Allouer efficacement des ressources dans un environnement distribué

Jonathan Lejeune [1], **Luciana Arantes** [2], **Julien Sopena** [2], **Pierre Sens** [2]

1. *Ecole des Mines de Nantes, CNRS, Inria, LINA*
   Nantes, France
   jonathan.lejeune@mines-nantes.fr
2. *Sorbonne Universités, UPMC, CNRS, Inria, LIP6*
   F-75005, Paris, France
   prénom.nom@lip6.fr

RÉSUMÉ. *Les algorithmes d'exclusion mutuelle généralisée permettent de gérer les accès concurrents des processus sur un ensemble de ressources partagées. Cependant, ils doivent assurer un accès exclusif à chaque ressource. Afin d'éviter les interblocages beaucoup de solutions reposent sur l'hypothèse forte d'une connaissance préalable des conflits entre les requêtes des processus. D'autres approches, qui ne requièrent pas une telle connaissance, utilisent un mécanisme de diffusion ou un verrou global, dégradant ainsi la complexité en messages et augmentant le coût de synchronisation. Nous proposons dans cet article un nouvel algorithme pour l'allocation de ressources partagées qui réduit les communications entre processus non conflictuels sans connaître à l'avance le graphe des conflits. Les résultats de nos évaluations de performances montrent que notre solution améliore le taux d'utilisation d'un facteur 1 à 20 comparé à un algorithme se basant sur un verrou global.*

ABSTRACT. *Generalized distributed mutual exclusion algorithms allow processes to concurrently access a set of shared resources. However, they must ensure an exclusive access to each resource. In order to avoid deadlocks, many of them are based on the strong assumption of a prior knowledge about conflicts between processes' requests. Some other approaches, which do not require such a knowledge, exploit broadcast mechanisms or a global lock, degrading message complexity and synchronization cost. We propose in this paper a new algorithm for shared resources allocation which reduces the communication between non-conflicting processes without a prior knowledge of processes conflicts. Performance evaluation results show that our solution improves resource use rate by a factor up to 20 compared to a global lock based algorithm.*

MOTS-CLÉS : *algorithme distribué, exclusion mutuelle généralisée, allocation multi-ressources, cocktail des philosophes, évaluation de performances*

KEYWORDS: *distributed algorithm, generalized mutual exclusion, multi-resource allocation, drinking philosophers, performance evaluation*

DOI:10.3166/TSI.x.1-?? © x Lavoisier

1. Introduction

Dans les applications parallèles et distribuées, les processus ont besoin d'accéder à une ou plusieurs ressources partagées de manière exclusive afin d'éviter les incohérences. Si le problème ne se résume qu'à une seule et unique ressource, il est possible alors d'utiliser un algorithme distribué d'exclusion mutuelle classique (par exemple (Lamport, 1978),(Ricart, Agrawala, 1981),(Suzuki, Kasami, 1985), (Raymond, 1989b), (Naimi, Trehel, 1987a), (Naimi, Trehel, 1987b)) pour assurer qu'à tout moment au plus un processus utilise la ressource (**propriété de sûreté**) et que toute demande d'accès à la ressource sera satisfaite en temps fini (**propriété de vivacité**). La portion de code qui accède à cette ressource est appelée section critique.

Cependant, la plupart des systèmes distribués tels que les Clouds ou les Grilles sont composés de plusieurs ressources informatiques (CPU, GPU, LAN, routeurs, ...). Les processus s'exécutant sur ces plateformes peuvent demander simultanément un accès exclusif à plusieurs de ces ressources. Ce problème est une généralisation de l'exclusion mutuelle classique: les processus commencent à exécuter leurs sections critiques respectives que lorsqu'ils ont obtenu l'ensemble des ressources demandées.

Dans ce cadre, les requêtes peuvent être conflictuelles si leurs ensembles de ressources ne sont pas disjoints. Il est possible alors de définir un graphe des conflits où les nœuds correspondent aux processus du système et où une arête modélise le partage d'une ressource entre deux nœuds. On peut alors introduire une troisième propriété appelée **propriété de concurrence** assurant que deux processus non conflictuels peuvent exécuter leur section critique simultanément.

Assurer un accès exclusif à chaque ressource du système n'est pas suffisant pour garantir la vivacité globale : des interblocages peuvent se produire. Pour le problème de l'exclusion mutuelle généralisée, ceci arrive par exemple lorsque deux processus sont chacun en train d'attendre la libération d'une ressource verrouillée par l'autre.

Ce problème multi-ressource, aussi appelé « AND-synchronisation » a été introduit par Dijkstra (Dijkstra, 1971) avec le problème du « Dîner des philosophes » où les processus demandent à chaque nouvelle requête le même ensemble de ressources. Ce problème a été étendu par Chandy et Misra (Chandy, Misra, 1984) sous le nom du problème du « cocktail des philosophes » où les processus peuvent demander à chaque nouvelle requête un ensemble variable de ressources.

Dans la littérature, il existe deux familles d'algorithmes pour ce problème de multi-ressource : les algorithmes hiérarchiques (Lynch, 1981 ; Styer, Peterson, 1988) et ceux simultanés (Awerbuch, Saks, 1990 ; Bouabdallah, Laforest, 2000 ; Ginat *et al.*, 1989 ; Maddi, 1997 ; Barbosa *et al.*, 2001). Dans la première famille, un ordre total est défini au préalable sur l'ensemble des ressources du système et les processus doivent acquérir les ressources individuellement en respectant cet ordre. Dans la seconde famille, les algorithmes possèdent des mécanismes internes qui permettent d'acquérir l'ensemble des ressources requises de manière atomique.

Cependant, la plupart des solutions incrémentales supposent que le graphe des conflits est connu a priori et ne change jamais pendant l'exécution de l'algorithme. Ceci induit une hypothèse forte sur le système. D'un autre côté, il existe des solutions de la famille simultanée qui ne nécessitent aucune connaissance préalable sur le graphe de conflits. Mais, pour sérialiser les requêtes, ces solutions induisent un coût de synchronisation important amenant à une dégradation des performances en termes de taux d'utilisation des ressources et du temps d'attente moyen.

D'autres solutions utilisent un ou plusieurs coordinateurs pour ordonnancer les requêtes en évitant les interblocages mais ces solutions ne sont pas pleinement distribuées et peuvent générer des contentions sur le réseau lorsque le système est chargé. Il existe aussi d'autres algorithmes utilisant un mécanisme de diffusion, mais ceux-ci ne passent pas à l'échelle en raison de leur forte complexité en nombre de messages.

Dans cet article, nous proposons une nouvelle approche décentralisée pour le verrouillage de plusieurs ressources dans un système distribué. Notre solution ne requiert pas l'hypothèse forte de connaître a priori le graphe des conflits, n'utilise ni de mécanisme de synchronisation globale ni de mécanisme de diffusion limitant ainsi la communication inutile entre deux processus non conflictuels. De plus, notre algorithme réordonne les requêtes afin d'exploiter le plus possible le parallélisme potentiel des requêtes non conflictuelles. Ces améliorations donnent de bonnes performances en termes de taux d'utilisation des ressources et de temps de réponse moyen.

Cet article est organisé de la manière suivante. La section 2 présente les principaux algorithmes distribués de la littérature qui résolvent le problème de l'exclusion mutuelle généralisée. Un schéma général de notre solution et son implémentation sont décrits respectivement dans les sections 3 et 4. La section 5 présente l'évaluation de performances de cette implémentation en comparant celle-ci avec deux solutions existantes de la littérature. La section 6 conclut l'article.

2. État de l'art

Le problème original de l'exclusion mutuelle peut être généralisé de différentes manières :

– une ressource partagée qui peut être accédée de manière concurrente dans la même session (problème de l'exclusion mutuelle de groupe (Joung, 1998 ; Bhatt, Huang, 2010 ; Aoxueluo *et al*., 2013));

– plusieurs exemplaires de la même ressource critique (problème du k-mutex, sémaphores) (Raymond, 1989a ; Raynal, 1991 ; Srimani, Reddy, 1992 ; Naimi, 1993 ; DeMent, Srimani, 1994 ; Satyanarayanan, Muthukrishnan, 1994 ; Bulgannawar, Vaidya, 1995 ; Reddy *et al*., 2008);

– plusieurs types de ressource (le problème multi-ressource).

Dans cet article, nous nous concentrons sur la dernière approche. Dans cette section, nous décrivons les principaux algorithmes distribués multi-ressources. Ils sont classés en deux familles : famille incrémentale et famille simultanée.

2.1. Famille incrémentale

Dans cette famille chaque processus verrouille de manière incrémentale ses ressources requises suivant un ordre préalablement défini sur l'ensemble des ressources du système. Chaque verrou peut être implémenté avec un algorithme d'exclusion mutuelle classique. Cependant, une telle stratégie peut être inefficace puisqu'un effet domino [1] des attentes peut se produire. L'effet domino dégrade la propriété de concurrence et par conséquent réduit fortement le taux d'utilisation des ressources.

Pour limiter l'effet domino, Lynch (Lynch, 1981) propose de construire un graphe dual au graphe de conflit : les nœuds sont les ressources et il existe une arête entre deux nœuds si les deux ressources correspondantes sont susceptibles d'être demandées au sein d'une même requête. En coloriant ce graphe et en minimisant le nombre de couleurs, il est alors possible de définir un ordre partiel sur l'ensemble des ressources du système en définissant un ordre total sur l'ensemble des couleurs. Les processus demanderont alors les ressources dans l'ordre des couleurs associées. Ceci réduit l'effet domino et améliore l'exploitation du parallélisme.

Pour réduire le temps d'attente, Styer et Peterson (Styer, Peterson, 1988) ont proposé une solution en considérant un coloriage quelconque (de préférence optimisé) avec un mécanisme d'annulation de verrouillage : un processus peut libérer une ressource (ou une couleur) même s'il ne l'a pas encore utilisée. Ceci permet de casser dynamiquement les possibles chaînes de processus en attente causée par l'effet domino. Sommairement, un processus libère toutes ou une partie de ses ressources acquises et essaye ensuite de les réacquérir jusqu'à satisfaction de la requête.

2.2. Famille simultanée

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

Chandy et Misra (Chandy, Misra, 1984) ont décrit le problème du cocktail des philosophes où les processus (les philosophes) partagent un ensemble de ressources (les bouteilles). Ce problème est une extension du problème du dîner des philosophes où les processus partagent un ensemble de fourchettes. Contrairement au problème du dîner, où les processus demandent toujours le même sous-ensemble de ressources, i.e., les deux mêmes fourchettes, le problème du cocktail des philosophes permet au site de demander un ensemble de ressources différent à chaque nouvelle requête. Le graphe de communication correspond directement au graphe des conflits et implique de le connaître a priori. Chaque processus partage une bouteille avec chacun de ses voisins. En orientant le graphe des conflits, il en résulte un graphe de précédence. Si les circuits sont évités dans ce graphe de précédence, les interblocages ne peuvent pas se produire. Il a été montré que le problème du dîner des philosophes respecte cette

1. un processus attend des ressources qui ne sont pas en cours d'utilisation mais qui sont verrouillées par des processus qui attendent l'acquisition d'autres ressources.

acyclicité. Cependant ceci n'est pas le cas pour le problème du cocktail. Chandy et Misra ont donc adapté le problème du cocktail en utilisant les procédures du dîner : pour verrouiller un sous-ensemble de bouteilles parmi les liens incidents d'un nœud correspondant, un processus doit acquérir toutes les fourchettes de tous les voisins avant de demander les bouteilles requises. Les fourchettes peuvent être vues comme des ressources auxiliaires et sont libérées lorsque le processus a obtenu toutes ses bouteilles. La phase d'acquisition des fourchettes sert à sérialiser les requêtes de bouteilles dans le système. Cette sérialisation évite les circuits dans le graphe de précédence et supprime par conséquent les interblocages. Cependant la phase d'acquisition des fourchettes induit des coûts de synchronisation (communications inutiles entre sites non conflictuels et nombreux messages envoyés en cas de graphe complet).

Ginat et al. (Ginat *et al*., 1989) ont remplacé la phase d'acquisition des fourchettes de l'algorithme de Chandy-Misra par une horloge logique (Lamport, 1978). Lorsqu'un processus demande des ressources, il estampille sa requête par une horloge logique et envoie un message à chaque voisin concerné. À la réception d'une requête, la bouteille associée est envoyée immédiatement au demandeur si l'estampille est plus petite que l'horloge du receveur. L'association d'une horloge logique et d'un ordre total sur les identifiants des processus permet de définir un ordre total sur les requêtes évitant ainsi les interblocages. Cependant la complexité en messages devient importante si le graphe de conflit est complet car l'algorithme utilise dans ce cas un mécanisme de diffusion.

Rhee (Rhee, 1995 ; 1998) présente un ordonnanceur où chaque processus gère l'ordonnancement d'une ressource attribuée arbitrairement. Chaque processus gérant une ressource maintient une file d'attente qui peut être réordonnancée en fonction des nouvelles requêtes pendantes évitant ainsi les interblocages. Cette approche requière des gestionnaires dédiés ce qui peut induire des goulots d'étranglement. De plus le protocole de coordination permettant d'éviter les interblocages entre les gestionnaires est coûteux.

Maddi (Maddi, 1997) a proposé un algorithme basé sur un mécanisme de diffusion. Chaque ressource est représentée par un unique jeton. À chaque demande, un processus diffuse aux autres un message de requête estampillé par une horloge logique. À sa réception, la requête est stockée dans une file locale triée selon les estampilles temporelles. Cet algorithme peut être vu comme plusieurs instances de l'algorithme d'exclusion mutuelle classique de Suzuki-Kasami (Suzuki, Kasami, 1985), présentant ainsi une complexité en messages importante.

L'algorithme de Bouabdallah-Laforest (Bouabdallah, Laforest, 2000) est plus détaillé car nous le comparons avec notre solution dans la section 5. Chaque ressource est représentée par un unique *jeton de ressource* et est associée à une file d'attente distribuée dont le premier élément est le possesseur du jeton. Pour avoir le droit d'accéder à une ressource, un processus doit posséder le *jeton de ressource* associé. Avant de demander un ensemble de ressources, le processus doit d'abord obtenir un *jeton de contrôle* qui est unique dans tout le système. L'algorithme d'exclusion mutuelle gérant ce jeton de contrôle est l'algorithme de Naimi-Tréhel (Naimi, Trehel, 1987a). Cet algorithme maintient d'une part une topologie logique d'arbre dynamique tel que la

racine soit toujours le dernier demandeur du jeton de contrôle, et d'autre part une file d'attente distribuée des requêtes pendantes. Un *jeton de ressource* peut être possédé par un processus ou être directement stocké dans le *jeton de contrôle* si la ressource n'est pas utilisée. Le *jeton de contrôle* contient un vecteur de M entrées (M est égal au nombre de ressources du système). Si un jeton n'est pas dans le *jeton de contrôle*, l'entrée correspondante du vecteur indique l'identifiant du dernier processus ayant demandé ce jeton. Ainsi, lorsqu'un site reçoit le *jeton de contrôle*, il prend des ressources requises inutilisées dans le *jeton de contrôle* et envoie pour chaque jeton manquant un message INQUIRE au dernier demandeur. Le *jeton de contrôle* permet de sérialiser les requêtes ce qui assure qu'une requête sera enregistrée de manière atomique dans les différentes files d'attente distribuées. Ainsi, aucun cycle n'est possible dans l'union des files. Cet algorithme a une bonne complexité en messages ($\mathcal{O}(Log(N))$) mais le *jeton de contrôle* induit un important goulot d'étranglement quand il y a peu de conflits, i.e., dans un scénario où la concurrence est potentiellement grande.

2.3. Synthèse de l'état de l'art

La plupart des solutions de la littérature supposent que le graphe des conflits est connu a priori et ne change jamais pendant l'exécution de l'algorithme ce qui induit une hypothèse forte et irréaliste sur l'application. Ces solutions peuvent néanmoins fonctionner sans connaissance préalable du graphe de conflits en considérant de façon pessimiste un graphe complet. Cependant une telle considération induit un coût de synchronisation élevé et dégrade le taux d'utilisation des ressources :

– Les algorithmes de la famille incrémentale sont très pénalisés par un effet domino inévitable des attentes car il est impossible de colorier de manière optimale un graphe complet (théorème de Brooks (Brooks, 1987), théorie des graphes). De plus, l'ordre défini sur l'ensemble des couleurs est arbitraire et limite les optimisations possibles.

– Les algorithmes de la famille simultanée font communiquer des processus qui ne rentrent pas en conflit et qui n'ont donc aucune raison d'interagir entre eux. Les algorithmes du cocktail des philosophes de Chandy-Misra (Chandy, Misra, 1984) et de Ginat-Shankar-Agrawala (Ginat *et al*., 1989) utilisent dans le cas d'un graphe complet, un algorithme de diffusion (acquisition des ressources auxiliaires pour l'un et enregistrement des requêtes pour l'autre) impliquant en plus d'un coût de synchronisation élevé, une forte complexité en messages. Enfin, l'algorithme de Bouabdallah-Laforest (Bouabdallah, Laforest, 2000) utilise une section critique de contrôle (verrou global) impliquant un goulot d'étranglement.

3. Schéma général de la solution

3.1. Définitions et hypothèses

Nous considérons un système distribué composé d'un ensemble de N sites fiables $\Pi = \{s_1, s_2, ...s_N\}$ et un ensemble de M ressources, $\mathcal{R} = \{r_1, r_2, ..., r_M\}$. Il y a un seul processus par site ou nœud. Par conséquent les mots « processus », « nœud » et « site » sont interchangeables.

Les processus ne partagent pas de mémoire et communiquent uniquement par passage de messages. Les nœuds sont supposés être connectés par des liens de communications point à point FIFO et fiables (ni perte ni duplication de message). Le graphe de communication est complet, *i.e.*, tout processus peut communiquer avec n'importe quel autre processus.

Un processus peut initier une nouvelle requête si et seulement si sa demande précédente a été satisfaite. Par conséquent, il y a au plus N requêtes pendantes dans le système. Nous supposons que le graphe de conflits est inconnu. Le temps de section critique est supposé fini.

3.2. Objectifs

Puisque l'on considère un graphe de conflits inconnu, il est difficile de s'appuyer sur un algorithme de famille incrémentale (cf. section 2.1) car leurs performances dépendent des techniques de coloration de graphe. Par conséquent, notre algorithme appartient à la famille simultanée (cf. section 2.2).

Ces algorithmes ont un point commun : ils possèdent un mécanisme permettant d'ordonner totalement les requêtes du système évitant les cycles dans le graphe de précédence et par conséquent les interblocages. Ce mécanisme associe un identifiant unique à toute requête quel que soit son ensemble de ressources. En définissant un ordre total sur ces identifiants, toute requête est différentiable d'une autre et permet ainsi un ordonnancement sans interblocage. Chandy-Misra (Chandy, Misra, 1984) utilisent un algorithme de « Dîner des philosophes », (Ginat *et al.*, 1989) et (Maddi, 1997) utilisent des horloges logiques et Bouabdallah-Laforest (Bouabdallah, Laforest, 2000) utilisent une section critique de contrôle. Cependant, l'utilisation d'horloge logique est généralement associée à un mécanisme de diffusion impliquant une forte complexité en messages, ce qui est problématique pour un passage à l'échelle. Les solutions de Chandy-Misra et de Bouabdallah-Laforest utilisent une section critique de contrôle : pour Chandy-Misra, le « dîner » dans un graphe complet est équivalent à l'algorithme d'exclusion mutuelle classique de Ricart-Agrawala (Ricart, Agrawala, 1981), et l'algorithme de Bouabdallah-Laforest utilise l'algorithme de Naimi-Tréhel (Naimi, Trehel, 1987a). L'algorithme de Ricart-Agrawala est basé sur des permissions et utilise aussi un mécanisme de diffusion (complexité en message de $\mathcal{O}(2N-1)$) alors que l'algorithme de Naimi-Tréhel est basé sur la circulation d'un jeton dans un arbre dynamique (complexité en message de $\mathcal{O}(log(N))$). L'algorithme de Bouabdallah-Laforest apparaît aujourd'hui comme le plus efficace. Bien que cet algorithme ait une bonne complexité en messages, il possède néanmoins deux limites qui dégradent le taux d'utilisation des ressources:

– deux processus non conflictuels doivent communiquer ensemble par le biais du jeton de contrôle impliquant un surcoût en synchronisation;

– l'ordonnancement des requêtes est statique : il dépend uniquement de l'ordre d'acquisition de la première étape (cette remarque est aussi valable pour les algorithmes avec horloge logique). En effet, il est impossible de revenir sur l'ordre de verrouillage. Une requête ne peut pas préempter une autre requête qui a eu le jeton de contrôle avant ce qui peut être problématique si on souhaite changer l'ordonnancement

dynamiquement.

Nous avons donc deux objectifs :

– éviter l'utilisation d'un verrou global pour éviter la communication entre deux processus non conflictuels;

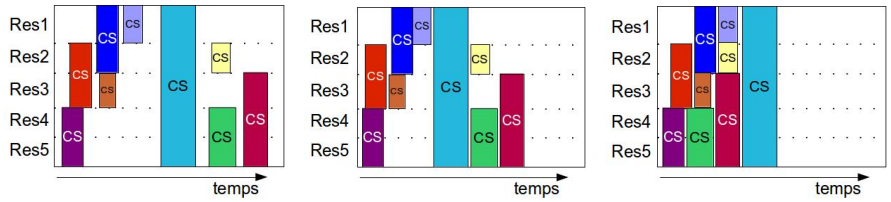
– pouvoir ordonnancer dynamiquement les requêtes.

La figure 1 illustre sous la forme d'un diagramme de Gantt l'impact de nos deux objectifs sur le taux d'utilisation des ressources comparé à l'algorithme de Bouabdallah-Laforest dans un système à 5 ressources :

– la suppression du verrou global permet de réduire le temps entre deux sections critiques conflictuelles successives (Figure 1(b));

– le mécanisme d'ordonnancement dynamique permet de donner l'accès aux ressources à des processus dans les intervalles de temps où les ressources ne seraient pas utilisées dans le cas d'un ordonnancement statique (Figure 1(c)).

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



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

FIGURE *1. Illustration de l'impact des objectifs sur le taux d'utilisation*

3.3. Suppression du verrou global

Nous décrivons ci-après notre mécanisme qui sérialise les requêtes sans utiliser de verrou global.

3.3.1. Mécanisme de compteurs

Le but du jeton de contrôle dans l'algorithme de Bouabdallah-Laforest est de donner un ordre de passage commun sur l'ensemble des files d'attente associées aux ressources. Pour supprimer le verrou global, nous proposons d'utiliser un compteur par ressource. Chaque compteur donne un ordre de passage par ressource pour chacune des requêtes. Le système maintient donc M compteurs à accès exclusif. Dans notre cas, chaque compteur est associé à un jeton dans lequel on stocke sa valeur.

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

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

3.3.2. Ordonnancement total des requêtes

Une requête req_i émise par le site $s_i \in \Pi$ pour une ressource donnée est associée à deux informations : le site initiateur de la requête s_i et le vecteur associé $v_i \in \mathbb{N}^M$. Les interblocages sont évités si nous définissons un ordre total sur les requêtes. Nous allons d'abord utiliser un ordre partiel sur l'ensemble des vecteurs en définissant une fonction $\mathcal{A} : \mathbb{N}^M \rightarrow \mathbb{R}$ qui transforme un vecteur d'entiers en un réel.

Puisque $\mathcal{A}$ donne un ordre partiel (deux vecteurs peuvent avoir le même réel résultant), nous devons utiliser un ordre total arbitraire $\prec$ sur les identifiants des sites pour ordonner totalement les requêtes. L'ordre total des requêtes est noté $\lhd$ où $req_i \lhd req_j$ ssi $\mathcal{A}(v_i) < \mathcal{A}(v_j) \vee (\mathcal{A}(v_i) = \mathcal{A}(v_j) \wedge s_i \prec s_j)$. Ainsi, en cas de valeur égale par $\mathcal{A}$, le site avec le plus petit identifiant sera le plus prioritaire. Bien que ce mécanisme évite les interblocages en assurant que toutes les requêtes sont différentiables, le respect total de la propriété de vivacité dépend aussi de la définition de $\mathcal{A}$. En effet, la famine est évitée si la définition de $\mathcal{A}$ assure que toute requête sera dans un temps fini la plus petite dans l'ordre $\lhd$.

La fonction $\mathcal{A}$ permet de définir une heuristique donnant une politique d'ordonnancement sur les requêtes. C'est un paramètre de l'algorithme qui peut permettre de favoriser les requêtes en fonction du nombre de ressources utilisées. Notons que si $\mathcal{A}$ est bien choisie, les cas d'égalité seront peu probables. L'utilisation de l'identifiant du site ne posera donc pas de problème d'équité.

3.4. Ordonnancement dynamique

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

L'idée de l'ordonnancement dynamique serait de limiter la détention des ressources aux seules sections critiques, en offrant la possibilité de prêter des ressources inutilement possédées.

L'introduction de « prêt » n'est cependant pas triviale, puisqu'elle remet en cause la propriété de vivacité : une ressource acquise puis prêtée ne permet plus d'entrer en section critique. Pour assurer la vivacité, le mécanisme de « prêt » doit assurer qu'un site finisse à terme par disposer simultanément de ses ressources précédemment acquises. Il devra ainsi éviter la famine due à l'éparpillement des ressources et les interblocages.

3.4.1. Éviter la famine

Puisque le prêt de jeton n'assure pas nécessairement que le processus emprunteur obtiendra l'ensemble des ressources requises, une famine peut apparaître. Pour pallier ce problème, nous proposons donc un mécanisme simple en restreignant le prêt à un seul site à la fois. Le processus prêteur est ainsi assuré de réacquérir et en temps fini toutes les ressources prêtées puisque le temps de section critique de l'emprunteur est supposé fini par hypothèse.

3.4.2. Éviter les interblocages

Le fait d'emprunter des ressources à plusieurs sites peut induire des cycles dans les files d'attente ce qui conduit à des interblocages. Nous proposons donc un mécanisme simple en restreignant l'emprunt à un seul site uniquement s'il possède toutes les ressources manquantes. Un site rentrera directement en section critique à la réception des ressources prêtées.

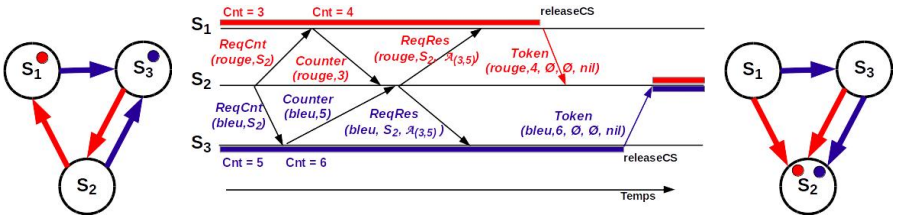
4. Implémentation

Dans cette section nous décrivons l'implémentation de notre algorithme. Par manque de place, nous ne présenterons pas son pseudo-code.

Chaque ressource est associée à un jeton unique qui contient le compteur de la ressource. Le détenteur d'un jeton peut ainsi exclusivement utiliser la ressource associée et incrémenter la valeur du compteur.

Chaque jeton est géré par un mécanisme qui s'inspire d'une version simplifiée de l'algorithme de Mueller (Mueller, 1999). Ce dernier est un algorithme distribué d'exclusion mutuelle à priorité et est basé sur la circulation d'un jeton dans une topologie logique. Cette topologie est un arbre dynamique où la racine de l'arbre est le détenteur du jeton de la ressource correspondante. Chaque jeton maintient une file d'attente des requêtes pendantes demandant la ressource contrôlée.

La figure 2 donne un exemple où nous considérons 3 processus (s_1, s_2, et s_3) et 2 ressources (r_{rouge} et r_{bleu}). La figure 2(a) montre la topologie initiale relative à chaque ressource où s_1 et s_3 possèdent respectivement les jetons associés à r_{rouge} et r_{bleu}. Le processus s_2 a deux pères, s_1 (arbre rouge) et s_3 (arbre bleu), tandis que le site s_1 (respectivement le site s_2) a un seul père s_3 (respectivements_2) associé à l'arbre bleu (respectivement rouge).Dans la figure 2(c), les topologies des arbres ont



(a) État initial (b) Exécution (c) État final

FIGURE *2. Exemple d'exécution*

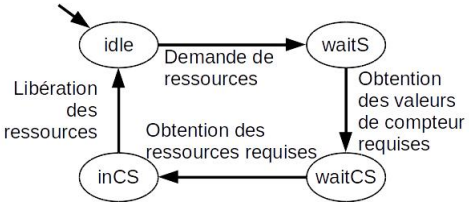


FIGURE *3. Machine à état d'un processus*

changé. Le site s_2 a obtenu les deux jetons, il est par conséquent devenu la nouvelle racine des deux arbres.

Le choix d'un algorithme à priorité comme celui de Mueller permet de réordonnancer les requêtes pendantes d'une file d'attente d'une ressource donnée à chaque fois qu'une requête de plus haute priorité, d'après la relation d'ordre $\lhd$, est reçue.

4.1. États des processus

Un processus peut être dans un des quatre états suivants :

– $Idle$: le processus ne demande aucune ressource;

– $waitS$: le processus attend les valeurs des compteurs requis;

– $waitCS$: le processus attend les ressources requises;

– $inCS$: le processus utilise les ressources requises (section critique).

La figure 3 montre la machine à état d'un processus.

4.2. Messages

Nous définissons cinq types de message :

– $ReqCnt(r, s_{init})$: envoyé par le processus s_{init} lorsqu'il demande la valeur courante du compteur associé à la ressource r;

– $Counter(r, val)$: envoyé par le détenteur du jeton associé à r pour répondre à une requête $ReqCnt$. Il contient la valeur val du compteur associé à r que le porteur du jeton a assigné à la requête en question;

– $ReqRes(r, s_{init}, mark)$: envoyé par s_{init} lorsqu'il demande le jeton pour accéder à la ressource r. Cette requête est taguée avec $mark$ qui correspond à la valeur retournée par la fonction $\mathcal{A}$;

– $ReqLoan(r, s_{init}, mark, missingRes)$: envoyée par s_{init} lorsqu'il demande un prêt pour la ressource r, tagué par la valeur $mark$ (retour de la fonction $\mathcal{A}$). Ce message contient également l'ensemble des ressources manquantes $missingRes$ que s_{init} attend;

– $Token(r, counter, wQueue, wLoan, s_{lender})$: message de jeton associé à la ressource r qui contient la valeur courante du compteur associé et la file d'attente $wQueue$ des requêtes pendantes classée dans l'ordre croissant de leur valeur $marks$ respective. La file $wLoan$ contient l'ensemble des requêtes de prêt concernant r. Si l'envoi du jeton concerne un prêt, s_{lender} correspond à l'identifiant du site prêteur.

Les messages de requête (types $ReqCnt$, $ReqRes$, et $ReqLoan$) sont retransmis à partir du site s_{init} jusqu'au nœud racine (porteur du jeton) en respectant la structure d'arbre correspondante alors que les messages de type $Counter$ et $Token$ sont envoyés directement au demandeur.

Le fait que la topologie d'arbre puisse changer dynamiquement pendant la retransmission d'un message de requête pose deux problèmes. En effet ceci peut conduire à (1) des cycles impliquant des retransmissions infinies et (2) une famine puisqu'un message de requête $mess$ peut ne jamais être pris en compte par le jeton si ce dernier est envoyé sur un site déjà visité par $mess$. Pour éviter le premier problème, chaque message de requête collectionne au cours de sa retransmission, les identifiants des sites visités. Pour le second problème, chaque site maintient un historique des messages de requêtes reçus, dans lequel les messages obsolètes (déjà pris en compte par le jeton) sont effacé grâce à un mécanisme d'horloge logique.

Afin d'économiser des envois de messages, il est possible d'agréger des messages de même type t adressés à un même site en un seul message de type t.

4.3. Implémentation du mécanisme de compteur

Lorsqu'un processus s_i souhaite accéder à un ensemble de ressources, il change son état de $Idle$ à $waitS$. Il doit ensuite obtenir les valeurs courantes des compteurs associés aux ressources requises. Si s_i possède déjà des jetons associés à des ressources requises, il sauvegarde pour sa requête la valeur courante des compteurs respectifs et les incrémente. Rappelons que seul le porteur d'un jeton, (le cas échéant s_i) peut incrémenter le compteur associé à la ressource en question. Par ailleurs, pour chaque valeur de compteur manquante pour les jetons non possédés, s_i envoie un message $ReqCnt$ message à l'un de ses pères : celui de l'arbre correspondant à la ressource en question. De plus, il enregistre dans une variable locale $CntNeeded$ les identifiants des ressources pour lesquels la valeur du compteur associé est encore inconnue.

Lorsqu'un site s_j reçoit un message de requête $ReqCnt$ d'un site s_i, s'il ne possède pas le jeton associé à r, il retransmet le message à son père appartenant à l'arbre relatif à r. À l'inverse, si s_j est le porteur du jeton mais ne requiert pas r, il envoie directement le jeton à s_i. Dans le cas contraire, s_j garde le jeton et envoie un message $Counter$, qui contient la valeur courante du compteur, à s_i et ensuite, incrémente le compteur.

À la réception d'un message $Counter$ pour une ressource r, s_i retire r de l'ensemble $CntNeeded$. Quand $CntNeeded$ devient vide, s_i a donc obtenu toutes les valeurs de compteur requises. Ces valeurs sont uniquement assignée à la requête de s_i. Il peut alors, dans ce cas, passer à l'état $waitCS$ et pour chaque jeton de ressource requise non possédé, il envoie un message $ReqRes$ au père respectif.

De façon similaire à un message $ReqCnt$, à la réception d'un message $ReqRes$ pour une ressource r, le processus s_j retransmet le message au père correspondant s'il ne possède pas le jeton associé àr. Si s_j possède le jeton et ne requiert pas r ou bien s'il est dans l'état $waitS$, il envoie le jeton directement à s_i. Dans le cas contraire, s_i et s_j sont conflictuels et il est nécessaire de décider qui des deux doit posséder la ressource r en priorité. Si s_j est en section critique (état $inCS$) ou si la priorité de sa requête est plus grande que celle de s_i ($req_j \lhd req_i$), il garde le jeton. Dans ce cas, la requête de s_i est stockée dans la file d'attente du jeton de r ($wQueue$). Sinon, s_j doit céder le jeton de r à s_i. À cet effet, s_j enregistre sa propre requête dans la file d'attente du jeton de r ($wQueue$) et envoie le jeton directement à s_i, i.e., un message $Token$.

Quand s_i reçoit un message $Token$ associé à la ressource r, il réalise deux ensembles de mises à jour : (1) il inclut r dans l'ensemble des jetons possédés. Si r est dans $CntNeeded$, i.e., s_i n'a pas encore reçu la valeur du compteur associé à r, il sauvegarde la valeur du compteur, l'incrémente et efface r de $CntNeeded$; (2) ensuite s_i prend en compte les messages de requête pendant concernant r dans l'historique local : il répond par un message $Counter$ à chaque site qui a émis un message $ReqCnt$ et ajoute dans $wQueue$ (respectivement $wLoan$) du jeton les informations associées à chaque message $ReqRes$ (resp. $ReqLoan$). Le processus s_i peut alors entrer en section critique (état $inCS$) si il possède tous les jetons des ressources requises. Si tel n'est pas le cas, il change son état à $waitCS$ si son ensemble $CntNeeded$ est devenu vide (i.e., s_i a obtenu toutes les valeurs de compteur requises). Dans ce cas, s_i envoie des messages $ReqRes$ pour chaque ressource manquante. À cause du deuxième ensemble de mises à jour, le site s_i doit s'assurer que sa requête est toujours la plus prioritaire dans $wQueue$ d'après la relation d'ordre $\lhd$. Si cela n'est pas le cas, le jeton est cédé au site ayant la plus haute priorité. À ce stade, nous sommes certains que s_i est le porteur légitime de l'ensemble des jetons qu'il possède. Il peut alors potentiellement satisfaire des requêtes de prêt stockées dans les files $wLoan$ associées aux jetons qu'il possède. Enfin, s_i peut éventuellement ensuite initier une requête de prêt si nécessaire (voir section 4.4).

Lorsqu'un site sort de la section critique, il dépile le premier élément de la file d'attente de l'ensemble des jetons possédés et les envoie à leur prochain porteur. Finalement s_i passe à l'état $Idle$.

Prenons l'exemple de la figure 2 avec 3 processus (s_1, s_2 et s_3) et 2 ressources (r_{rouge} et r_{bleu}), où la configuration initiale est donnée figure 2(a), que nous avons précédemment décrite. Les processus s_1 et s_3 sont en section critique utilisant r_{rouge} et r_{bleu} respectivement. La figure 2(b) montre les messages échangés entre les processus quand s_2 demande l'accès aux deux ressources. Dans un premier temps, s_2 envoie à chacun de ses pères, s_1 (arbre rouge) et s_3 (arbre bleu), un message de requête $ReqCnt$ pour obtenir la valeur courante des compteurs associés. Quand s_2 a reçu les deux valeurs de compteur, il envoie des messages de requête $ReqRes$ se propageant sur les arbres pour demander les jetons. Lorsque s_1 et s_3 sortent de section critique, ils envoient respectivement les jetons r_{rouge} et r_{bleu} à s_2, qui peut entrer en section critique une fois qu'il a reçu les deux jetons. La figure 2(c) montre la configuration finale des arbres logiques quand s_2 est en section critique.

4.4. Le mécanisme de prêt

L'exécution d'un prêt dépend de quelques conditions relatives au site prêteur et au site emprunteur :

– À la réception du jeton un processus s_i peut demander un prêt à condition qu'il soit dans l'état $waitCS$ (i.e., il a obtenu toutes les valeurs de compteur requises) et le nombre de ressources requises manquantes est plus petit ou égal à un seuil donné. Si tel est le cas, s_i envoie un message $ReqLoan$ aux pères associés aux ressources manquantes. De la même manière qu'un message $ReqRes$, un message $ReqLoan$ pour une ressource est retransmis jusqu'au porteur du jeton associé à la ressource en question.

– À la réception d'un message $ReqLoan$ pour le site s_i pour une ressource r, le porteur du jeton s_j doit d'abord vérifier si le prêt est possible. Tous les jetons requis dans le message (ensemble $missingRes$) peuvent être prêtés seulement si toutes les conditions suivantes sont respectées :

- s_j possède toutes les ressources manquantes à s_i (l'ensemble $missingRes$ indiqué dans le message $ReqLoan$);
- aucune des ressources possédées par s_j n'est un prêt;
- s_j n'a prêté aucune ressource à un autre site;
- s_j n'est pas en section critique;
- la requête de s_i a une plus haute priorité que la requête de s_j si les deux ont envoyé une requête de prêt pour leur requête courante de section critique

Si le prêt est possible, les jetons associés aux ressources en question sont envoyés à s_i avec s_{lender} égal à s_j. Sinon, si s_j ne requiert pas la ressource r de la requête de prêt ou si s_j est dans l'état $waitS$, il cède directement le jeton associé à r à s_i. Sinon, la requête de prêt est enregistrée dans les $wLoan$ des jetons correspondants pour être potentiellement satisfaite plus tard à la réception de nouveaux jetons.

Si s_i ne peut toujours pas entrer en section critique à la réception des jetons empruntés (si pendant ce temps il a cédé d'autres jetons pour des requêtes plus prioritaires), alors la requête de prêt est annulée. Par conséquent, s_i renvoie immédiatement

les jetons empruntés à s_{lender}. Ceci évite des états incohérents où un site posséderait des jetons empruntés mais non utilisés.

À la sortie de section critique, s_i renverra les jetons empruntés directement à s_j.

4.5. Optimisations

4.5.1. Réduction des coûts de synchronisation des requêtes à une seule ressource

Il est possible de réduire les coûts de synchronisation des requêtes requérant une seule ressource en passant directement l'état du processus demandeur de $Idle$ à $waitCS$. Puisque ces requêtes ne requièrent qu'un seul compteur, stocké dans le jeton, le site racine de l'arbre correspondant est en mesure d'appliquer la fonction $\mathcal{A}$ et de considérer le message $ReqCnt$ comme un message $ReqRes$. Ainsi, une telle optimisation réduit l'échange de messages.

4.5.2. Économie de messages $ReqRes$

Une fois qu'un processus s_i a obtenu l'ensemble des valeurs de compteur requises il envoie pour chaque ressource, un message $ReqRes$ qui transitera dans l'arbre correspondant jusqu'à atteindre le possesseur du jeton (site racine). Le nombre de retransmissions de ces messages peut être réduit par :

– la réduction de la longueur du chemin en nombre de sites intermédiaires entre le site demandeur s_i et le site racine s_j : à la réception d'un message $Counter$ de s_j, le processus s_i prend s_j comme père puisqu'il est le porteur de jeton le plus récent du point de vue de s_i.

– la limitation des retransmissions avant que le message n'atteigne le site racine. À la réception d'un message $ReqRes$ pour une ressource r, un processus s_j ne retransmet pas ce message si (1) il est dans l'état $waitCS$, requiert également r et que sa requête a une plus grande précédence que la requête de s_i ou (2) s_j a prêté le jeton associé à r. Si une de ces conditions est respectée, s_j sait qu'il obtiendra le jeton correspondant à r avant s_i. La requête de s_i sera donc à terme enregistrée dans la file d'attente $wQueue$ du jeton.

5. Évaluation de performances

Dans cette section, nous présentons des résultats expérimentaux en comparant notre solution à deux algorithmes de l'état de l'art :

– Un algorithme incrémental qui utilise M instances de l'algorithme de Naimi-Tréhel (Naimi, Trehel, 1987b), un algorithme d'exclusion mutuelle efficace grâce à sa complexité en messages comprise entre $\mathcal{O}(LogN)$ et $\mathcal{O}(1)$;

– L'algorithme de **Bouabdallah-Laforest** (Bouabdallah, Laforest, 2000) (voir section 2).

Afin de montrer l'impact du mécanisme de prêt, nous considérons deux versions de notre algorithme appelées **Avec prêt** et **Sans prêt** qui correspondent respectivement à l'activation et la désactivation du mécanisme de prêt. Lorsque ce mécanisme est activé, un site peut initier une requête de prêt lorsqu'il lui manque seulement une

seule ressource. Nous nous intéressons à l'évaluation des deux métriques suivantes : (1) le taux d'utilisation des ressources et (2) le temps d'attente pour obtenir le droit d'utiliser les ressources requises, i.e., entrer en section critique.

Comme nous avons pu l'expliquer précédemment, notre algorithme requiert la définition d'une fonction paramètre $\mathcal{A}$. Pour cette évaluation de performances, nous avons choisi une fonction $\mathcal{A}$ qui calcule la moyenne des valeurs non nulles du vecteur de compteur. Cette définition évite la famine car les valeurs de compteurs augmentent à chaque nouvelle requête émise impliquant que la valeur minimum retournée par $\mathcal{A}$ augmente à chaque nouvelle requête. Ainsi la propriété de vivacité est ainsi toujours respectée [2]. L'avantage du mécanisme de la fonction $\mathcal{A}$ réside dans le fait que la famine est évitée seulement grâce à un calcul qui n'implique aucun surcoût en communication.

5.1. Plate-forme et paramètres d'expérimentation

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

Une expérience est caractérisée par :

– N : le nombre de processus (32 dans notre cas);

– M : le nombre total de ressources dans le système (80 dans notre cas);

– α : le temps d'exécution de la section critique (quatre valeurs possibles : 5 ms, 15 ms, 25 ms et 35 ms selon le nombre de ressources demandées);

– β : l'intervalle de temps entre le moment où un processus libère la section critique et le moment où il la redemande;

– γ : la latence réseau pour envoyer un message entre deux processus (environ 0,6 ms dans notre cas);

– ρ : le rapport $\beta/(\alpha + \gamma)$, qui exprime la fréquence à laquelle la section critique est demandée. La valeur de ce paramètre est inversement proportionnelle à la charge en requêtes : une valeur basse donne une charge haute et vice-versa;

– $SizeReq$: le nombre maximum de ressources qu'un site peut demander. Ce paramètre est compris entre 1 et M. Plus la valeur de ce paramètre augmente, plus le parallélisme potentiel de l'application diminue impliquant également une augmentation de la probabilité d'avoir des requêtes conflictuelles.

À chaque nouvelle requête, un processus choisit x ressources uniformément entre 1 et $SizeReq$. Le temps de section critique (paramètre α) de la requête dépend alors de la valeur de x : plus cette valeur est grande et plus le temps de section critique risque

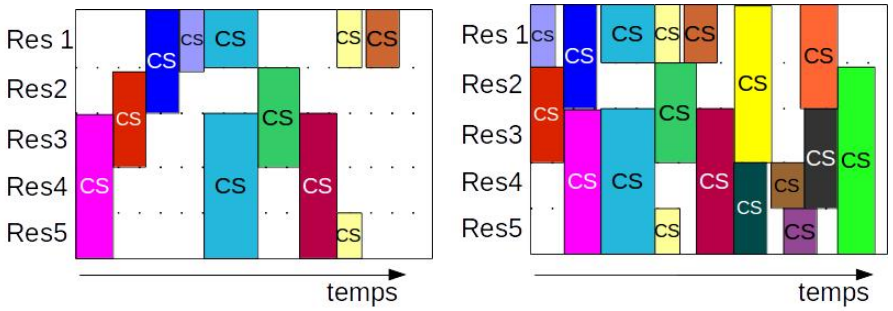
2. D'autres fonctions sont envisageables, on trouvera par exemple dans (Lejeune, 2014), l'étude d'une fonction qui calcule la moyenne sur l'ensemble de valeurs.

d'être grand car nous considérons qu'une requête demandant beaucoup de ressources doit en pratique avoir un plus grand temps de calcul en section critique.

La charge du système est caractérisée par le nombre de requêtes pendantes. Ce nombre dépend de la taille des requêtes et de la fréquence des demandes. Pour simuler une forte charge (beaucoup de requêtes pendantes) et une charge moyenne (quelques requêtes pendantes), nous avons fixé respectivement ρ à $0.1\frac{N}{M}$ et $30\frac{N}{M}$.

5.2. Taux d'utilisation des ressources

La métrique principale pour l'évaluation des algorithmes est le taux d'utilisation. Cette métrique globale est le pourcentage de temps où les ressources sont utilisées (100% signifie que toutes les ressources ont été sans cesse utilisées durant l'expérience). On peut la voir comme le pourcentage de l'aire colorée des diagrammes de la figure 4. Ainsi, la figure 4(a) donne un exemple d'exécution où les ressources ne sont pas utilisées efficacement alors que la figure 4(b) illustre une exécution avec un meilleur taux d'utilisation (zone blanche moins importante). Dans nos évaluations, le temps d'une expérience est égal à 30 secondes.

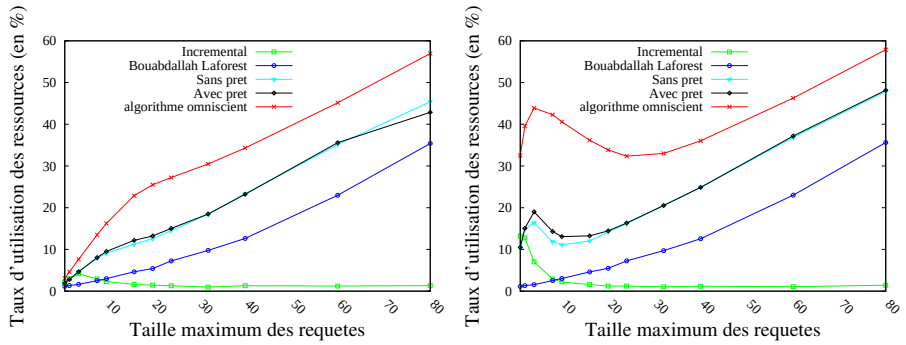


(a) Exécution inefficace (b) Exécution efficace

FIGURE *4. Illustration du taux d'utilisation pour l'exclusion mutuelle généralisée*

En faisant varier $SizeReq$, nous montrons dans la figure 5, l'impact du nombre de ressources demandées dans une requête, appelé *taille de requête* sur le taux d'utilisation des ressources dans le cas d'une charge moyenne (figure 5(a)) et haute (figure 5(b)). La taille de requête x peut être différente à chaque nouvelle requête et est choisie en fonction d'une loi aléatoire uniforme entre 1 et $SizeReq$.

En plus des algorithmes considérés, nous avons inclus dans ces deux figures une courbe témoin représentant un algorithme en mémoire partagée (*algorithme omniscients*) ayant une connaissance globale des nouvelles requêtes émises et un coût de synchronisation nul. Cette connaissance globale lui permet d'ordonnancer les requêtes afin d'utiliser au mieux les ressources. Ainsi, l'écart avec cette courbe donnera le coût de synchronisation des algorithmes.



(a) Charge moyenne (b) Charge haute

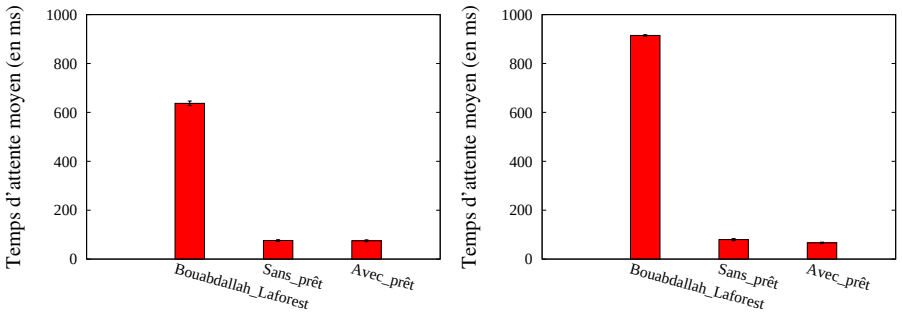
FIGURE *5. Impact de la taille des requêtes sur le taux d'utilisation*

Globalement, dans les deux figures, lorsque $SizeReq$ augmente, le taux d'utilisation augmente également. Lorsque la taille de requête est minimale, le nombre maximal de ressources qui peuvent être utilisées est égal au nombre de processus N qui est plus petit que le nombre total de ressource M. D'autre part, lorsque la taille moyenne des requêtes augmente, chaque exécution de section critique concerne un plus grand nombre de ressources qui implique donc une augmentation du taux d'utilisation.

Nous remarquons que dans le scénario à haute charge (Figure 5(b)), la forme de la courbe de l'algorithme en mémoire partagée, augmente dans un premier temps, puis diminue jusqu'à atteindre une valeur seuil ($SizeReq = 20$) et augmente de nouveau par la suite. Dans la première phase, la probabilité d'avoir des conflits est faible comparée à la taille des requêtes et la différence entre N et M. Après $SizeReq = 4$, le nombre de conflits commence à augmenter et la chute de la courbe qui suit est une conséquence de la sérialisation des requêtes conflictuelles. Enfin, lorsque $SizeReq$ est plus grand que 20, la probabilité d'avoir des conflits est maximum mais chaque accès à une section critique requiert un nombre plus important de ressources ce qui améliore le taux d'utilisation global. Par conséquent, l'ultime augmentation de la courbe n'est pas causée par l'augmentation de la concurrence des requêtes non conflictuelles mais par l'augmentation de la taille moyenne des requêtes.

Nous pouvons également remarquer que lorsque la taille moyenne des requêtes augmente, les comportements des algorithmes sont différents. Pour l'algorithme incrémental, le taux d'utilisation diminue et se stabilise car cet algorithme ne bénéficie pas de l'augmentation de la taille moyenne des requêtes à cause de l'effet domino (voir section 2.1).

Le taux d'utilisation de l'algorithme de Bouabdallah-Laforest augmente régulièrement. Bien que cet algorithme soit très désavantagé par son verrou global lorsqu'il y a peu de conflits (en particulier en haute charge), son taux d'utilisation augmente avec la taille des requêtes. Nous observons que dans cet algorithme, le taux d'utilisation augmente plus vite que l'algorithme incrémental car il exploite davantage la concur-



(a) Charge moyenne (b) Charge haute

FIGURE *6. Temps d'attente moyen($SizeReq = 4$)*

rence entre requêtes non conflictuelles. Cependant, cette augmentation n'est pas aussi efficace que celle de nos algorithmes : indépendamment de la taille des requêtes, ces derniers présentent un meilleur taux d'utilisation que l'algorithme de Bouabdallah-Laforest qui souffre aussi bien de son goulot d'étranglement que de son ordonnancement statique. Notons qu'en fonction de la taille des requêtes, nos algorithmes ont un taux d'utilisation de 0,4 à 20 fois plus important.

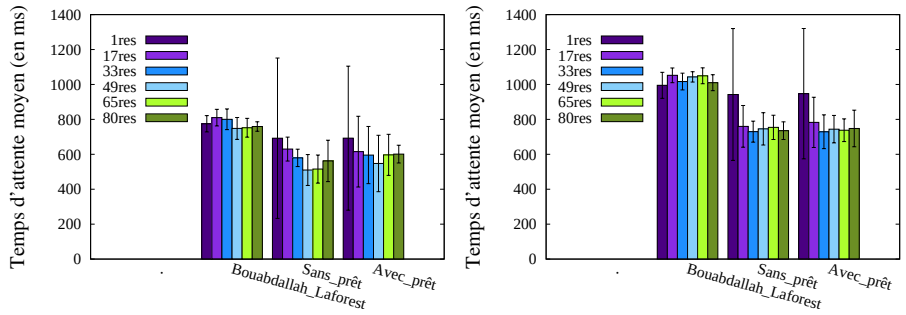
Les courbes relatives à nos deux algorithmes suivent le même comportement que la courbe de l'algorithme en mémoire partagée. En cas de forte charge, lorsque le mécanisme de prêt est activé, l'algorithme présente un meilleur taux d'utilisation quand la taille maximale des requêtes est comprise entre 4 et 16 (gain de plus de 15 %). Un tel écart montre l'efficacité du mécanisme de prêt. Nous pouvons en effet constater que ce mécanisme diminue l'effet négatif des conflits induits par les requêtes moyennes et ne dégrade pas les performances lorsque la taille moyenne des requêtes est haute.

5.3. Temps d'attente moyen

Dans cette section, nous étudions le temps d'attente moyen d'une requête qui correspond à l'intervalle de temps entre le moment où la requête est émise et le moment où le processus obtient le droit d'entrer en section critique.

Pour des valeurs de charge moyenne et haute, les figures 6 et 7 montrent respectivement, le temps d'attente moyen en considérant une petite taille ($SizeReq = 4$) et la plus haute taille considérée ($SizeReq = 80$). Dans la figure 7 nous détaillons le temps d'attente pour différentes tailles de requête. Nous n'avons pas inclus dans ces figures, les performances de l'algorithme incrémental car il est fortement pénalisé par l'effet-domino: le temps d'attente moyen est trop important par rapport au temps d'exécution de l'expérience.

Nous pouvons observer dans les figures 6(a) et 6(b) que nos algorithmes ont un temps d'attente moyen moins important que l'algorithme de Bouabdallah-Laforest lorsque la taille maximale des requêtes est petite (environ 11 fois moins important



(a) Charge moyenne (b) Charge haute

FIGURE 7. *Temps d'attente moyen pour une taille donnée de requête* $(SizeReq = 80)$

en forte charge et 8 fois moins important en charge moyenne). Un tel comportement confirme que nos algorithmes bénéficient de leur faible coût de synchronisation. Nous notons également une amélioration de 20 % lorsque le mécanisme de prêt est activé en forte charge ce qui est cohérent avec les courbes précédentes concernant le taux d'utilisation des ressources.

D'autre part, contrairement à nos algorithmes, dans les figures 7(a) et 7(b), le temps d'attentes moyen et l'écart-type de l'algorithme de Bouabdallah-Laforest est similaire quelque soit la taille de la requête. Bien que notre algorithme soit le plus efficace, son ordonnancement pénalise les petites requêtes. En effet nous observons dans ces mêmes figures que le temps d'attente moyen des petites requêtes et l'écart-type respectif sont les plus importants. À cause de notre politique d'ordonnancement, i.e., la fonction $\mathcal{A}$, l'ordre d'accès d'une requête à une seule ressource dépend de la valeur du compteur correspondant. En d'autres termes, la moyenne des valeurs du vecteur retournée par la fonction concerne dans ce cas seulement un unique compteur qui est incrémenté en fonction de la fréquence à laquelle la ressource correspondante est demandée : une ressource populaire aura un compteur élevé comparé à d'autres qui sont demandés moins souvent.

6. Conclusion et perspectives

Dans cet article, nous avons présenté un nouvel algorithme distribué pour allouer de manière exclusive différentes ressources. Il ne nécessite pas de connaitre à l'avance le graphe des conflits et limite les communications entre processus non conflictuels puisqu'il remplace le verrou global de l'algorithme de Bouabdallah-Laforest par un mécanisme de compteurs. L'ordre total des requêtes évitant les interblocages est assuré par la définition d'une fonction $\mathcal{A}$ en tant que paramètre de l'algorithme. Les résultats de l'évaluation de performances confirment que notre mécanisme de compteurs améliore le taux d'utilisation des ressources et réduit le temps d'attente moyen. Cependant, ce mécanisme n'évite pas complètement l'effet-domino qui dégrade les

performances de l'algorithme. Pour limiter cet effet, nous avons ajouté un mécanisme de prêt qui réordonnance dynamiquement les requêtes pendantes, réduisant ainsi la probabilité que l'effet domino se produise (avec un gain allant jusqu'à 20 %).

D'autre part, puisque notre solution limite la communication entre processus non conflictuels, il serait particulièrement intéressant de tester notre algorithme sur une topologie physique hiérarchique telle qu'on peut la trouver dans les grandes infrastructures tel que les Clouds et les grands cluster. En effet, la suppression du verrou global permet maintenant d'éviter la communication inutile entre les différents sites géographiques. Il y a ainsi moins de risques à ce qu'un processus attende un message d'un site très éloigné. De plus, dans notre évaluation de performances, nous avons montré que l'émission d'une requête de prêt lorsqu'il ne manque plus qu'une ressource requise à un processus permettait d'améliorer le taux d'utilisation lorsque la taille des requêtes était moyenne. Il serait ainsi intéressant d'évaluer l'impact de ce seuil sur nos métriques.

Remerciements

Les expériences présentées dans cet article ont été menées sur la plate-forme Grid'5000, soutenu par un groupe scientifique organisé par Inria, le CNRS, RENATER, plusieurs universités ainsi que d'autres organisations.

Bibliographie

Aoxueluo, Wu W., Cao J., Raynal M. (2013). A generalized mutual exclusion problem and its algorithm. In *Parallel processing (icpp), 2013 42nd international conference on*, p. 300-309.

Awerbuch B., Saks M. (1990, oct). A dining philosophers algorithm with polynomial response time. In *Foundations of computer science, 1990. proceedings., 31st annual symposium on*, p. 65 -74 vol.1.

Barbosa V. C., Benevides M. R. F., Filho A. L. O. (2001). A priority dynamics for generalized drinking philosophers. *Inf. Process. Lett.*, vol. 79, n° 4, p. 189-195.

Bhatt V., Huang C. (2010). Group mutual exclusion in $O(\log n)$ RMR. In *Podc 2010, zurich, switzerland, july 25-28, 2010*, p. 45–54.

Bouabdallah A., Laforest C. (2000). A distributed token/based algorithm for the dynamic resource allocation problem. *Operating Systems Review*, vol. 34, n° 3, p. 60-68.

Brooks R. L. (1987). On colouring the nodes of a network. In *Classic papers in combinatorics*, p. 118–121. Springer.

Bulgannawar S., Vaidya N. H. (1995). A distributed k-mutual exclusion algorithm. In *Icdcs*, p. 153-160.

Chandy K. M., Misra J. (1984). The drinking philosopher's problem. *ACM Trans. Program. Lang. Syst.*, vol. 6, n° 4, p. 632-646.

DeMent N. S., Srimani P. K. (1994). A new algorithm for k mutual exclusions in distributed systems. *Journal of Systems and Software*, vol. 26, n° 2, p. 179-191.

Dijkstra E. W. (1971). Hierarchical ordering of sequential processes. *Acta Informatica*, vol. 1, p. 115-138. (10.1007/BF00289519)

Ginat D., Shankar A. U., Agrawala A. K. (1989). An efficient solution to the drinking philosophers problem and its extension. In *Wdag (disc)*, p. 83-93.

Joung Y.-J. (1998). Asynchronous group mutual exclusion (extended abstract). In *Podc*, p. 51-60.

Lamport L. (1978, July). Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, vol. 21, p. 558–565.

Lejeune J. (2014). *Distributed mutual exclusion algorithmic : toward an efficient resource management*. Theses, . Consulté sur https://tel.archives-ouvertes.fr/tel-01077962

Lynch N. A. (1981). Upper bounds for static resource allocation in a distributed system. *J. Comput. Syst. Sci.*, vol. 23, n° 2, p. 254-278.

Maddi A. (1997). Token based solutions to m resources allocation problem. In *Sac*, p. 340-344.

Mueller F. (1999). Priority inheritance and ceilings for distributed mutual exclusion. In *Real-time systems symposium, 1999. proceedings. the 20th ieee*, p. 340 -349.

Naimi M. (1993, octobre). Distributed algorithm for k-entries to critical section based on the directed graphs. *SIGOPS Oper. Syst. Rev.*, vol. 27, n° 4, p. 67–75.

Naimi M., Trehel M. (1987a). How to detect a failure and regenerate the token in the log(n) distributed algorithm for mutual exclusion. In *Wdag*, p. 155-166.

Naimi M., Trehel M. (1987b). An improvement of the log(n) distributed algorithm for mutual exclusion. In *Icdcs*, p. 371-377.

Raymond K. (1989a). A distributed algorithm for multiple entries to a critical section. *Inf. Process. Lett.*, vol. 30, n° 4, p. 189-193.

Raymond K. (1989b). A tree-based algorithm for distributed mutual exclusion. *ACM Trans. Comput. Syst.*, vol. 7, n° 1, p. 61-77.

Raynal M. (1991). A distributed solution to the k-out of-m resources allocation problem. In *Icci*, p. 599-609.

Reddy V. A., Mittal P., Gupta I. (2008). Fair k mutual exclusion algorithm for peer to peer systems. In *Icdcs*, p. 655-662.

Rhee I. (1995). A fast distributed modular algorithm for resource allocation. In *Icdcs*, p. 161-168.

Rhee I. (1998). A modular algorithm for resource allocation. *Distributed Computing*, vol. 11, n° 3, p. 157-168.

Ricart G., Agrawala A. K. (1981, January). An optimal algorithm for mutual exclusion in computer networks. *Commun. ACM*, vol. 24, p. 9–17.

Satyanarayanan R., Muthukrishnan C. R. (1994). Multiple instance resource allocation in distributed computing systems. *J. Parallel Distrib. Comput.*, vol. 23, n° 1, p. 94-100.

Srimani P. K., Reddy R. L. N. (1992). Another distributed algorithm for multiple entries to a critical section. *Inf. Process. Lett.*, vol. 41, n° 1, p. 51-57.

Styer E., Peterson G. L. (1988). Improved algorithms for distributed resource allocation. In *Podc*, p. 105-116.

Suzuki I., Kasami T. (1985). A distributed mutual exclusion algorithm. *ACM Trans. Comput. Syst.*, vol. 3, n° 4, p. 344-349.