple qu'avec 4 gros clusters sur 20, la probabilité d'émettre un message sur le réseau WAN est de 80%.



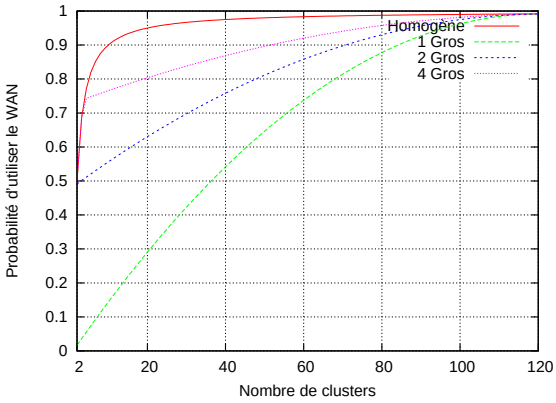
(a) $c-1$ petits clusters de m_p machines



(b) Impact des clusters de plus petite taille



(c) k gros cluster et $c-k$ machines isolées



(d) Impact des clusters de plus grande taille

FIGURE 7.19 – Surcoût par une grille suivant la répartition de ses machines

En conclusion, ces deux études théoriques nous montrent que si l'on s'écarte du cas extrême d'une grille composée d'un gros cluster et de $n-1$ clusters d'une seule machine individuelle, on tend rapidement à retrouver le surcoût étudié précédemment pour une répartition homogène.

7.5 Hétérogénéité de la concurrence

Lors des évaluations de performances, nous avons toujours considéré une concurrence d'accès à la section critique homogène. Dans nos expériences, elle était caractérisée par le paramètre ρ. Or même si l'utilisation d'une distribution de Poisson pour simuler les fréquences d'accès à la section critique est une métrique reconnue pour évaluer les performances d'algorithme d'exclusion mutuelle, elle peut masquer certains effets de bord qui pourraient apparaître avec telle ou telle application réelle. En effet, le nombre de endframe requêtes concurrentes peut être amené à varier fortement :

- temporellement : si l'utilisation de l'exclusion mutuelle n'est pas uniforme sur la durée de l'application.
- spatialement : si la fréquence d'utilisation diffère suivant les clusters.
- temporellement et spatialement : charge ponctuelle dans un cluster donné.

Dans cette section nous allons étudier l'impact de ces différents types d'hétérogénéités sur notre algorithme de composition et, le cas échéant, chercher à en améliorer les performances en lui permettant de s'adapter à de tels environnements.

7.5.1 Hétérogénéité temporelle : Composition dynamique

Notre approche permet de choisir la composition en fonction de l'application. Pour l'instant, ce choix se fait a priori, avant le lancement de celle-ci. Il pourrait alors être intéressant de pouvoir modifier ce choix dynamiquement en fonction de la concurrence d'accès à la section critique. Il s'agirait surtout ici de changer le type de l'algorithme *inter*.

Une façon simple de modifier cet algorithme serait de faire un consensus entre les coordinateurs de la grille. Une fois l'ensemble de ces sites d'accord, on pourrait initialiser une nouvelle instance d'algorithme *inter*. Reste encore à gérer les requêtes pendantes au moment du consensus. Deux approches semblent possibles : soit l'on conserve momentanément l'ancienne file d'attente, soit l'ensemble des coordinateurs attendant le jeton *inter* ré-émettent leur requête dans le nouvel algorithme.

Si les changements de comportement de l'application ne sont pas trop fréquents, le surcoût dû à la modification de l'algorithme *inter* devrait être acceptable. De plus, l'utilisation de cycles d'hystérésis pourra permettre d'éviter les phénomènes d'aller-retour autour des valeurs seuils.

Il est aussi à remarquer que l'on peut profiter du temps d'accès des coordinateurs à la section critique *inter* pour recouvrir la latence induite par le changement d'algorithme. En effet, en raison de la préemption *intra*, le temps que passe un coordinateur en section critique correspond souvent à de multiples accès *intra* par les nœuds applicatifs.

Se pose enfin la question du mécanisme déclenchant le changement d'algorithme. Dans un premier temps, on pourrait ajouter une interface à notre algorithme de composition

pour contrôler son évolution et forcer le changement d'algorithme *inter*. Il reviendrait alors à l'application d'ajuster la composition en fonction de sa charge. Une telle approche présente certes l'avantage de pouvoir anticiper les pics de charges, mais elle demande une modification des applications, ce qui n'est pas toujours envisageable.

Une autre approche, plus transparente pour l'application, serait d'intégrer directement dans l'algorithme des coordinateurs un mécanisme contrôlant le changement d'algorithme *inter*. Ce mécanisme pourrait exploiter la connaissance de la charge de chaque instance de l'algorithme *intra*. Cependant, cette approche limite le choix de l'algorithme *intra* aux algorithmes permettant aux coordinateurs d'avoir la connaissance de l'ensemble des requêtes pendantes de son cluster. Les algorithmes de Naimi-Tréhel et de Martin sont donc ici à écarter, puisque ces deux algorithmes ne garantissent pas à un nœud de recevoir l'ensemble des requêtes du système. À l'inverse, l'algorithme de Suzuki-Kasami qui repose sur un mécanisme de diffusion est quant à lui parfaitement adapté à notre problème car toutes les requêtes *intra* seraient envoyées au coordinateur. Cette solution a aussi un surcoût en nombre de messages échangés. En effet, pour avoir une vision globale de la charge du système il faut ajouter un échange d'information entre les coordinateurs.

7.5.2 Hétérogénéité spatiale : Ajouter une préemption *inter*

Si elle peut varier dans le temps, la fréquence des requêtes peut aussi varier géographiquement suivant les clusters. Cette hétérogénéité spatiale trouve son explication dans deux phénomènes parfois complémentaires. Le premier, matériel, résulte d'une différence entre les puissances de calculs des différents clusters. En effet, les machines équipant chaque cluster peuvent différer tant en nombre qu'en puissance. La fréquence des demandes d'accès à la section critique est alors plus élevée dans les clusters ayant une plus grande puissance de calcul. Le deuxième phénomène pouvant conduire à une hétérogénéité spatiale est lui d'ordre logiciel. En effet, la répartition de l'application dans la grille peut ne pas être uniforme. Si tel est le cas, la fréquence d'exécution de code concurrent peut être amenée à varier d'un cluster à l'autre.

Du point de vue de la composition, l'hétérogénéité spatiale pose un nouveau problème car notre structure hiérarchique masque totalement les requêtes *intra* d'un cluster aux coordinateurs des autres clusters. Ce qui présentait un avantage en termes de complexité (agrégation des requêtes *intra*), devient dans ce cadre un inconvénient en termes d'équité et de performance. En effet, du point de vue de l'algorithme *inter* toutes les requêtes des coordinateurs sont équivalentes. Ainsi, tous les coordinateurs sont servis de la même façon, et ce quel que soit leur nombre de requêtes *intra* pendantes. Les sites applicatifs d'un cluster, ayant une fréquence de requêtes plus élevée, seront donc pénalisés par notre mécanisme. Ceci est d'autant plus gênant si ces mêmes sites représentent une des forces de calcul les plus importantes.

Pour remédier à cette carence, il nous faudrait donc tenir compte du nombre des requêtes *intra* pendantes dans chaque cluster afin d'optimiser la circulation des différents jetons. Ceci nécessite de pouvoir récupérer cette information au niveau des coordinateurs. La solution passe alors par l'emploi d'un algorithme à priorité dynamique comme algorithme *inter*. Chaque coordinateur doit alors pouvoir augmenter sa priorité en fonction du nombre de requêtes *intra* pendantes dans son cluster.

On peut facilement construire un tel algorithme en partant de celui de Suzuki-Kasami et en ajoutant la possibilité à un participant d'émettre plusieurs requêtes. Comme dans l'algorithme original, chaque requête incrémente le compteur RN et est diffusé à l'ensemble du système. Un coordinateur, sortant de section critique, peut alors tenir compte du nombre de requêtes pendantes dans chacun des autres clusters, en donnant la priorité au cluster i ayant la plus grande différence : $RN[i] - LR[i]$. Si l'on veut assurer la propriété d'équité faible, on doit ajouter dans ce mécanisme de préemption *inter*, un compteur de préemption, lequel sera pris en compte dans l'algorithme d'ordonnancement pour éviter toute famine.

Cette solution nécessite tout de même que les coordinateurs émettent une requête *inter* par requête *intra*. L'amélioration des performances, tant en termes d'équité que de délai moyen d'obtention se fait au prix d'une disparition de l'effet agrégation des requêtes *intra*. Une solution intermédiaire serait de limiter le nombre des requêtes *inter* émises à une pour k requêtes *intra*. Le paramètre k contrôle le taux d'agrégation et doit être un compromis entre la complexité en nombre de messages *inter* et la latence d'accès à la section critique.

7.5.3 Charge ponctuelle dans un cluster : Renforcement de la préemption *intra*

Le troisième cas d'hétérogénéité à traiter est celui d'une augmentation brutale de la fréquence d'utilisation de l'exclusion mutuelle dans un cluster donné. Un tel comportement de l'application peut apparaître lors de synchronisation des nœuds applicatifs d'un cluster. Du point de vue de la composition, il pose un nouveau problème, celui de l'efficacité de la *préemption intra*. Cet effet naturel de la composition assure à l'ensemble des requêtes pendantes d'un cluster au moment où le coordinateur récupère le jeton *inter* d'être satisfaites avant que ce dernier ne le relâche. En revanche, il ne garantit rien aux requêtes émises après cette arrivée.

En régime stationnaire la probabilité d'avoir de telles requêtes reste très restreinte, mais dans le cas d'une augmentation brutale de la charge dans un cluster, il peut rester un nombre conséquent de requêtes pendantes lorsque le coordinateur relâche le jeton *inter*. Pour résoudre ce problème, on peut envisager deux solutions.

La première ne nécessite pas de modification de notre algorithme de composition. L'idée est d'utiliser comme algorithme *intra* un algorithme à priorité. En affectant aux coordinateurs une priorité inférieure à celle des nœuds applicatifs, on favorise ces derniers, limitant ainsi dans un cluster le nombre de requêtes non traitées au moment de la libération du jeton *inter*. Cette solution présente l'avantage de ne pas modifier l'algorithme de composition : sa preuve reste donc valide. Cependant elle limite le choix des algorithmes *intra* aux algorithmes à priorité. De plus, le degré de préemption *intra* supplémentaire dépend de l'algorithme choisi, ce qui rend plus difficile son ajustement.

Une deuxième solution, consiste à ajouter un mécanisme de "préemption" dans l'algorithme de composition. Ce mécanisme, que l'on pourrait qualifier d'*artificiel* viendrait suppléer à la *préemption naturelle* le cas échéant. L'algorithme 3 présente le pseudo-code des coordinateurs ainsi modifié : ajout des lignes 151 à 154 à l'algorithme des coordina-

```
135 Coordinator Algorithm ()
136   /* Initially, it holds the intra-token */
137   while TRUE do
138     if ¬ intra.PendingRequest() then
139       state ← OUT
140       Wait for intra.PendingRequest()
141     state ← WAIT_FOR_IN
142     inter.CS_Request()
143     /* Holds inter-token. CS */
144     intra.CS_Release()
145     if ¬ inter.PendingRequest() then
146       state ← IN
147       Wait for inter.PendingRequest()
148     state ← WAIT_FOR_OUT
149     intra.CS_Request()
150     /* Holds intra-token CS */
151     nbPreempt ← 0
152     while IntraPendingRequest ∧ nbPreempt < nbMaxPreempt
        do
153       IntraCSRelease()
154       nbPreempt ← nbPreempt + 1
155       IntraCSRequest()
156       /* Hold Intra CS */
157     inter.CS_Release()
```

Algorithm 3: Algorithme du coordinateur avec préemption

teurs (Algorithme 1). Le principe est d'utiliser la fonction testant la présence de requête pendante (algorithme 3, ligne 152) avant de relâcher le jeton *inter* (ligne 157). S'il reste des demandes à satisfaire le coordinateur relâche le jeton *intra*, puis émet une requête pour le récupérer (ligne 153 et 155).

Pour être efficace, cette méthode nécessite des algorithmes *intra* vérifiant l'*équité forte*. Dans ce cas, cette nouvelle requête étant postérieure à l'ensemble des requêtes applicatives pendantes, ces dernières seront satisfaites avant que le coordinateur ne récupère le jeton *intra*. Pour éviter tout risque de famine, et donc garantir l'équité faible, on limite le nombre de préemptions à l'aide du compteur *nbPreempt*.

Pour terminer, on peut remarquer que l'ajout de ce mécanisme venant renforcer la préemption se fait sans ajout de contrainte sur les algorithmes composés. Il devient, de plus, complètement transparent pour l'algorithme de composition en l'absence de requête *intra* pendante. Son utilisation systématique est donc envisageable. Ceci éviterait d'ailleurs d'avoir à analyser l'application pour choisir de le mettre en œuvre.

7.6 Conclusion

Dans ce chapitre, nous avons présenté un algorithme permettant de composer facilement différents algorithmes d'exclusion mutuelle afin de faciliter le passage à l'échelle dans les grilles de calculs. Cette approche permet de prendre en compte l'hétérogénéité des latences de communications : des communications locales de type LAN ayant une latence de l'ordre d'une dizaine de microsecondes et des communications de type WAN - pour les communications *inter* sites sur de longues distances - de l'ordre de la dizaine de millisecondes. Une telle composition reste complètement transparente pour l'application et pour la plupart des algorithmes d'exclusion mutuelle classiques.

Du point de vue des performances, les résultats de nos expériences conduites sur une grille réelle ont montré que notre approche réduit significativement le temps moyen d'attente d'une section critique et le nombre de messages émis. Ces bons résultats proviennent d'un ensemble d'effets "naturels", à savoir : le filtrage des requêtes, agrégation des requêtes *intra*, agrégation des requêtes *inter* et préemption dans les clusters.

Ces expériences ont aussi montré que le degré de parallélisme de l'application utilisant l'exclusion mutuelle a un impact significatif sur les performances des compositions. Or, l'aspect générique de notre approche hiérarchique permet d'envisager une multitude de compositions. Cela est d'ailleurs facilité par la preuve générique apportée à la section 6.6 qui a montré que toute composition d'algorithme d'exclusion mutuelle vérifie les propriétés de vivacité et de sûreté de l'exclusion mutuelle.

Ce libre choix dans les compositions peut donc permettre, entre autres, une mise en œuvre adaptée à chaque type de d'application. Ainsi, les expériences que nous avons menées ont montré qu'il était intéressant, en présence d'un nombre élevé de requêtes concourantes, d'utiliser comme algorithme *inter* un algorithme basé sur une topologie en anneau. Avec une moindre concurrence, l'algorithme de Naimi-Tréhel (arbre dynamique) se montre particulièrement bien adapté. Enfin dans le cas d'une application hautement parallèle, un algorithme à diffusion du type Suzuki-Kasami permettra de réduire au strict minimum le temps d'attente. Notons que, dans ce dernier cas, l'utilisation dans un système à large échelle d'un tel algorithme n'est rendue possible que grâce aux bonnes propriétés de la composition.

À l'inverse, d'autres expériences ont permis de mettre en évidence le faible impact du choix de l'algorithme *intra* sur la performance globale des différentes compositions.

Une grande partie de l'efficacité de notre architecture provenant de la mise en correspondance de la topologie physique sur la topologie logique, nous avons étudié ses performances suivant le nombre de clusters distants présents dans la grille. Nous avons, ainsi, émulé différentes topologies. Les résultats ont montré qu'en dehors de certaines topologies au partitionnement extrême (1 à 2 machines par cluster) notre approche permettait de limiter fortement le surcoût par rapport aux performances d'un unique cluster. À l'inverse on a pu observer qu'avec une utilisation classique ("à plat") l'augmentation des délais d'attente était déjà très important avec un très petit nombre de clusters.

Ces expériences ont aussi permis de vérifier que la topologie pour laquelle la composition offre le plus grand gain correspond à une grille composée de N machines réparties en $\sqrt{N}$ clusters.

Pour généraliser ces résultats obtenus avec une répartition homogène des machines,

nous avons montré que ces topologies représentaient une limite rapidement atteinte sitôt qu'on s'écarte du cas extrême d'une grille composée d'un cluster et de plusieurs machines individuelles.

Enfin, nous proposons des solutions pour adapter notre mécanisme à une utilisation hétérogène de l'exclusion mutuelle, tant dans le temps que spatialement suivant l'appartenance à tel ou tel cluster. Ainsi nous avons proposé des mécanismes permettant de modifier dynamiquement la composition, d'ajouter une préemption *inter* et de renforcer la préemption *intra*.

En conclusion, l'algorithme de composition que nous proposons ici permet d'obtenir de réels gains de performance et sa généricité lui permet de s'adapter à différents types d'application. Certains de ces mécanismes peuvent toutefois être améliorés pour tenir compte, encore plus finement, de la topologie ou/et de l'application.

Chapitre 8

Conclusion

Sommaire

8.1 Contributions

Dans cette thèse, nous avons adressé deux problèmes liés à la mise en œuvre des algorithmes d'exclusion mutuelle sur des systèmes répartis à grande échelle, à savoir celui de la gestion des défaillances et de l'adaptation aux environnements de type grille.

Tolérance aux défaillances : Dans une première partie, nous avons présenté un état de l'art sur la tolérance aux défaillances dans les algorithmes distribués d'exclusion mutuelle reposant sur la circulation d'un jeton. De cette étude nous avons pu dégager une synthèse des différents systèmes considérés et des solutions qui ont été proposées.

Partant de cette étude, nous avons proposé un algorithme d'exclusion mutuelle tolérant aux défaillances. Cet algorithme présente des propriétés permettant sa mise en œuvre dans des systèmes distribués à grande échelle, à savoir : une complexité en $\mathcal{O}(log(n))$ pour la circulation des requêtes et du jeton, une surveillance point à point et limitée aux sites en attente de la section critique, une utilisation réduite de la diffusion dans les mécanismes de recouvrement et une limitation dans l'annulation des requêtes pendantes. De plus, notre algorithme offre de bonnes propriétés d'équité, en garantissant à un site enregistré dans la file d'attente de voir son ordre d'arrivée respecté quel que soient le nombre et le type de défaillance.

Enfin, nous avons testé et évalué notre algorithme dans un environnement réel. Ces mesures de performances ont permis de confirmer les bonnes propriétés de notre algorithme. Il s'est, en outre, montré peu sensible aux types d'applications, conservant ainsi de bonnes performances quel que soit leur degré de parallélisme. En ce sens, il se montre donc particulièrement polyvalent. Cette étude de performances a également montré que notre algorithme supportait bien de petites valeurs pour les temporisateurs de détection. Ceci permet de réduire sensiblement les délais de recouvrement.

Adaptation aux topologies de type grilles : Dans une deuxième partie, nous avons proposé un algorithme permettant de composer génériquement les algorithmes d'exclusion mutuelle existants pour les adapter aux contraintes spécifiques des grilles de calcul. Par rapport aux autres approches, qui composent statiquement des algorithmes, notre algorithme se distingue par son aspect générique. Il permet ainsi d'adapter la composition aux besoins de l'application ou aux contraintes d'une topologie particulière. De plus, en ne modifiant pas les algorithmes composés, nous avons pu prouver que toute composition d'algorithmes d'exclusion mutuelle vérifiait bien les propriétés de sûreté et de vivacité.

Les compositions, ainsi obtenues, permettent non seulement d'optimiser les délais d'accès à la section critique mais aussi de limiter la complexité en termes de nombre de messages. Le temps moyen d'obtention de la section critique se trouve également réduit par une préemption locale au niveau de chaque cluster. Celle-ci permet un ré-ordonnancement des accès à la section critique de façon à diminuer le nombre de transferts sur le réseau d'inter-connexion des clusters. Contrairement à d'autres approches de composition cette préemption n'est pas obtenue par un mécanisme ad hoc mais se fait naturellement en exploitant les propriétés des algorithmes composés. Le nombre de messages moyen pour chaque accès à la section critique se trouve réduit par des effets d'agrégation induits par le mécanisme de composition. Cette factorisation des requêtes permet à notre approche de conserver de bonnes performances en présence d'un grand nombre de requêtes concurrentes.

Cette approche a été testée et évaluée sur la grille expérimentale *GRID'5000*. Ces expériences ont non seulement validé les bonnes propriétés de notre algorithme, mais elles ont aussi permis de comparer l'efficacité de plusieurs compositions. Il nous a été ainsi possible de proposer des choix de composition en fonction du degré de parallélisme des applications.

Dans une deuxième série d'expériences, nous avons émulé plusieurs topologies de grille et nous avons comparé l'efficacité de la composition suivant différentes répartitions des machines dans la grille. Ces résultats ont montré que les gains offerts étaient déjà intéressants avec une grille composée de deux clusters. Par ailleurs nous avons pu démontrer que la composition offrait des gains optimum pour une grille répartissant n machines dans $\sqrt{n}$ clusters.

8.2 Perspectives

Tolérance aux défaillances : L'algorithme tolérant aux défaillances, proposé dans ce manuscrit, considère un système statique. Il pourrait alors être intéressant d'intégrer un mécanisme permettant l'arrivée ou le départ volontaire de nœuds dans le système. Outre les problèmes de gestion de groupe, cette dynamicité change le nombre de défaillances possibles. Si dans un système statique le nombre de pannes est structurellement limité à $n - 1$ défaillances, leur nombre est potentiellement infini dans un système dynamique. La gestion de cette nouvelle contrainte pourrait mettre en valeur les bonnes propriétés d'équité de notre algorithme. En effet en évitant de réinitialiser le système lors des recouvrements, notre algorithme garantit à tout nœud ayant obtenu une position dans la file d'attente d'accéder en un temps fini à la section critique quel que soit le nombre de

défaillances.

D'autre part, il peut être également intéressant de reprendre les idées de notre algorithme pour résoudre le problème de l'exclusion mutuelle *avec annulation volontaire de requête* (*abortable mutual exclusion*). Si ce problème a été de nombreuses fois traité dans les systèmes à mémoire partagée, il l'est beaucoup moins pour les systèmes distribués. Or nos mécanismes d'acquittement et de gestion des prédécesseurs permettent d'obtenir une file d'attente doublement chainée, qui se prêterait parfaitement au départ volontaire d'un nœud. Notre algorithme pourrait donc être facilement adapté pour traiter ce problème.

Enfin, la partie de notre algorithme permettant de reconstruire la file des requêtes pourrait être ré-utilisée hors du contexte de l'exclusion mutuelle pour fiabiliser une file répartie.

Adaptation aux topologies de type grilles : Cette partie de nos travaux propose un algorithme hiérarchique à deux niveaux. Or certains grands clusters peuvent représenter à eux seuls des systèmes à large échelle. Il peut alors être intéressant d'augmenter le niveau de hiérarchisation. Notre algorithme se prête très bien a ce type de construction. En effet, contrairement aux approche statiques, il permet de composer n'importe quel algorithme d'exclusion mutuelle et a fortiori des algorithmes déjà composés. Si ces nouvelles constructions sont directement réalisables, elles demandent toutefois à être testées pour en mesurer les gains sur les performances. Notons que si le découpage des machines d'un cluster en groupe logique n'est pas dicté par des contraintes liées au réseau local, il pourra être basé sur notre résultat de répartition optimum : $\sqrt{n}$ groupes logiques par cluster.

D'autre part, nous avons ici considéré la gestion d'un unique verrou distribué. Il pourrait alors être intéressant de construire un algorithme hiérarchique optimisant la circulation de plusieurs verrous dans la grille. Des mécanismes de préemption locale pourrait permettre de regrouper naturellement les verrous pour traiter efficacement des requêtes multiples issues d'un même nœud. Ce nouvel algorithme pourrait alors être vu comme un ordonnanceur multiple d'exclusion mutuelle.

Algorithme tolérant aux défaillances pour les grilles : Cette thèse adresse deux problèmes différents : la tolérance aux fautes et l'adaptation aux grilles. L'idée de faire un algorithme tolérant aux défaillances pour grille est alors assez naturelle. Une première proposition a été faite dans ce sens, par dans [LMRN06]. Cet algorithme reprend notre algorithme tolérant aux défaillances [SAS06a] et le compose de manière statique suivant l'approche développée dans [BAS04]. Cet algorithme n'a toutefois pas encore été testé.

En complément de cette proposition, il pourrait être intéressant de concevoir un algorithme hiérarchique générique tolérant aux défaillances. Ainsi, il serait possible de composer les nombreux algorithmes présentés dans notre état de l'art. Cet aspect générique serait d'autant plus intéressant que les algorithmes d'exclusion mutuelle tolérant aux fautes existant présentent des caractéristiques très différentes. Permettre une libre composition permettrait de s'adapter aux multiples hypothèses imposées par les réseaux physiques.

Un tel algorithme générique pose toutefois le problème de la gestion de la défaillance des coordinateurs. Une première solution est de considérer les coordinateurs comme des passerelles. Un cluster dont le coordinateur est tombé en panne sera considéré comme

entièrement défaillant (qu'un de ses nœuds exécute ou non la section critique). La composition de notre algorithme tolérant aux défaillances par notre algorithme hiérarchique est alors une solution.

Une deuxième approche est de considérer une redondance des coordinateurs au sein des clusters sachant que plus il y aura de coordinateurs par cluster et moins la composition sera efficace. Cette solution demande donc de faire un compromis entre le degré de tolérance aux défaillances désiré et les performances de la composition.

Une troisième approche, plus difficile à mettre en œuvre, est celle de l'utilisation de coordinateurs tournant. Cette solution présente l'avantage de ne pas restreindre le nombre des défaillances tout en conservant les bonnes performances de la composition en absence de panne. La difficulté résulte ici de l'intégration du nouveau coordinateur dans les structures logiques de l'algorithme s'exécutant entre les clusters.

Annexe A

Vérification formelle de notre algorithme générique de composition

A.1 Introduction

Cette annexe présente une vérification des propriétés *qualitative* (sûreté et vivacités) de notre algorithme de composition. S'il vient en complément de la preuve formelle présentée à la section 6.6, elle est une première étape en vue de faire une étude quantitative des propriétés d'équité et de qualité de service.

Ainsi, nous avons modélisé notre algorithme de composition à l'aide de réseaux de Petri (RdP). En effet, les RdP occupent une place privilégiée dans le domaine de la modélisation des systèmes concurrents grâce, entre autres, à une expression simple de la synchronisation. Ils offrent un grand nombre d'outils d'analyse structurelle et comportementale. Parmi les méthodes comportementales nous avons utilisé la technique dite de "*model checking*" pour vérifier les propriétés attendues de notre algorithme. Ces propriétés seront exprimées en logique temporelle linéaire ("*LTL*").

Dans la section A.2 nous présentons une modélisation générique de notre solution en réseaux de Petri. Dans La section A.3, nous donnerons la traduction des propriétés de l'exclusion mutuelle en LTL. Avant d'en faire la vérification, nous décrirons des instanciations possibles des algorithmes composés (section A.4). Les résultats du *model checking* sont présentés dans la section A.5. Nous terminons cette annexe par les perspectives ouvertes par ces travaux.

A.2 Algorithme de composition - approche formelle

Parmi les outils couramment utilisés pour modéliser les systèmes distribués et concurrents, les réseaux de Petri occupent une place privilégiée. Ils permettent de représenter facilement la synchronisation et ils offrent des outils d'analyse du système basés sur des approches structurelles et comportementales.

Parmi l'ensemble des classes de réseaux de Petri, notre choix s'est porté sur les *Réseaux de Petri de Haut-Niveau* finis [Jen82], que nous noterons *RdPHN* et en particulier la classe

des Réseaux de Petri Bien-Formés stochastiques (*RdPBF*)[CDFH93] [18].

Trois raisons motivent ce choix : la capacité de ce modèle à représenter des systèmes de grande taille de façon compacte, une mise en oeuvre efficace de la vérification et l'intégration des aspects stochastiques (essentiellement via les transitions *temporisées* et *immédiates*), qui permettent d'une part, de simplifier la modélisation et d'autre part, de faciliter le passage à l'étude quantitative.

Pour modéliser notre algorithme de manière fidèle, nous suivons une approche incrémentale. Celle-ci permet de dégager les éléments de base sur lesquels reposent le fonctionnement global de l'algorithme de composition.

A.2.1 Modélisation du comportement d'une application utilisant un service d'exclusion mutuelle

Dans un système distribué, le schéma classique suivant lequel évolue un processus pour accéder à une section critique est une suite (potentiellement) infinie de trois étapes correspondant aux trois états de l'automate 2.1(a) décrit dans la section 2.2.1. À savoir : NO_REQ, REQ et CS. Le *RdPHN* de la figure A.1 formalise ce comportement. Dans ce réseau un processus x se déplace circulairement dans les trois places correspondant à ces trois états.

Initialement, la place NO_REQ est marquée par x, le processus effectue une tâche locale (pendant un certain temps) et n'est pas intéressé par la section critique. Lorsque le processus a besoin de la section critique pour exécuter du code concurrent, il l'exprime par le franchissement de la transition $requestCS()$. Ce changement d'état est enregistré par le marquage de la place REQ. L'obtention de l'autorisation d'accès à la section critique se traduit par le franchissement de la transition $AccessCS()$ (et le marquage de la place CS). La place CS est alors marquée par x et le processus peut réaliser sa tâche critique. Lorsqu'il a finit, il retourne à ses tâches locales en franchissant la transition $releaseCS()$.

Ainsi le sous-réseau, composé des places NO_REQ, REQ, CS et de leurs transitions adjacentes, modélise le comportement purement applicatif des processus. L'unicité d'accès à la place CS et la gestion de la file d'attente des requêtes sont assurées par un mécanisme de contrôle distribué : l'algorithme d'exclusion mutuelle. Cet algorithme interagit avec la partie applicative du processus à chaque changement d'état (*i.e.*, sur chacune des transitions).

Pour simplifier cette première modélisation, nous n'émettons aucune hypothèse sur le choix de l'algorithme d'exclusion mutuelle. L'important est ici de modéliser l'utilisation d'un tel algorithme et non l'algorithme lui même. Nous l'abstrayons donc dans la figure A.1 par le nuage nommé "*mutex*". Ce nuage couvre tout le sous-réseau de Petri modélisant les mécanismes de l'algorithme. À la bordure de ce dernier, on trouve deux places : la place *grantReq* dont le marquage par x modélise l'enregistrement d'une requête d'un processus par l'algorithme et la place *grant* qui modélise par son marquage l'autorisation d'accès à la section critique.

18. La syntaxe des *RdPBF* étant ardue, nécessitant parfois des expressions très longues, nous nous permettons l'utilisation d'un "sucre syntaxique" facilitant la description.

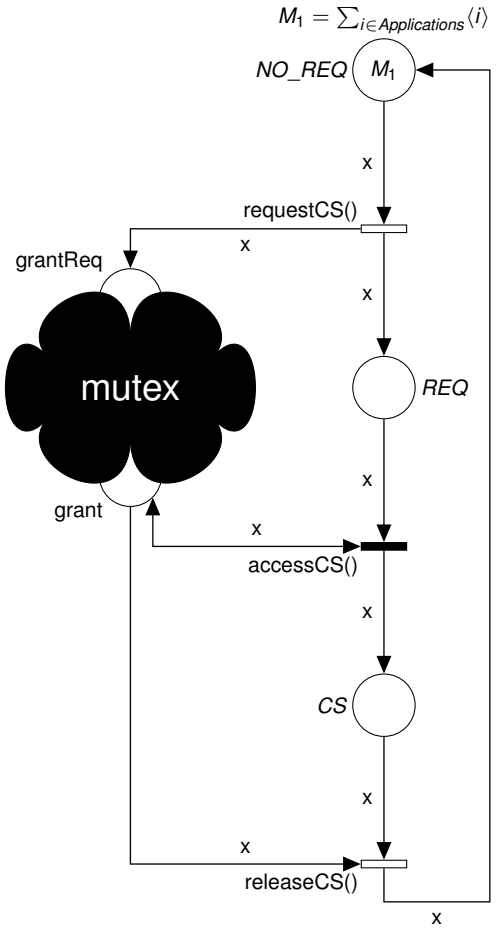


FIGURE A.1 – *RdPHN du comportement dans un cluster*

On retrouve dans cette modélisation l'A.P.I. des algorithmes d'exclusion mutuelle décrit dans le pseudo code de l'algorithme 2 : $requestCS()$ (ligne 123) et $releaseCS()$ (ligne 128). Un algorithme utilise ces 2 fonctions après un temps aléatoire de calcul. Ceci explique la temporisation des 2 transitions correspondantes ( ).

À l'inverse, le franchissement de la transition $AccessCS()$ correspond à la fin de l'instruction bloquante "*wait for token*" du pseudo-code (algorithme 123, ligne 126). Le temps passé dans la place *REQ* n'est donc pas aléatoire. Il est déterminé par la disponibilité de l'autorisation. Cette condition est réalisée par l'arc test sur la place *grant*. L'absence de temporisation se traduit par une transition immédiate ( ).

A.2.2 Modélisation de l'algorithme de composition

Grâce au *RdPHN* précédent, la modélisation de notre algorithme de composition (section 6.2) devient une opération moins ardue. En effet, l'algorithme de composition peut se voir comme l'utilisation "synchronisée" de deux instances d'algorithmes d'exclusion mutuelle (une pour le niveau *inter* et une pour le niveau *intra*).

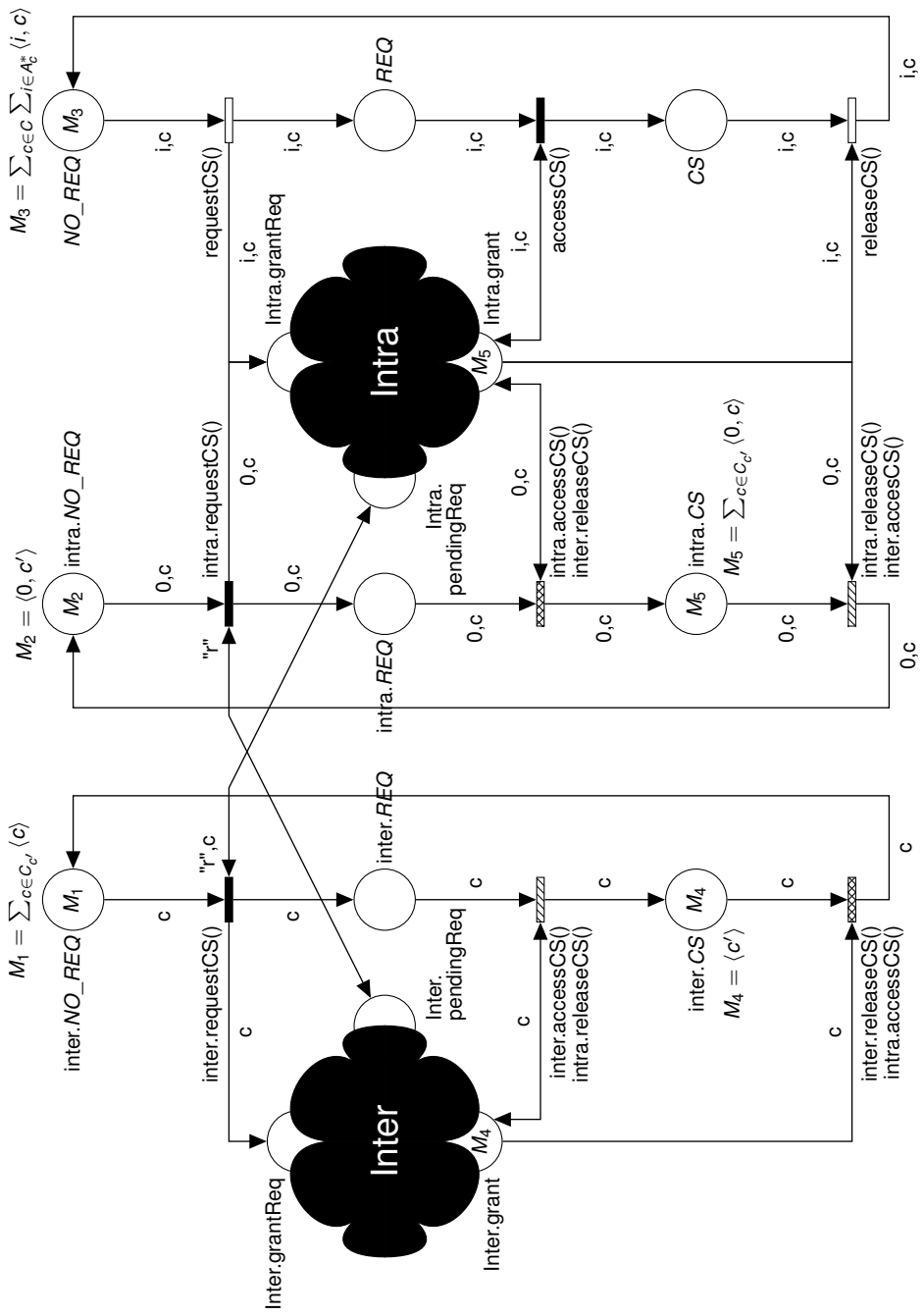


FIGURE A.2 – RdPHN de l'algorithme de composition.

Pour commencer la description de la figure A.2 qui modélise l'approche par composition, notons que l'on choisit dans ce réseau de replier toutes les instances des algorithmes *intra* dans un unique nuage "*intra* ". Comme dans la section 6.2, on considère alors que tous les algorithmes *intra* sont des instances locales d'un même algorithme. Le couple $\langle i, c\rangle$ identifie alors le processus i du cluster c. Ainsi, en gardant toutes les transitions par l'identifiant c du cluster, on *isole* chacune des instances locales.

Parmi les processus d'un cluster, on distingue le coordinateur en l'identifiant par l'indice 0. On explicite son comportement vis-à-vis de son algorithme *intra* en dépliant les places et les transitions du modèle précédent (figure A.1) par rapport à cet indice. Les étiquettes des éléments du sous-réseau, ainsi déplié, sont préfixées par "*intra.*".

Le sous-réseau constitué des places NO_REQ, REQ, CS et de leurs transitions adjacentes, modélise le comportement des nœuds applicatifs ($\langle i, c\rangle$, pour $i \neq 0$) pour chacun des clusters. Dans la suite de cet annexe nous nommerons ce sous-réseau : "sous-réseau applicatif". Les noms de ces places et de ces transitions ne sont pas préfixés par "*intra.*", ces nœuds ne participant qu'à une seule instance d'algorithme. Le schéma de ces sous-réseaux reste donc parfaitement identique au schéma de la figure A.1, exprimant ainsi la transparence de la composition pour les applications décrites dans la section 6.2.

Le sous-réseau - nommé par la suite "sous-réseaux *intra* " est composé des places préfixées par "*intra.*" et de leurs transitions adjacentes. Il diffère au niveau de ses transitions du schéma de la figure A.1. En effet, il modélise le comportement des coordinateurs vis-à-vis de leur algorithme *intra*. Or, comme on l'a présenté dans la section 6.2, les coordinateurs ne font pas de requête pour eux mêmes mais uniquement pour assurer la circulation du jeton. Le temps passé dans les places "*intra.*NO_REQ " et "*intra.*CS " n'est donc pas aléatoire mais reste conditionné par la "mécanique" de la composition. Elles sont donc modélisées par des transitions immédiates.

De même, on définit le "sous-réseau *inter* " comme étant le sous-réseau composé des places préfixées par "*inter.*" et de leurs transitions adjacentes. Il définit le comportement des coordinateurs vis-à-vis de l'algorithme *inter*. Le système n'ayant qu'un coordinateur par cluster, ils sont distingués ici par l'identifiant c de leur cluster. Enfin, tout comme pour le sous-réseau *intra*, et pour les mêmes raisons, les transitions "*inter.*requestCS()" et "*inter.*CS " doivent être des transitions immédiates.

On a donc ici modélisé, pour chaque coordinateur, un comportement *intra* correspondant à la couleur $\langle 0, c\rangle$ dans le sous-réseau *intra* et un comportement *inter* correspondant à la couleur $\langle c\rangle$ dans le sous-réseau *inter*. Cette modélisation reflète bien l'esprit de l'automate de la figure 6.2(b) (section 6.2.2) où chaque état d'un coordinateur est défini par un tuple composé de son état *inter* et de son état *intra*. Pour respecter cet automate il faut donc synchroniser ces deux comportements.

On peut diviser la synchronisation de ces deux sous-réseaux en deux grandes parties. La première concerne les transitions $inter.requestCS()$ et $intra.requestCS()$ qui conditionnent l'émission d'une requête. La seconde concerne les transitions $inter.releaseCS()/intra.accessCS()$ et $intra.releaseCS()/inter.accessCS()$ qui contrôlent le "bon relâchement" des sections critiques.

L'émission d'une requête par un coordinateur peut être assimilée, du point de vue système, à une tranmission ("*forward*") d'un niveau vers l'autre. Elle est donc conditionnée par la réception d'une requête *inter* (resp. *intra*). Ceci nécessite de réifier cette information

de l'algorithme. Ainsi dans le pseudo-code de la figure, s'ajoute à l'A.P.I. standard, la fonction *pendingRequest*() (ligne 133). Pour modéliser cette réification, on ajoute au nuage une place *pendingReq*. Son marquage devra indiquer la réception d'une requête. La synchronisation de la transition *inter.requestCS*() (resp. *intra.requestCS*()) se fait alors en ajoutant un arc test sur la place *intra.pendingReq* (resp. *inter.pendingReq*).

Le relâchement de la section critique *inter* (resp. *intra*) est conditionné par l'accès à la section critique *intra* (resp. *inter*). Modéliser ce comportement revient à synchroniser de façon croisée une transition *accesCS*() avec une transition *releaseCS*(). Sur le schéma du modèle de la figure A.2, les transitions nommées *inter.accessCS*()/*intra.releaseCS*() ( ) schématisent la même transition immédiate, réalisant ainsi la synchronisation désirée. Il en est de même pour les transitions immédiates nommées *intra.accessCS*()/*inter.releaseCS*() ( ) .

On note ici que la duplication de ces deux transitions permet d'une part de simplifier le modèle pour la lecture et d'autre part de faire apparaître le schéma comportementale des coordinateurs au niveau de chaque algorithme (inter et intra) ainsi que les contraintes de synchronisation qui lient chacune des instances.

Enfin, il reste à spécifier le marquage initial. On définit les ensembles suivants :

C **:** ensemble des identifiants des clusters.

$C_{c'} = \{c \in C | c \neq c'\}$ **:** ensemble des identifiants des clusters privé de c'.

A_c **:** ensemble des identifiants des noeuds du cluster c.

$A_c^* = \{i \in A_c | i \neq 0\}$ **:** ensemble des identifiants des noeuds applicatifs du cluster c.

Si la notation $M(p)$ représente le marquage de la place p, l'initialisation décrite dans la section 6.2 se traduit alors par le marquage initial suivant :

- Tous les noeuds applicatifs sont dans l'état NO_REQ

$$M(NO\_REQ) = \sum_{c \in C} \sum_{i \in A_c^*} \langle i, c \rangle$$

- Un des coordinateurs du système possède la section critique *inter* mais est dans l'état NO_REQ pour l'algorithme *intra* :

$$M(intra.NO\_REQ) = \langle 0, c' \rangle \quad \text{et} \quad M(inter.CS) = M(inter.grant) = \langle c' \rangle$$

- Le reste des coordinateurs détiennent leur section critique *intra* et sont dans l'état NO_REQ pour l'algorithme *inter* :

$$M(inter.NO\_REQ) = \sum_{c \in C_{c'}} \langle c \rangle \quad \text{et} \quad M(intra.CS) = M(intra.grant) = \sum_{c \in C_{c'}} \langle 0, c \rangle$$

Notons que c' est choisi de façon arbitraire. La génération d'un tel élément peut être facilement effectuée en ajoutant une place p_Init, initialisée par $\sum_{c \in C} \sum_{i \in A_c} \langle i, c \rangle$ (qui est un marquage symétrique), connectée en entrée à une transition immédiate t_Init, qui elle même est connectée en sortie aux différentes places (du réseaux) concernées par l'initialisation. Un franchissement (unique) de cette transition, en utilisant des variables libres, permet de générer une initialisation des places concernées, sans engendrer d'asymétries supplémentaires. Pour ne pas alourdir le $RdPHN$ présenté, ce mécanisme n'y est pas figuré. Il est par contre nécessaire lors de la vérification pour maintenir la symétrie.

A.3 Expression en logique temporelle linéaire des propriétés de l'exclusion mutuelle

Nous allons ici exprimer les propriétés de l'exclusion mutuelle introduite dans le chapitre 2 à la section 2.2.2 en utilisant une Logique Temporelle Linéaire (LTL) [Pnu77]. Nous commençons d'abord par définir les différentes proposions atomiques qui vont nous aider à les exprimer.

- P_1 : le processus i du cluster c est au repos

$$\langle i, c\rangle \in M(NO\_REQ)$$

- P_2 : le processus i du cluster c demande la section critique

$$\langle i, c\rangle \in M(REQ)$$

- P_3 : le processus i du cluster c est en section critique

$$\langle i, c\rangle \in M(CS)$$

- P_4 : le nombre de processus applicatifs en section critique est inférieur ou égal à 1 [19]

$$\#(CS) \leqslant 1$$

- P_5 : il n'y a pas de processus applicatif en section critique

$$\#(CS) = 0$$

- P_6 : il y a un unique processus applicatif en section critique

$$\#(CS) = 1$$

- P_7 : il y a au moins un processus applicatif qui demande la section critique

$$\#(REQ) \geqslant 1$$

Notons que dans un soucis de lisibilité, P_1,P_2 et P_3 ne seront pas paramétrées par i et c. Ainsi, les propriétés à vérifier s'expriment de la manière suivante :

Validité (well-formedness) : si le processus est en état NO_REQ (resp. REQ, CS), il ne pourra pas passer par l'état CS (resp. NO_REQ, REQ) sans être avant passé par l'état REQ (resp. CS, NO_REQ).

$$F_1 = G(P_1 \ \Rightarrow \ (!P_3 \ U \ P_2)) \wedge G(P_2 \ \Rightarrow \ (!P_1 \ U \ P_3)) \wedge G(P_3 \ \Rightarrow \ (!P_2 \ U \ P_1))$$

Exclusion mutuelle : il y a toujours au plus un nœud applicatif en section critique.

$$F_2 = G(P_4)$$

19. La notation $\#(P)$ est utilisée pour exprimer : le nombre de jetons décolorés marquant la place P.

Progression : toujours, s'il y a au moins un nœud applicatif en état REQ et qu'il n'y aucun nœud applicatif en état CS, alors un nœud applicatif sera en état CS dans le futur.

$$F_3 = G((P_7 \wedge P_5) \Rightarrow F(P_6))$$

Équité faible : un nœud obtiendra toujours la section critique après l'avoir demandé.

$$F_4 = G(P_2 \Rightarrow F(P_3))$$

nchissement de la transition $inter.accessCS()$

A.4 Modélisation des algorithmes d'exclusion mutuelle

Pour vérifier les propriétés précédentes sur notre algorithme de composition, il faut instancier les abstractions (nuages) des algorithmes *inter* et *intra*. Une première approche, consiste à remplacer chaque nuage par un *RdPHN* modélisant de manière fidèle les algorithmes utilisés comme [SK85], [NT87b], [Mar85]. Cependant, ce degré de réalisme n'a de sens que pour effectuer une étude quantitative liée au comportement intrinsèque de chaque algorithme : temps moyen d'attente, complexité moyenne en nombre de messages, etc ...

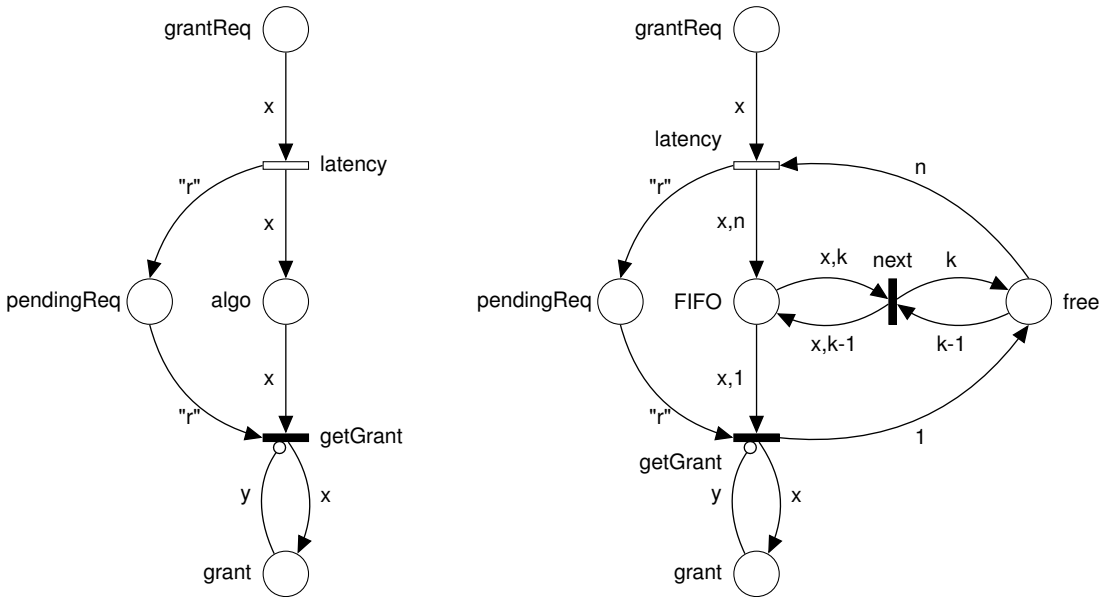
Dans cet étude, nous avons choisi de ne composer que des modélisations des propriétés de l'exclusion mutuelle. Cette approche ne permet certes pas de garantir que toutes les compostions seront valides. Mais elle permet de faire une première étude qualitative de l'algorithme de composition sur l'ensemble des exécutions engendrées par la composition de ces modèles. Nous nous sommes donc attachés à rester le plus générique possible dans la modélisation de ces propriétés.

On propose ici deux *RdPHN* garantissant chacun certaines propriétés décrites dans la section précédente. Le schéma de la figure A.3(a) considère l'exclusion mutuelle au sens strict : il garantit les propriétés de *validité*, *d'exclusion mutuelle* et de *progression*. Tandis que celui de la figure A.3(b) garantie en plus la propriété d'*équité faible*

Commençons par étudier le réseau de la figure A.3(a). On remarque d'abord que l'on retrouve bien, sur ce schéma, les trois places présentes en bordure des nuages de la figure A.2. À savoir : *grantRequest*, *pendingReq* et *grant*. S'ajoute à ces places, la place *algo* qui modélise la gestion des requêtes en attente.

L'évolution de ce réseau est régie par deux transitions. La première, nommée *latency*, modélise la réception d'une requête sans spécifier son mode d'acheminement (un envoi direct comme dans l'algorithme de Suzuki et Kasami [SK85] ou de transferts successifs comme dans celui de Martin [Mar85]). Le temps des traitements et celui passé dans le réseau étant aléatoires, cette transition est temporisée ( ). L'accès à la section critique est, quant à lui, modélisé par la transition *getGrant*. L'unicité d'accès à la section critique est assurée par un arc inhibiteur [20] sur la place *grant*. La propriété de *progression* sur les requêtes enregistrées (*i.e.*, marquant la place *algo*) est modélisée par le caractère immédiat ( ) de la transition *getGrant*.

20. Dans un cadre général, l'utilisation d'arcs inhibiteurs peu poser un problème théorique de décidabilité. Cela n'affecte cependant pas notre modèle car il est borné.



(a) Algorithme d'exclusion mutuelle sans équité

(b) Algorithme d'exclusion mutuelle avec équité

FIGURE A.3 – Algorithmes d'exclusion mutuelle.

Pour finir, remarquons que les domaines des places *algo* et *pendingReq* ne sont pas identiques. En effet, seule la présence d'une requête a besoin d'être réifiée. Lors du franchissement de la transition *latency*, par une couleur $\langle x \rangle$, la place *pendingReq* est marquée par la constante "r" (signifiant la présence d'une requête). L'identité du demandeur n'est donc pas réifiée.

Dans ce réseau, l'*équité faible* n'est pas garantie : certaines couleurs marquant la place *algo* peuvent ne jamais franchir la transition *getGrant*. Elles peuvent être perpétuellement doublées par de nouvelles requêtes.

La figure A.3(b) reprend le schéma précédent, en substituant à la place *algo* la modélisation d'une file de requêtes. Celle-ci contient deux places : la place $FIFO$ dont le marquage $\langle x, k \rangle$ associe à chaque requête x une position k dans la file et une place $free$ qui stocke l'ensemble des positions disponibles. Lorsqu'elle arrive dans la place $FIFO$, une requête se voit assigner la dernière position n (taille de la FIFO). Elle progresse ensuite dans la file en franchissant la transition immédiate *next* qui décrémente sa position dans la file d'attente. Seule la requête possédant la position 1 peut franchir la transition *getGrant*. Les demandes étant traitées dans l'ordre de leur enregistrement, la propriété d'*équité faible* est garantie pour toutes les requêtes enregistrées (*i.e.*, ayant franchies la transition *latency*).

Garantir cette propriété pour toutes les requêtes émises (*i.e.*, marquant la place *grantReq*), nécessite la modélisation d'une hypothèse supplémentaire. En effet les algorithmes d'exclusion mutuelle font tous, au minimum, l'hypothèse de canaux à pertes équitables (*"fair lossy channel"*). Cette hypothèse garantit qu'un message émis une infinité de fois, sera reçu une infinité de fois. Autrement dit : on ne perd pas toujours le même

message. Pour modéliser cette hypothèse, il y a deux solutions. La première consiste (si le modèle le permet) à rendre équitable le franchissement de la transition *latency* : pour toute exécution (infinie) du modèle, si la transition *grantReq* est franchissable, alors elle finira toujours par être franchie. La deuxième consiste à introduire cette hypothèse directement dans les propriétés temporelles à vérifier. Avec cette deuxième approches, toutes les exécutions où la transition *grantReq* est franchissable mais n'est jamais franchie sont, systématiquement, ignorées pendant la vérification.

Le modèle choisi, étant une variante classique des réseaux de Petri, et ne permettant pas de rendre les transitions équitables, nous prenons la deuxième solution. L'expression de la propriété d'équité faible, précédemment introduite, doit donc être revue. Nous rajoutons pour cela deux propositions atomiques :

- P_8 : une requête du nœud i' du cluster c' n'a pas été traitée (émise mais non reçue) dans un algorithme *intra*

$$\langle i', c' \rangle \in M(intra.grantRequest)$$

- P_9 : une requête du coordinateur du cluster c'' n'a pas été traitée (émise mais non reçue) dans l'algorithme *inter*

$$\langle c'' \rangle \in M(inter.grantRequest)$$

L'expression de l'équité faible doit donc être modifiée comme suit.

Équité faible : toujours, si un processus i du cluster c demande la section critique, soit dans le futur il aura la section critique, soit au moins un message destiné à un nœud applicatif i' de c' n'arrivera jamais, soit au moins un message destiné au coordinateur d'un cluster c'' n'arrivera jamais.

$$F_4 = G(P_2 \Rightarrow (F(P_3) \vee FG(P_8) \vee FG(P_9))$$

A.5 Vérification du modèle

La méthode classique pour la vérification de modèle (*model-checking*) [Var96] de propriétés LTL repose sur la théorie des automates. Dans cette approche, l'automate représentant l'ensemble des exécutions que peut prendre le système (le modèle du système) est confronté à l'automate reconnaissant l'ensemble des exécutions *invalidant* la propriété à vérifier. Si le deuxième automate reconnaît une des exécutions du premier alors la propriété originelle n'est pas vérifiée.

Le problème majeur de ce type de méthodes est la taille souvent excessive des automates traités. En effet, elle peut être exponentielle par rapport à la taille de la description du système. Cette explosion combinatoire du nombre d'états est due essentiellement à l'exécution concurrente des différents objets qui composent le système et à leur nécessaire synchronisation.

Pour limiter les effets néfastes de cette explosion combinatoire, différentes solutions ont été proposées. Elles ont pour objectif de réduire la représentation de ces automates et/ou de leur substituer des automates plus petits. Une de ses solutions se base sur l'observation qu'un système concurrent est souvent constitué d'éléments ayant des comportements

Topo. / Propr.	6 nœuds			8 nœuds		
	6 a.	2 c. 3 a.	3 c. 2 a.	8 a.	2 c. 4 a.	4 c. 2 a.
F_1	1438	70823	145455	2888	619362	1793654
F_2	218	9988	20205	391	74817	212666
F_3	318	14548	30662	569	108295	320577
F_4	785	36844	76018	1716	345375	708019

TABLE A.1 – Quelques données sur la verification réalisée.

fortement symétriques, *i.e.*, identiques à une permutation près. La factorisation de la représentation de ces comportements similaires permet la construction d'automates réduits, qui peuvent être utilisés pour la vérification [BHI04, Baa07].

L'algorithme que nous avons développé étant fortement symétrique, c'est cette approche que nous allons adopter pour effectuer notre vérification. En effet, les symétries comportementales sont observées sur notre système à tous les niveaux : de même que les comportements des nœuds applicatifs au sein d'un cluster sont symétriques, aussi, le sont les comportements des nœuds coordinateurs.

Contrairement aux *RdPHN* généraux, le modèle des *RdPBF* permet une construction *automatique* des graphes quotients (représentant l'ensemble des exécutions du système), en exploitant les *symétries comportementales*. Ces symétries sont automatiquement détectées grâce à l'utilisation d'une syntaxe particulière. Ainsi, pour vérifier une propriété il n'est pas nécessaire de considérer individuellement chaque couleur des domaines (impliqués dans cette dernière), la vérification pouvant se faire sur un unique représentant choisi de façon arbitraire.

L'outil utilisé pour la représentation de notre *RdPBF* et la génération de l'espace d'états est une version modifiée de *GreatSPN*[21]. Cet outil est mondialement connu pour sa représentation et sa gestion efficace des *RdPBF*. La vérification proprement dite (*model-checking*) est réalisée en utilisant l'outil *Spot*[22]. La connexion des deux outils a permis la vérification de notre modèle, vis-à-vis, des propriétés citées dans les sections précédentes. Nous réalisons cette vérification en deux étapes : (1) dans un premier temps, nous vérifions les modèles des algorithmes d'exclusion mutuelle (figure A.3) en les insérant dans le modèle général de la figure A.1 ; (2) puis nous insérons ces algorithmes dans le *RdPBF* de notre approche compositionnelle (figure A.2). Trivialement, pour le point (1), le *model-checker* vérifie les propriétés F_1 à F_3 sur le modèle de la figure A.3(a) et les propriétés F_1 à F_4 sur le modèle A.3(b).

Pour le point (2), nous utilisons en *intra* et en *inter* le modèle *RdPHN* de la figure A.3(b). **Les résultats obtenus montrent que notre solution satisfait les propriétés F_1 à F_4 sur l'ensemble de ces modèles testés.** Le tableau A.1 récapitule le nombre d'états parcourus pour vérifier ces propriétés sur la composition d'algorithmes faiblement équitables (figure A.3(b)) pour quelques configurations. Le but étant ici de mettre en évidence la complexité de la vérification d'un tel problème et son évolution vis-à-vis de la topologie.

21. http ://www.di.unito.it/ greatspn/
22. http ://spot.lip6.fr/

Six topologies notées "xc." et "ya." sont reportées dans ce tableau : x représente le nombre de cluster (et donc le nombre de noeuds coordinateurs) et y le nombre de nœuds applicatifs par cluster. Les configurations notées seulement par "y a." représentent le cas traditionnel sans composition (*i.e.*, où il n'y a qu'un seul algorithme gérant l'ensemble des nœuds). Pour nos calculs de test, six nœuds applicatifs sont regroupés en 2 puis 3 clusters et huit nœuds applicatifs sont regroupés en 2 puis 4 clusters. Ce découpage de la grille en clusters de taille identique, n'est pas lié à l'utilisation des techniques de symétrie, mais permet simplifier le modèle fourni aux outils de vérification. En effet, quelle que soit la taille des clusters, les noeuds conservent un comportement symétrique. On remarque la complexité (en terme de nombre d'états) de la vérification de l'algorithme de composition par rapport à celle relativement faible d'un algorithme ordinaire (sans composition).

A.6 Conclusion

Dans cette annexe, nous avons présenté la vérification de notre algorithme de composition par *model cheking* Pour effectuer la vérification, nous avons isolé une architecture particulière commune à tous les algorithmes d'exclusion mutuelle. Puis nous en avons établi une interface générique *RdP*. Cela nous a permis de modéliser d'une manière compositionnelle notre mécanisme. En utilisant cette A.P.I., nous avons pu injecter des abstractions d'algorithmes d'exclusion mutuelle vérifiant des propriétés bien choisies. Cette simplification, suffisante pour une étude qualitative, a permis la vérification de notre algorithme vis-à-vis de propriétés désirées. Une étude minutieuse de ces propriétés nous a permis de déduire leur expression en logique temporelle linéaire. Lors de cette formalisation, nous avons pris en compte l'hypothèse d'équité devant les pertes de messages faite par la plupart des algorithmes distribués.

Durant l'ensemble de ce processus de vérifications, nous avons cherché à utiliser les propriétés de symétrie. Ainsi, après les avoir mises en évidence dans l'algorithme proposé,nous les avons exploitées pour en faire une modélisation *RdP* concise de notre algorithme et pour simplifier l'expression *LTL* des propriétés. La conservation de ces symétries dans le modèle et dans les propriétés, nous a permis d'optimiser le processus de vérification par l'utilisation d'algorithmes spécifiques de *model cheking*.

Cette première étude ouvre de nombreuses perspectives très intéressantes. La modélisation fine en RdP d'algorithmes classiques tel que Suzuki et Kasami [SK85], Naimi-Tréhel [NT87b] ou Martin [Mar85] nous permettra d'effectuer une étude quantitative précise de notre solution et nous permettra de déterminer l'influence de notre solution sur le comportement des nœuds applicatifs. Cette étude permettra de calculer les indices de performance liés aux comportements intrinsèques des algorithmes composés.

Bibliographie

[AA95] Divyakant Agrawal and Amr El Abbadi. A token-based fault-tolerant distributed mutual exclusion algorithm. *Journal of Parallel and Distributed Computing*, 24(2) :164–176, february 1995.

[ABJ05] Gabriel Antoniu, Luc Bougé, and Mathieu Jan. Juxmem : An adaptive supportive platform for data sharing on the grid. *Scalable Computing : Practice and Experience*, 6(3) :45–55, September 2005.

[Ang80] Dana Angluin. Local and global properties in networks of processors (extended abstract). In *Annual ACM Symposium on Theory of Computing (STOC)*, pages 82–93, 1980.

[BAA89] José M. Bernabéu-Aubán and Mustaque Ahamad. Applying a path-compression technique to obtain an efficient distributed mutual exclusion algorithm. In *Proceedings of the 3rd International Workshop on Distributed Algorithms*, pages 33–44, London, UK, 1989. Springer-Verlag.

[Baa07] Souheib Baarir. *Exploitation des symétries partielles pour la vérification et l'évaluation de performances des systèmes finis.* PhD thesis, Université Pierre et Marie Curie - Paris VI, 2007.

[BAS04] Marin Bertier, Luciana Arantes, and Pierre Sens. Hierarchical token based mutual exclusion algorithms. In *4th IEEE/ACM International Symposium on Cluster Computing and the Grid*, April 2004.

[BAS06] Marin Bertier, Luciana Arantes, and Pierre Sens. Distributed mutual exclusion algorithms for grid applications : A hierarchical approach. *Journal of Parallel and Distributed Computing*, 66 :128–144, 2006.

[BG02] Gordon Bell and Jim Gray. What's next in high-performance computing ? *Communiction of ACM*, 45(2) :91–95, 2002.

[BHI04] Souheib Baarir, Serge Haddad, and Jean-Michel Ilié. Exploiting Partial Symmetries in Well-formed nets for the Reachability and the Linear Time Model Checking Problems. In *Proceeding of IFAC Workshop on Discrete Event Systems, part of 7th CAAP*, Reims - France, 2004. Springer Verlag.

[BSL08] Soheib Baarir, Julien Sopena, and Fabrice Legond. On the formal verification of a generic hierarchical mutual exclusion algorithm. In *The 28th International Conference on Formal Techniques for Networked and Distributed Systems - FORTE08*, Lecture Notes in Computer Science, Tokyo, Japan, June 2008. Springer.

[Bur81] James E. Burns. Symmetry in systems of asynchronous processes. In *22nd Annual Symposium on Foundations of Computer Science (FOCS), Nashville, Tennessee, USA*, pages 169–174, 1981.

[CASD85] Flaviu Cristian, Houtan Aghili, Ray Strong, and Danny Dolev. Atomic broadcast : from simple message diffusion to byzantine agreement. In *Proceedings of the 15th International Symposium on Fault-Tolerant Computing Systems (FTCS-15)*, 1985.

[CDD+06] F. Cappello, F. Desprez, M. Dayde, E. Jeannot, Y. Jegou, S. Lanteri, N. Melab, R. Namyst, P. Vicat-Blanc Primet, O. Richard, E. Caron, J. Leduc, and G. Mornet. Grid5000 : a nation wide experimental grid testbed. *International Journal on High Performance Computing Applications*, Vol. 20(No. 4) :481–494, 2006.

[CDFH93] Giovanni Chiola, Claude Dutheillet, Giuliana Franceschinis, and Serge Haddad. Stochastic well-formed coloured nets for symmetric modelling applications. *IEEE Transactions on Computers*, 42(11) :1343–1360, November 1993.

[CM84] K. Mani Chandy and Jayadev Misra. The drinking philosopher's problem. *ACM Transactions on Programming Languages and Systems*, 6(4) :632–646, 1984.

[CM88] K. Mani Chandy and Jayadev Misra. *Parallel program design : a foundation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1988.

[CM92] K. Mani Chandy and Jayadev Misra. *Synchronisation et état global dans les syst'emes répartis*. Eyrolles, 1992.

[CR79] Ernest Chang and Rosemary Roberts. An improved algorithm for decentralized extrema-finding in circular configurations of processes. *Communications of the ACM*, 22(5) :281–283, may 1979.

[CR83] Osvaldo S.F. Carvalho and Gérard Roucairol. On mutual exclusion in computer networks. *Communications of ACM*, 26(2) :146–147, 1983.

[CSL90a] I. Chang, M. Singhal, and M. Liu. A hybrid approach to mutual exclusion for distributed system. In *IEEE International Computer Software and Applications Conference*, pages 289–294, 1990.

[CSL90b] Ye-In Chang, Mukesh Singhal, and Ming T. Liu. A fault-tolerant algorithm for distributed mutual exclusion. In *Proc. of the 9th IEEE Annual Symposium on Reliable Distributed Systems*, pages 146–154, 1990.

[CSL91] Ye-In Chang, Mukesh Singhal, and Ming 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, Mar 1991.

[DGFGK05] Carole Delporte-Gallet, Hugues Fauconnier, Rachid Guerraoui, and Petr Kouznetsov. Mutual exclusion in asynchronous systems with failure detectors. *Journal of Parallel and Distributed Computing*, 65(4) :492–505, 2005.

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

[Dij87] Edsger W. Dijkstra. Position paper on "fairness". circulated privately, October 1987.

[DKR82] Danny Dolev, Maria M. Klawe, and Michael Rodeh. An o (n log n) unidirectional algorithm for extrema finding in a circle. *Journal of Algorithms*, 3(3) :245–260, September 1982.

[Era95] Stéphane Eranian. *Un service de synchronisation distribué tolérant les pannes : implémentation dans CHORUS*. PhD thesis, Université Denis Diderot - Paris VII, 1995.

[Erc04] K. Erciyes. Distributed mutual exclusion algorithms on a ring of clusters. In *International Conference on Computational Science and Its Applications*, volume 3045 of *LNCS*, pages 518–527, 2004.

[Fei05] Dror G. Feitelson. The supercomputer industry in light of the top500 data. *Computing in Science and Engineering*, 7(1) :42–47, 2005.

[FLP85] Michael Fischer, Nancy Lynch, and Mike Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM (JACM)*, 32(2) :374–382, April 1985.

[For94] Message Passing Interface Forum. Message Passing Interface Standard. Technical report, University of Tennessee, Knoxville, TN, USA, May 1994.

[For97] Message Passing Interface Forum. MPI-2 : Extensions to the Message-Passing Interface. Technical report, University of Tennessee, Knoxville, TN, USA, May 1997.

[Fra82] Randolph Wm Franklin. On an improved algorithm for decentralized extrema finding in circular configurations of processors. *Communications of the ACM*, 25(5) :336–337, may 1982.

[GLB] The Globus project. http://www.globus.org/.

[HM94] Jean-Michel Hélary and Achour Mostefaoui. A o(log 2 n) fault-tolerant distributed mutual exclusion algorithm based on open-cube structure. *14th International Conference on Distributed Computing Systems (ICDCS)*, pages 89–96, 1994.

[HP94] Lisa Higham and Teresa M. Przytycka. Asymptotically optimal election on weighted rings. In Erik Meineche Schmidt and Sven Skyum, editors, *4th Scandinavian Workshop on Algorithm Theory - SWAT*, pages 207–218, Aarhus, Denmark, July 1994.

[HPR88] Jean-Michel Hélary, Noël Plouzeau, and Michel Raynal. A distributed algorithm for mutual exclusion in an arbitrary network. *Computer journal*, 31(4) :289–295, 1988.

[HS80] Dan S. Hirschberg and James Bartlett Sinclair. Decentralized extrema-finding in circular configurations of processors. *Communications of the ACM*, 23(11) :627–628, november 1980.

[HT93] Vassos Hadzilacos and Sam Toueg. Fault-tolerant broadcasts and related problems. *Distributed systems (2nd Ed.)*, pages 97–145, 1993.

[HT01] A. Housni and M. Tréhel. Distributed mutual exclusion by groups based on token and permission. In *International Conference on Computational Science and Its Applications*, pages 26–29, June 2001.

[Jen82] Kurt Jensen. High-level petri nets. In *Proceedings of the 3rd European Workshop on Application and Theory of Petri Nets*, Varenna - Italy, 1982.

[KD94] Sandeep Kulkarni and Dhananjay Madhav Dhamdhere. A token based k-resilient mutual exclusion algorithm for distributed systems. *Information Processing Letters*, 50 :151–157, 1994.

[Lam74] Leslie Lamport. A new solution of dijkstra's concurrent programming problem. *Communications of the ACM*, 17(8) :453–455, 1974.

[Lam77] Leslie Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, 3(2) :125–143, 1977.

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of ACM*, 21(7) :558–565, 1978.

[Lan77] Gérard Le Lann. Distributed systems - towards a formal approach. In *International Federation for Information Processing (IFIP) Congress*, pages 155–160, 1977.

[Lan78] Gérard Le Lann. Algorithms for distributed data-sharing systems which use tickets. In *Proceedings of the Third Berkeley Workshop on Distributed Data Management and Computer Networks*, pages 259–272, August 1978.

[Lap85] Jean-Claude Laprie. Dependable computing and fault tolerance : concepts and terminology. In *Proceedings of the 15th International Symposium on Fault-Tolerant Computing Systems (FTCS-15)*, pages 2–11, Ann Arbor, Michigan, USA, june 1985.

[LMRN06] M. Loallemi, Y. Mansouri, A. Rasoulifard, and M. Naghibzadeh. Fault-tolerant hierarchical token-based mutual exclusion algorithm. In *International Symposium in Communications and Information Technologies*, pages 171–176, 2006.

[LS88] Leslie Lamport and Fred Schneider. Another position paper on fairness. *Software Engineering Notes*, 13(3) :18–19, July 1988.

[lue98] F. lueller. Prioritized token-based mutual exclusion for distributed systems. In *International Parallel Processing Symposium*, pages 791–795, March 1998.

[Lyn96] Nancy A. Lynch. *Distributed Algorithms.* Morgan Kaufmann, 1996.

[Mae85] Mamoru Maekawa. A square root n algorithm for mutual exclusion in decentralized systems. *ACM Transactions on Computer Systems (TOCS)*, 3(2) :145–159, 1985.

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

[MK94] Madhuram and Kumar. A hybrid approach for mutual exclusion in distributed computing systems. In *IEEE Symposium on Parallel and Distributed Processing*, 1994.

[MS94] D. Manivannan and Mukesh Singhal. An efficient fault-tolerant mutual exclusion algorithm for distributed systems. In *ISCA International Conference on Parallel and Distributed Computing Systems*, pages 525–530, 1994.

[MS96] Dakshnamoorthy Manivannan and Mukesh Singhal. Decentralized token generation scheme for token-based mutual exclusion algorithms. *International Journal of Computer Systems Science and Engineering*, 11(1) :45–54, 1996.

[Mue01] Frank Mueller. Fault tolerance for token-based synchronization protocols. In *Workshop on Fault-Tolerant Parallel and Distributed Systems, IEEE*, April 2001.

[MVF+92] Rozier M., Abrossimov V., Armand F., Boule I., Gien M., Guillemont M. ans S. Langlois, C. Kaiser, Lénoard P., and Neuhauser W. Overview of the chorus distributed operating system. In *Proceding of USENIX Workshop on Micro-kernels and Other Kernel Architecture*, pages 39–69, Seattle, April 1992.

[NLM90] Shojiro Nishio, Kin F. Li, and Eric G. Manning. A resilient mutual exclusion algorithm for computer networks. *IEEE Transactions on Parallel and Distributed Systems*, 1(3) :344–355, July 1990.

[NM91] Mitchell L. Neilsen and Masaaki Mizuno. A dag-based algorithm for distributed mutual exclusion. In *International Conference on Distributed Computing Systems (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. *Lecture Notes In Computer Science LNCS*, 312 :155–166, 1987.

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

[NT96] Mohamed Naimi and Michel Trehel. A log(n) distributed mutual exclusion algorithm based on the path reversal. *Journal of Parallel and Distributed Computing (JDPC)*, 34 :1–13, 1996.

[ON02] F. Omara and M. Nabil. A new hybrid algorithm for the mutual exclusion problem in the distributed systems. *International Journal of Intelligent Computing and Information Sciences*, 2(2) :94–105, 2002.

[Pet82] Gary L. Peterson. An o(nlog n) unidirectional algorithm for the circular extrema problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(4) :758–762, october 1982.

[Pnu77] Amir Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 46–57, Providence, Rhode Island, USA, October 1977. IEEE.

[RA81] Glenn Ricart and Ashok K. Agrawala. An optimal algorithm for mutual exclusion in computer networks. *Communications of ACM*, 24(1) :9–17, 1981.

[RA83] G. Ricart and A. K. Agrawala. Author response to 'on mutual exclusion in computer networks' by carvalho and roucairol. In *Communications of the ACM*, volume 26/2, pages 147–148, February 1983.

[Ray89] Kerry Raymond. A tree-based algorithm for distributed mutual exclusion. *ACM Transactions on Computer Systems*, 7(1) :61–77, 1989.

[Ray91] Michel Raynal. A simple taxonomy for distributed mutual exclusion algorithms. *Operating Systems Review*, 25(2) :47–50, 1991.

[Riz97] Luigi Rizzo. Dummynet : a simple approach to the evaluation of network protocols. *ACM Computer Communication Review*, 27(1) :31–41, 1997.

[SABS05] Julien Sopena, Luciana Arantes, Marin Bertier, and Pierre Sens. A fault-tolerant token-based mutual exclusion algorithm using a dynamic tree. In *Euro-Par 2005 Processing, Lisbon , PORTUGAL*, volume 3648/2005, pages 654–663, Heidelberg, Deutschland, August 2005. Springer Berlin.

[SALS08] Julien Sopena, Luciana Arantes, Fabrice Legond, and Pierre Sens. The impact of clustering on token-based mutual exclusion algorithms. In *Euro-Par 2008 Processing, Las Palmas , SPAIN*, volume 3648/2005, Heidelberg, Deutschland, August 2008. Springer Berlin.

[SALS09] Julien Sopena, Luciana Arantes, Fabrice Legond, and Pierre Sens. Building effective mutual exclusion services for grids. *Accepted to be published in Journal of Supercomputing, Special Issue : Secure, Manageable and Controllable Grid Services*, 2009.

[Sar44] Jean-Paul Sartre. *Huis clos.* Folio, numéro 807, Gallimard, 1944.

[SAS06] Julien Sopena, Luciana Arantes, and Pierre Sens. Performance evaluation of a fair fault-tolerant mutual exclusion algorithm. In *The 25th IEEE Symposium on Reliable Distributed Systems (SRDS 2006 - Leeds, UK)*, pages 225–234, Los Alamitos, CA, USA, October 2006. IEEE Computer Society.

[SBL09] Julien Sopena, Soheib Baarir, and Fabrice Legond. Vérification formelle d'un algorithme générique et hiérarchique d'exclusion mutuelle. *Numéro spécial du journal Technique et Science Informatique : Réseaux de Petri et algorithmes*, 28, 2009.

[Sch88] Hermann Amandus Schwarz. Ueber ein flachen kleinsten flacheninhalts betreffendes problem der variationsrechnung. *Acta Societatis scientiarum Fennicae*, XV :318, 1888.

[Sin89] Mukesh Singhal. A heuristically-aided algorithm for mutual exclusion in distributed systems. *IEEE Transactions on Computers*, 38(5) :651–662, 1989.

[Sin91] Mukesh Singhal. A class of deadlock-free meakawa-type algorithms for mutual exclusion in distributed systems. *Distributed Computing*, 4 :131–138, 1991.

[Sin92] M. Singhal. A dynamic information structure for mutual exclusion algorithm for distributed systems. *IEEE Transactions on Parallel and Distributed Systems*, 3(1) :121–125, 1992.

[Sin93] Mukesh Singhal. A taxonomy of distributed mutual exclusion. *Journal of Parallel and Distributed Computing (JPDC)*, 18(1) :94–101, 1993.

[SK85] Ichiro Suzuki and Tadao Kasami. A distributed mutual exclusion algorithm. *ACM Transactions on Computer Systems*, 3(4) :344–349, 1985.

[SLAS07] Julien Sopena, Fabrice Legond, Luciana Arantes, and Pierre Sens. A composition approach to mutual exclusion algorithms for grid applications. In *The 36th International Conference on Parallel Processing (ICPP07 - XiAn, CN)*, pages 65–75. IEEE Computer Society, September 2007.

[Var96] Moshe Y. Vardi. An automata-theoretic approach to linear temporal logic. In Faron Moller and Graham M. Birtwistle, editors, *Banff Higher Order Workshop*, volume 1043, pages 238–266. LNCS, 1996.

[vdS87] Jan L. A. van de Snepscheut. Fair mutual exclusion on a graph of processes. *Distributed Computing*, 2(2) :113–115, 1987.

[Vel93] Martin G. Velazquez. A survey of distributed mutual exclusion algorithms. Technical report cs-93-116, Colorado State University, USA, 1993.

Liste des publications de ces travaux

Revues internationales avec comité de lecture

[SALS09] Julien Sopena, Luciana Arantes, Fabrice Legond, and Pierre Sens. Building effective mutual exclusion services for grids. *Accepted to be published in Journal of Supercomputing, Special Issue : Secure, Manageable and Controllable Grid Services*, 2009.

Revues nationales avec comité de lecture

[SBL09] Julien Sopena, Soheib Baarir, and Fabrice Legond. Vérification formelle d'un algorithme générique et hiérarchique d'exclusion mutuelle. *Numéro spécial du journal Technique et Science Informatique : Réseaux de Petri et algorithmes*, Vol. 28, 2009.

Conférences internationales

[SALS08] Julien Sopena, Luciana Arantes, Fabrice Legond, and Pierre Sens. The impact of clustering on token-based mutual exclusion algorithms. In *Euro-Par 2008 Processing, Las Palmas , SPAIN*, volume 3648/2005, Heidelberg, Deutschland, August 2008. Springer Berlin.

[BSL08] Soheib Baarir, Julien Sopena, and Fabrice Legond. On the formal verification of a generic hierarchical mutual exclusion algorithm. In *The 28th International Conference on Formal Techniques for Networked and Distributed Systems (FORTE 2008 - Tokyo, Japan)*, Lecture Notes in Computer Science, June 2008. Springer.

[SLAS07] Julien Sopena, Fabrice Legond, Luciana Arantes, and Pierre Sens. A composition approach to mutual exclusion algorithms for grid applications. In *The 36th International Conference on Parallel Processing (ICPP07 - XiAn, CN)*, pages 65–75. IEEE Computer Society, September 2007.

[SAS06a] Julien Sopena, Luciana Arantes, and Pierre Sens. Performance evaluation of a fair fault-tolerant mutual exclusion algorithm. In *The 25th IEEE Symposium on Reliable Distributed Systems (SRDS 2006 - Leeds, UK)*, pages 225–234, Los Alamitos, CA, USA, October 2006. IEEE Computer Society.

[SABS05] Julien Sopena, Luciana Arantes, Marin Bertier, and Pierre Sens. A fault-tolerant token-based mutual exclusion algorithm using a dynamic tree. In *Euro-Par 2005 Processing, Lisbon , PORTUGAL*, volume 3648/2005, pages 654–663, Heidelberg, Deutschland, August 2005. Springer Berlin.

Conférences nationales

[SAS06b] Julien Sopena, Luciana Arantes, and Pierre Sens. Un algorithme équitable d'exclusion mutuelle tolérant les fautes. In *Actes de la 5ème Conférence Française sur les Systèmes d'Exploitation (CFSE'06), Chapitre français de l'ACM-SIGOPS, GDR ARP*, pages 97–107, Perpignan, France, Octobre 2006.

[SALS08b] Julien Sopena, Luciana Arantes, Fabrice Legond, and Pierre Sens. Un algorithme équitable d'exclusion mutuelle tolérant les fautes. In *Actes de la 6ème Conférence Française sur les Systèmes d'Exploitation (CFSE'08), Chapitre français de l'ACM-SIGOPS, GDR ARP*, Fribourg, Suisse, Février 2008.

Résumé

Cette thèse étudie les algorithmes distribués d'exclusion mutuelle dans le cadre des systèmes répartis à grande échelle. Elle s'intéresse plus particulièrement à la gestion des défaillances, ainsi qu'à la prise en compte des spécificités des topologies de type grille.

Dans une première partie, nous proposons un nouvel algorithme d'exclusion mutuelle tolérant aux défaillances, basé sur l'algorithme de Naimi-Tréhel. Ce nouvel algorithme conserve la complexité originale en $\mathcal{O}(log(n))$, limite l'utilisation de la diffusion et minimise le nombre des ré-émissions de requête en cas de défaillance. Des expériences, réalisées dans un cadre réel, ont permis de montrer qu'il est particulièrement bien adapté aux systèmes répartis à grande échelle.

Dans une deuxième partie, nous proposons un algorithme générique permettant de composer les algorithmes d'exclusion mutuelle de la littérature, pour prendre en compte la spécificité des grilles de calcul, à savoir : une grande différence de latence et de débit entre les réseaux locaux et le réseau inter-connectant les différentes grappes (clusters). Des expériences menées sur la grille expérimentale GRID'5000 ont, entre autre, permis de dégager des choix de composition en fonction du type d'application. Dans une autre étude de performances, nous étudions l'importance de la répartition des machines dans les différents clusters, sur l'efficacité de la composition.

Abstract

The aim of this thesis is to study mutual exclusion distributed algorithms in large scale distributed systems. It addresses both the problem of fault tolerance as well as how to exploit the physical characteristics of Grid topologies.

In the first part of the thesis, we propose a new fault tolerant mutual exclusion algorithm which is based on Naimi-Tréhel token-based algorithm. This new algorithm keeps the $\mathcal{O}(log(n))$ message complexity of the original algorithm, restrict message broadcast and minimize the number of request retransmission in case of failure. Experiments conducted in a real platform have shown that our algorithm is very suitable for large scale distributed systems.

In the second part of the thesis, we propose a general algorithm which allows the composition of different distributed mutual exclusion algorithms found in the literature by taking into account the characteristics of Grids, i.e., the difference of latency and bandwidth between a cluster local network and the network which interconnects the different clusters. Experiments conducted on top the experimental platform GRID'5000 have shown that the choice of the algorithms to compose depends on the type of the application. In a second set of experiments, we have studied the influence of the Grid architecture in the effectiveness of the composition.