ComPAS'2014 : Parallélisme / Architecture / Système
Neuchâtel, Suisse, du 23 au 25 avril 2014

Un algorithme distribué efficace d'exclusion mutuelle généralisée sans connaissance préalable des conflits

Jonathan Lejeune, Luciana Arantes, Julien Sopena, Pierre Sens

UPMC/CNRS-LIP6 et INRIA,
4, place Jussieu
75252 Paris Cedex 05, France
prénom.nom@lip6.fr

Résumé
Les algorithmes d'exclusion mutuelle généralisée tel que l'algorithme du cocktail des philosophes de Chandy-Misra permettent de gérer les accès concurrents des processus sur un ensemble de ressources partagées. Afin d'optimiser le taux d'utilisation des ressources, ces algorithmes doivent ordonnancer les requêtes de manière à maximiser les exécutions parallèles des requêtes. Beaucoup de solutions reposent sur l'hypothèse forte d'une connaissance préalable des conflits entre les processus. D'autres approches n'ont pas besoin d'une telle connaissance mais n'optimisent pas le parallélisme des requêtes non-conflictuelles et utilisent bien souvent des mécanismes de diffusion dégradant la complexité en messages. Nous proposons dans cet article une solution générale permettant de tirer parti des requêtes non-conflictuelles. En suivant ce schéma, nous décrivons deux nouveaux algorithmes basés sur un arbre dynamique. Nos évaluations montrent que notre solution améliore le taux d'utilisation d'un facteur 1 à 10 en fonction du nombre de requêtes conflictuelles.

Mots-clés : algorithme distribué, exclusion mutuelle généralisée, cocktail des philosophes

1. Introduction

Dans les applications parallèles, les processus doivent accéder aux ressources partagées de manière exclusive afin d'éviter les incohérences. Si le problème concerne une seule et unique ressource, il est possible alors d'utiliser un algorithme distribué d'exclusion mutuelle classique (par exemple [9],[18],[20], [15], [13], [14]) 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 un accès exclusif à plusieurs de ces ressources et commencer à exécuter leurs sections critiques que lorsqu'ils ont obtenu les ressources demandées. Ceci est une généralisation du problème de l'exclusion mutuelle.
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 un lien 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. Un interblocage est un état du système dans lequel aucune progression n'est possible. 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 [7] avec le problème du "Dîner des philosophes" où les processus demandent un ensemble de ressources statiques. Ce problème a été étendu par Chandy et Misra [6] sous le nom du problème du "cocktail des philosophes" où les processus peuvent demander à chaque nouvelle requête un ensemble variable de ressources. Il existe deux familles d'algorithmes pour ce problème de multi-ressource : les algorithmes hiérarchiques ([10, 19]) et ceux simultanés ([1, 4, 8, 11, 3]). 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 dans la littérature 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. Néanmoins, ces solutions peuvent fonctionner sans connaissance préalable du graphe de conflits en considérant de façon pessimiste un graphe complet. Dans ce cas, les applications ne peuvent pas bénéficier de manière optimale du parallélisme potentiel des processus non-conflictuels. La propriété de concurrence peut être violée et entraîner une dégradation du taux d'utilisation des ressources et du temps d'attente moyen. Les algorithmes de la famille hiérarchique sont pénalisées par un "effet domino" d'attentes en cascade lorsque le nombre de conflits devient grand. La plupart des algorithmes de la famille simultanée sont pénalisés par un goulot d'étranglement dégradant ainsi le parallélisme lorsqu'il y a peu de conflits dans le système. Ce goulot d'étranglement est généralement dû à l'acquisition d'une ressource globale auxiliaire qui permet de contrôler la sérialisation les requêtes. 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 dû à leur forte complexité de messages.

Dans cet article, nous proposons un nouveau schéma d'algorithme décentralisé pour le verrouillage de ressources dans un système distribué, qui ne requiert pas de connaissances a priori sur le graphe des conflits. Notre solution n'utilise pas de mécanisme de diffusion, ordonne dynamiquement les requêtes pour bénéficier au mieux du parallélisme potentiel et empêche deux processus non conflictuels de communiquer. Ces améliorations donnent de bonnes performances en termes de taux d'utilisation des ressources lorsqu'il y a peu de conflits dans le système et limite la dégradation de performance lorsque la quantité de conflits augmente.

Cet article est organisé de la manière suivante. La section 2 présente les principaux algorithmes distribués de la littérature qui résolvent le problème du multi-ressource. La section 3 donne les définitions et les hypothèses du système considéré. Le schéma général de notre solution est décrit section 4. La section 5 présente les performances de deux algorithmes découlant de notre schéma général et les compare avec deux algorithmes de la littérature. La section 6 conclut l'article.

2. État de l'art

Le problème du multi-ressource dans un système distribué est une des généralisations du problème de l'exclusion mutuelle classique. Dans ce problème le système est composé de plusieurs types de ressources en un seul exemplaire. Cette section présente les principaux algorithmes multi-ressource. Ils sont classés en deux familles : incrémentale et 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 des attentes peut se produire : un processus attend des ressources qui ne sont pas en cours d'utilisation mais qui sont verrouillées par des processus qui attendent l'acquisition d'autres ressources. L'effet domino dégrade la propriété de concurrence et par conséquent réduit fortement le taux d'utilisation des ressources.
Pour éviter l'effet domino, Lynch [10] propose de construire un graphe dual au graphe de conflit : les nœuds sont les ressources et il existe un lien entre deux nœuds si les deux ressources correspondantes sont susceptibles d'être demandées au sein d'une même requête. En coloriant ce graphe et en minimisant le nombre de couleurs, il est possible alors de définir un ordre partiel sur l'ensemble des ressources du système si on définit un ordre total sur l'ensemble des couleurs. Les processus demanderont alors les ressources dans l'ordre des couleurs associés. Ceci réduit l'effet domino et améliore l'exploitation du parallélisme. Cependant le coloriage de graphe est un problème NP-complet et il est difficile de trouver un coloriage optimal.
Styer et Peterson [19] ont proposé une solution en considérant un coloriage quelconque (de préférence optimisé) pour réduire le temps d'attente avec un mécanisme d'annulation de verrouillage : un processus peut libérer une ressource (ou une couleur) même si il ne l'a pas encore utilisée. Ceci permet de casser dynamiquement les possibles chaînes de processus en attente 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.
Cependant, puisque nous souhaitons considérer un graphe de conflits inconnu qui est équivalent à considérer un graphe complet, aucune coloration optimale n'est possible (théorème de Brook de la théorie des graphes [5]). Ainsi les optimisations dues au coloriage du graphe ne sont pas suffisantes pour assurer une bonne exploitation du parallélisme.

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 [6] 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. Le problème du cocktail des philosophes permet au site de demander un ensemble de ressources différent à chaque nouvelle requête contrairement au problème du dîner des philosophes où les processus demandent en permanence le même ensemble de ressources. Le graphe de communication correspond directement au graphe des conflits et implique de le connaître a priori. Chaque processus partage une bouteille ou une fourchette (en fonction du problème considéré) avec chaque voisin. En orientant le graphe des conflits, il en résulte un graphe de précédence. Si les circuits sont évités dans ce graphe de précédence, les interblocages ne peuvent pas se produire. Il a été montré que le problème du dîner des philosophes respecte cette acyclicité. Cependant ceci n'est pas le cas pour le problème du cocktail. Chandy et Misra

ont adapté le problème du cocktail en utilisant les procédures du dîner : pour verrouiller un sous-ensemble de bouteilles 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, en considérant un graphe de conflit complet, le graphe de communication devient aussi complet et la complexité en messages augmente de manière significative. De plus, la phase d'acquisition des fourchettes réduit le degré de parallélisme.

Ginat et al. [8] ont remplacé la phase d'acquisition des fourchettes de l'algorithme de Chandy-Misra par une horloge logique [9]. 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 [16, 17] 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.

Maddi [11] 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 [20].

L'algorithme de Bouabdallah-Laforest [4] est plus détaillé car nous le comparons avec notre solution dans la section 5. Cet algorithme est basé sur la circulation de jetons entre les processus. Chaque ressource est représentée par un unique jeton et est associée à une file d'attente distribuée dont le premier élément est le possesseur du jeton. Avant de demander un ensemble de ressources, le processus doit d'abord obtenir un unique jeton de contrôle. L'algorithme d'exclusion mutuelle gérant ce jeton de contrôle est l'algorithme de Naimi-Tréhel [13]. Un jeton protégeant une ressource peut être possédé par un processus ou être directement stocké dans le jeton de contrôle si il n'est pas utilisé. Le jeton de contrôle contient 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 ($O(Log(N))$) mais le jeton de contrôle induit un important goulot d'étranglement quand il y a peu de conflits dans les requêtes pendantes réduisant le parallélisme.

D'autre part, des extensions du problème multi-ressource ont été proposées pour prendre en compte des systèmes dynamiques où l'ensemble des processus peut changer durant l'exécution de l'algorithme. Awerbuch et Saks [1] proposent le maintien d'une file d'attente globale. BarIlan et Peleg [2] ont étendu la solution de Awerbuch-Saks pour améliorer le temps d'attente dans une version centralisée. Weidman et al. [21] ont adapté l'algorithme de Chandy-Misra.

3. Définitions et hypothèses

Nous considérons un système distribué composé d'un ensemble de N sites fiables $\Pi = \{s_1, s_2, ...s_N\}$. L'ensemble Π est totalement ordonné par la relation d'ordre $\prec$ où $s_i \prec s_j$ si $i < j$. 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.
Le système contient un ensemble $R = \{r_1, r_2..., r_m\}$ de m ressources différentes. Aucun ordre n'est défini sur R.
Nous notons T le temps d'exécution du système et $D_i^t \subseteq R$, l'ensemble des ressources qu'un site $s_i \in \Pi$ demande à l'instant $t \in T$.
Un site s_i commence sa section critique à l'instant t' ($t' > t$) lorsqu'il possède toutes les ressources $r \in D_i^t$. Nous supposons que les processus demandent la section critique en appelant la procédure $Request_CS(D_i^t)$ et la libère en appelant la procédure $Release_CS$. De plus, 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.

4. Schéma général de la solution

4.1. Principe

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 graphes. 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 [6] utilisent un algorithme de "Dîner des philosophes", [8] et [11] utilisent des horloges logiques et Bouabdallah-Laforest [4] utilisent une section critique de contrôle. Cependant, l'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 [18], et l'algorithme de Bouabdallah-Laforest utilise l'algorithme de Naimi-Tréhel [13]. L'algorithme de Ricart-Agrawala est basé sur des permissions et utilisent aussi un mécanisme de diffusion (complexité en message de $O(2N-1)$) alors que l'algorithme de Naimi-Tréhel est basé sur la circulation d'un jeton dans un arbre dynamique (complexité en message de $O(log(N))$). Par conséquent, notre algorithme se base sur l'algorithme de Bouabdallah-Laforest. Ce dernier est décomposé en deux étapes :
- **Première étape** : obtenir la section critique de contrôle.
- **Deuxième étape** : demander toutes les ressources nécessaires, attendre un acquittement et enfin libérer la section critique de contrôle de la première étape.

Bien que cet algorithme ait une bonne complexité en messages, la première étape dégrade le parallélisme de l'application car :
- deux processus non conflictuels doivent communiquer ensemble par le biais du jeton de

contrôle,

– la sérialisation des requêtes ne dépend pas du nombre de ressources nécessaires associées mais uniquement de l'ordre d'acquisition de la première étape (cette remarque est aussi valable pour les algorithmes avec horloge logique). Rappelons que plus l'ensemble de ressources requis est petit, moins la requête a de chance d'être en conflit avec une autre. Ainsi dans l'algorithme de Bouabdallah-Laforest les "grandes" requêtes peuvent bloquer, pendant l'acquisition de leurs ressources, des petites requêtes non conflictuelles qui auraient pu s'exécuter simultanément si elles avaient eu le jeton de contrôle avant. Par conséquent, prendre en compte le nombre de ressources dans la politique d'ordonnancement permet d'améliorer le parallélisme.

Nous avons donc deux objectifs :

– paralléliser la première étape pour éviter la communication entre deux processus non conflictuels,
– modifier l'ordonnancement dynamiquement afin de privilégier les petites requêtes mais en préservant toujours la vivacité pour les grandes requêtes.

4.2. Description du schéma général

4.2.1. Paralléliser la première étape

Le but de la première étape est de donner un ordre de passage commun sur l'ensemble des files d'attentes associées aux ressources. Pour éviter le goulot d'étranglement, le principe de cette parallélisation est d'utiliser *un compteur par ressource* qui donnera un ordre de passage pour une ressource donnée. Le système maintient donc m compteurs à accès exclusif. La première étape consiste à demander la valeur de chaque compteur associé aux ressources requises et de les incrémenter de un : ceci assure une valeur différente à chaque nouvelle lecture. Une fois que le site connaît l'ensemble des compteurs, sa requête peut être associée à un vecteur de m entiers dans l'ensemble $\mathbb{N}^m$ (les ressources non demandées ont une valeur nulle dans le vecteur). Par conséquent, la requête est définie de manière unique quel que soit l'instant où elle a été émise et quel que soit son ensemble de ressources requises. Ainsi, lors de la seconde étape, le processus pourra demander chaque ressource indépendamment les unes des autres en indiquant ce vecteur.

4.2.2. Éviter les interblocages

Une requête req_i émise par le site $s_i \in \Pi$ pour une ressource donnée est associée à deux informations : le site initiateur de la requête (s_i) et un vecteur $v_i \in \mathbb{N}^m$ (cf. section 4.2.1). Les interblocages sont évités si nous définissons un ordre total sur les requêtes. Nous allons d'abord utiliser un ordre partiel sur l'ensemble des vecteurs en définissant une application surjective $\mathcal{A} : \mathbb{N}^m \to \mathbb{R}$ qui transforme un vecteur d'entiers en un réel. Puisque $\mathcal{A}$ donne un ordre partiel, nous devons utiliser l'ordre total défini sur Π pour ordonner totalement les requêtes. Nous notons cet ordre total $\triangleleft$ où $req_i \triangleleft req_j$ 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. L'application $\mathcal{A}$ permet de définir une politique d'ordonnancement sur les requêtes et peut être vu comme un paramètre de l'algorithme. La seconde étape ordonnancera les requêtes en fonction de $\triangleleft$. Si l'ordre total sur les requêtes évite l'interblocage en assurant que toutes les requêtes sont différentiables, il ne garantit pas la vivacité. La famine peut être évitée avec une application $\mathcal{A}$ qui assure que dans un temps fini toute requête sera la requête la plus petite dans l'ordre $\triangleleft$.

4.2.3. Privilégier les petites requêtes

L'application $\mathcal{A}$ est un moyen de définir une politique d'ordonnancement sur les requêtes. Dans notre solution, $\mathcal{A}$ est égale à la moyenne des valeurs du vecteur de la requête. Puisque les ressources non requises ont une valeur nulle dans le vecteur, les petites requêtes deman-

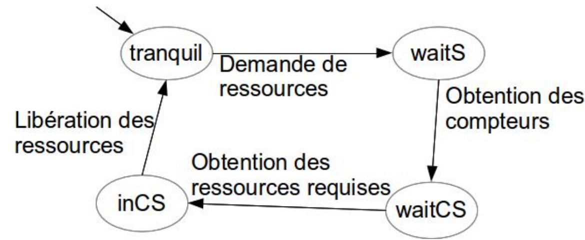


FIGURE 1 – Machine à états du schéma général

dant peu de ressources seront privilégiées. De plus, la famine est impossible car les compteurs augmentent à chaque nouvelle requête impliquant que la valeur minimum de $\mathcal{A}$ augmentera également à chaque nouvelle requête. La propriété de vivacité est ainsi toujours respectée.

4.2.4. État des processus

Les processus ont quatre états possibles
- $tranquil$: le processus ne demande rien.
- $waitS$: le processus attend les compteurs requis (première étape).
- $waitCS$: le processus attend les ressources requises (seconde étape).
- $inCS$: le processus utilise les ressources requises (section critique).

La figure 1 représente la machine à états des processus.

4.2.5. Messages

Chaque type de message contient l'identifiant de la ressource associée r. Nous définissons quatre types de message :
- $\mathtt{request1}(r)$: un message pour demander la valeur du compteur associé à r.
- $\mathtt{counter}(r)$: un message indiquant la valeur du compteur associé à r.
- $\mathtt{request2}(r)$: un message pour demander la ressource r. Il contient l'identifiant du site initiateur et un réel égal au résultat de $\mathcal{A}$ du vecteur d'entiers.
- $\mathtt{token}(r)$: un message représentant le jeton de la ressource r.

Il est possible d'économiser des messages en les agrégeant. En effet, les messages du même type adressés au même destinataire seront agrégés en un seul message de ce type. Par conséquent, la réception d'un message concerne non pas une seule ressource mais un ensemble de ressources. L'envoi de messages implique donc des tampons locaux, un par type de message et par destinataire.

5. Évaluation de performances

Nous présentons dans cette section une évaluation de performances comparant deux algorithmes de notre schéma général avec un algorithme incrémental fixant un ordre prédéfini d'accès aux ressources et utilisant m instances d'algorithme de Naimi-Tréhel avec files locales [14] et l'algorithme de Bouabdallah-Laforest [4]. Nous décrivons brièvement les spécificités des deux algorithmes, puis le protocole d'expérimentation et enfin les performances concernant le taux d'utilisation et le temps d'attente moyen.

5.1. Description des deux algorithmes

5.1.1. Premier algorithme

L'idée principale est de combiner les solutions existantes de la littérature en respectant le schéma général de la section 4. Pour la première étape, comme les compteurs sont des ressources auxiliaires, nous utilisons m instances d'un algorithme classique d'exclusion mutuelle basé sur la circulation d'un jeton. L'algorithme choisi est celui de Naimi-Tréhel avec files locales [14] car il est performant avec une complexité en messages qui s'adapte à la charge ($O(Log(N))$ avec une faible charge et constante en cas de forte charge). Les compteurs sont stockés directement dans ces jetons auxiliaires. À la réception d'un jeton, le compteur est lu et incrémenté et éventuellement transmis à un prochain demandeur.

Concernant la deuxième étape, nous avons besoin de réordonnancer les requêtes en fonction des nouvelles requêtes émises qui peuvent être plus prioritaires. Ceci est analogue à un algorithme d'exclusion mutuelle à priorité où chaque requête est associée à un niveau de priorité. Nous avons choisi m instances simplifiées de l'algorithme à priorités de Mueller [12] qui est une extension de l'algorithme de Naimi-Tréhel [13]. Il a une complexité moyenne en messages de $O(\frac{N}{2})$. Cet algorithme est basé sur la circulation d'un jeton dans une topologie d'arbre dynamique où la racine est le porteur du jeton. Dans notre cas, une priorité correspond au résultat de l'appel à $\mathcal{A}$. Contrairement à l'algorithme de Mueller, plus la valeur réelle est basse plus la priorité est haute.

5.1.2. Second algorithme

Le principal inconvénient du premier algorithme est que les requêtes pour une seule ressource doivent aussi exécuter la première étape alors qu'elles pourraient directement demander la ressource réelle. Cependant la première étape est indispensable pour assurer la vivacité. En effet, si les requêtes à une ressource contournent la première étape, elles peuvent accéder en permanence aux ressources et donc empêcher les autres requêtes d'être satisfaites. Le principe de ce second algorithme est de fusionner les structures logiques des deux étapes. Au lieu des $2m$ jetons (m pour les compteurs et m pour les ressources) du premier algorithme, nous avons plus que m jetons. Le compteur est ainsi stocké directement dans le jeton protégeant la ressource. Ainsi, lorsqu'un processus demande des ressources, pour chaque ressource il envoie un message $\mathtt{Request1}$ (requête pour la première phase) à son père dans la structure logique en indiquant si la demande concerne une seule ressource. Lorsqu'un porteur de jeton reçoit un message $\mathtt{Request1}$ pour une seule ressource, il la traite comme une requête de phase 2. En effet, la demande ne concernant qu'une seule ressource, le porteur du jeton peut directement calculer le résultat de $\mathcal{A}$ avec le compteur correspondant. Si la requête concerne plusieurs ressources, le porteur du jeton envoie au demandeur un message $\mathtt{counter}$ contenant la valeur du compteur de sa ressource ou bien directement le jeton si il ne l'utilise pas.

5.2. Protocole d'expérimentation

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

Une application est caractérisée par :

- N : le nombre de processus (32 dans notre cas).
- m : le nombre total de ressources dans le système (80 dans notre cas).
- α : le temps d'exécution de la section critique (variable entre 5 ms et 35 ms).
- β : intervalle de temps entre le moment où un processus libère la section critique et le moment où il la redemande.

- γ : latence réseau pour envoyer un message entre deux processus (négligeable comparé à α).
- ρ : le rapport $\beta/(\alpha+\gamma)$, qui exprime la fréquence à laquelle la section critique est demandée. La valeur de ce paramètre est inversement proportionnelle à la charge en requêtes : une valeur basse donne une charge haute et vice-versa. Dans nos expériences, nous avons considéré une charge haute et moyenne.
- ϕ : le nombre maximum de ressources qu'un site peut demander. Ce paramètre est compris entre 1 et m. Plus sa valeur est élevée, plus le parallélisme potentiel diminue car la probabilité d'avoir des requêtes conflictuelles augmente.

À chaque nouvelle requête, un processus choisit x ressources. Le temps de section critique de la requête dépend alors de la valeur de x : plus cette valeur est grande et plus le temps de section critique risque d'être grand car nous considérons qu'une requête demandant beaucoup de ressources doit en pratique avoir un plus grand temps de calcul en section critique. Nous présentons chaque métrique en charge haute et moyenne.

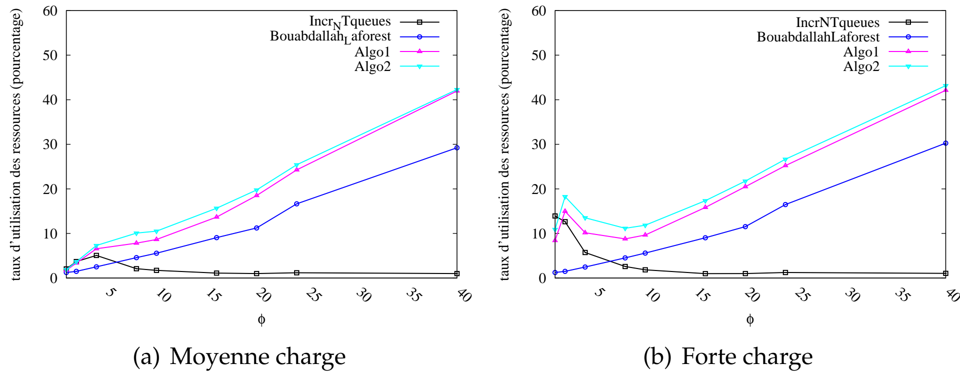
5.3. Taux d'utilisation des ressources

Nous montrons ici l'impact de ϕ sur le taux d'utilisation des ressources. Cette métrique est exprimée en pourcentage de temps où les ressources sont utilisées (100% indique que toutes les ressources sont utilisées tout le temps de l'expérience). Nous montrons l'impact de ϕ sur deux scénarios :
- la valeur x est toujours égale à ϕ (Figure 2),
- la valeur x peut être différente à chaque nouvelle requête et est choisie selon une loi aléatoire uniforme entre 1 et ϕ (Figure 3).

Dans la figure 2, nous avons borné ϕ à $m/2$ car au-delà de cette borne, il ne peut pas exister deux requêtes qui s'exécutent en même temps impliquant une sérialisation totale des requêtes et une impossibilité de satisfaire la propriété de concurrence. Le motif de demande devient alors similaire à un motif d'exclusion mutuelle classique.

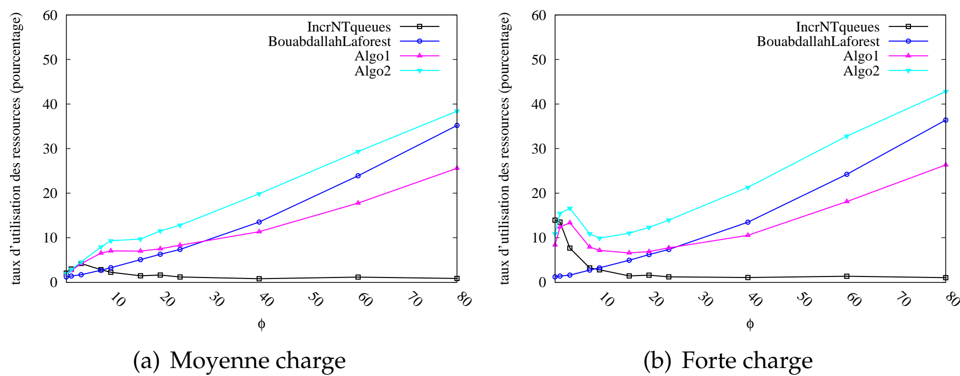
D'une façon générale, dans les figures 2 et 3, lorsque ϕ augmente le taux d'utilisation augmente globalement. En effet, le fait que le nombre de sites N soit inférieur au nombre de ressources m explique le faible taux d'utilisation lorsque les requêtes sont petites (peu de ressources). Quand la taille moyenne des requêtes augmente, les exécutions de sections critiques impliquent un nombre de ressources plus important.

Comparons désormais les différents algorithmes. Lorsqu'il n'y a pas de conflit ($\phi = 1$), l'algorithme incrémental est le plus efficace car les ressources sont gérées de manière indépendante et aucun mécanisme de résolution de conflit n'est utilisé. Nos deux algorithmes ont alors un taux d'utilisation plus bas que l'algorithme incrémental. L'algorithme de Bouabdalah-Laforest reste le moins performant. Lorsque le nombre de conflits augmente, nous remarquons que le taux d'utilisation de :
- l'algorithme incrémental diminue et se stabilise. Cet algorithme ne bénéficie pas de l'augmentation de la taille des requêtes à cause de l'effet domino qui a un poids trop important : le moindre conflit le pénalise énormément.
- l'algorithme de Bouabdalah-Laforest augmente très régulièrement. Bien que cet algorithme reste très pénalisé par son goulot d'étranglement lorsqu'il y a peu de conflit (notamment en charge haute) son taux d'utilisation augmente proportionnellement à la taille des requêtes. Dans la figure 2, comme les requêtes de chaque point font exactement la même taille, la dégradation par rapport à nos deux algorithmes n'est donc pas dû principalement à sa politique d'ordonnancement. En effet, cette dégradation est due au coût de l'attente du jeton de contrôle qui absorbe les gains que l'on a grâce aux exécutions concurrentes. Dans la figure 3, on remarque que l'augmentation est plus rapide car l'algorithme arrive néanmoins à tirer parti des requêtes concurrentes sans pour autant égaler notre deuxième algorithme. En effet, il reste toujours pénalisé par son goulot d'étranglement et sa politique d'ordonnancement



(a) Moyenne charge (b) Forte charge

FIGURE 2 – Taux d'utilisation des ressources quand $x = \phi$



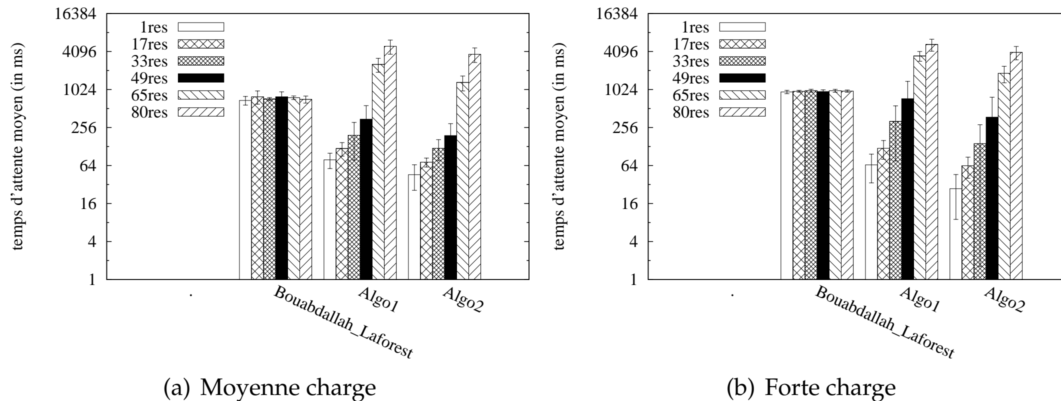
(a) Moyenne charge (b) Forte charge

FIGURE 3 – Taux d'utilisation des ressources quand $1 \leq x \leq \phi$

qui ne tient pas compte de la taille des requêtes l'empêchant d'optimiser la parallélisation.
– nos deux algorithmes subissent une augmentation soudaine entre les valeurs d'abscisse 1 et 2 puis une baisse entre les valeurs 3 et 8 puis, augmentent à nouveau régulièrement par la suite. La hausse soudaine entre les points 1 et 2 fait passer le taux d'utilisation du simple au double car la taille des requêtes a en moyenne doublé et que le nombre de ressources maximales utilisables en même temps est toujours inférieur à m. La diminution qui s'ensuit des abscisses 3 à 8 est liée à la sérialisation des requêtes conflictuelles qui ont une taille encore trop faible par rapport au nombre de conflits pour faire augmenter le taux d'utilisation. Enfin l'augmentation ultime s'explique par le fait que la taille moyenne des requêtes prend le dessus sur la sérialisation des requêtes conflictuelles. Sur la figure 2, les deux algorithmes sont plus ou moins équivalents car les requêtes ont toutes la même taille alors que sur la figure 3, l'algorithme 2 est plus performant car il privilège les plus petites requêtes.

5.4. Temps d'attente moyen par taille de requête

Dans la figure 4, nous montrons le temps d'attente moyen pour entrer en section critique classé par taille de requête quand $1 \leq x \leq m$. Le temps d'attente représente le temps entre le moment où commence la demande de ressources et le moment de l'acquisition de l'ensemble des ressources. Les graphiques ont une échelle logarithmique sur l'axe des ordonnées. Nous n'indiquons pas les performances de l'algorithme incrémental car l'effet domino le pénalise énor-



(a) Moyenne charge (b) Forte charge

FIGURE 4 – Temps d'attente moyen par taille de requête avec $1 \leq x \leq m$

mément comme nous avons pu le remarquer dans les précédentes courbes : le temps d'attente moyen des points considérés est trop important par rapport au temps de l'expérience.
Nous remarquons que l'algorithme de Bouabdallah-Laforest est très équitable : le temps d'attente moyen est le même quel que soit le nombre de ressources demandées. Cette unique valeur est en fait le temps moyen d'acquisition du jeton de contrôle qui masque et absorbe l'impact de la taille des requêtes.
Nos deux algorithmes ont un temps d'attente moyen proportionnel à la taille des requêtes (forme d'escalier). Cependant, les temps d'attente moyens pour le deuxième algorithme sont moins importants que pour le premier l'algorithme grâce à la fusion des deux étapes qui induit moins de communications pour obtenir les ressources. De plus nous remarquons que les petites (1 et 17) et moyennes (33 et 49) requêtes sont favorisées par la baisse de charge alors que les grosses requêtes (65 et 80) ont un temps d'attente relativement équivalent sur les deux charges. En effet quand la charge diminue, le nombre de requêtes pendantes diminue également. Par conséquent, les petites et moyennes requêtes ont moins de chance d'être en concurrence avec des requêtes conflictuelles. Ceci réduit donc leur temps d'attente. Cependant, bien que le nombre de requêtes pendantes soit moins important en charge moyenne, les grosses requêtes ne bénéficie pas de la baisse de charge car elles sont pénalisées par leur taille. En effet, une taille importante induit des conflits avec un plus grand nombre de requêtes.

6. Conclusion

Nous avons présenté un nouveau schéma pour verrouiller un ensemble de ressources différentes dans un système distribué. Notre solution n'évite pas les attentes en cascades mais réduit la probabilité qu'elles se produisent. Nos deux algorithmes découlent d'un schéma général améliorant le taux d'utilisation des ressources car (1) il favorise les requêtes avec peu de ressources et (2) exploite le parallélisme du système quand il y a peu de conflits. Lorsque la taille des requêtes augmente, nos solutions gardent globalement de meilleures performances que l'algorithme de Boubadallah-Laforest au détriment du temps d'attente des requêtes requérant beaucoup de ressources. Cependant, dans notre schéma général, il est possible de trouver un compromis pour maximiser le taux d'utilisation des ressources et minimiser le temps d'attente des grosses requêtes. Ce compromis peut être trouvé grâce à l'application $\mathcal{A}$ qui détermine la priorité des requêtes et qui pourrait prendre en compte la taille de la requête.

7. Remerciements

Ces travaux ont été financés par l'Agence nationale française pour la recherche (ANR), dans le cadre du projet MyCloud (ANR-10-SEGI-0009, http ://mycloud.inrialpes.fr/). Les expériences ont été réalisées sur la plateforme Grid'5000 (https ://www.grid5000.fr/).

Bibliographie

1. Awerbuch (B.) et Saks (M.). – A dining philosophers algorithm with polynomial response time. – In *FOCS*, pp. 65 –74 vol.1, oct 1990.
2. Bar-Ilan (J.) et Peleg (D.). – Distributed resource allocation algorithms (extended abstract). – In *WDAG*, pp. 277–291, 1992.
3. Barbosa (V. C.), Benevides (M. R. F.) et Filho (A. L. O.). – A priority dynamics for generalized drinking philosophers. *Inf. Process. Lett.*, vol. 79, n4, 2001, pp. 189–195.
4. Bouabdallah (A.) et Laforest (C.). – A distributed token/based algorithm for the dynamic resource allocation problem. *Operating Systems Review*, vol. 34, n3, 2000, pp. 60–68.
5. Brooks (R. L.). – On colouring the nodes of a network. *In : Classic Papers in Combinatorics*, pp. 118–121. – Springer, 1987.
6. Chandy (K. M.) et Misra (J.). – The drinking philosopher's problem. *ACM Trans. Program. Lang. Syst.*, vol. 6, n4, 1984, pp. 632–646.
7. Dijkstra (E. W.). – Hierarchical ordering of sequential processes. *Acta Informatica*, vol. 1, 1971, pp. 115–138. – 10.1007/BF00289519.
8. Ginat (D.), Shankar (A. U.) et Agrawala (A. K.). – An efficient solution to the drinking philosophers problem and its extension. – In *WDAG (Disc)*, pp. 83–93, 1989.
9. Lamport (L.). – Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, vol. 21, July 1978, pp. 558–565.
10. Lynch (N. A.). – Upper bounds for static resource allocation in a distributed system. *J. Comput. Syst. Sci.*, vol. 23, n2, 1981, pp. 254–278.
11. Maddi (A.). – Token based solutions to m resources allocation problem. – In *SAC*, pp. 340–344, 1997.
12. Mueller (F.). – Priority inheritance and ceilings for distributed mutual exclusion. – In *Real-Time Systems Symposium, 1999. Proceedings. The 20th IEEE*, pp. 340 –349, 1999.
13. Naimi (M.) et Trehel (M.). – How to detect a failure and regenerate the token in the log(n) distributed algorithm for mutual exclusion. – In *WDAG*, pp. 155–166, 1987.
14. Naimi (M.) et Trehel (M.). – An improvement of the log(n) distributed algorithm for mutual exclusion. – In *ICDCS*, pp. 371–377, 1987.
15. Raymond (K.). – A tree-based algorithm for distributed mutual exclusion. *ACM Trans. Comput. Syst.*, vol. 7, n1, 1989, pp. 61–77.
16. Rhee (I.). – A fast distributed modular algorithm for resource allocation. – In *ICDCS*, pp. 161–168, 1995.
17. Rhee (I.). – A modular algorithm for resource allocation. *Distributed Computing*, vol. 11, n3, 1998, pp. 157–168.
18. Ricart (G.) et Agrawala (A. K.). – An optimal algorithm for mutual exclusion in computer networks. *Commun. ACM*, vol. 24, January 1981, pp. 9–17.
19. Styer (E.) et Peterson (G. L.). – Improved algorithms for distributed resource allocation. – In *PODC*, pp. 105–116, 1988.
20. Suzuki (I.) et Kasami (T.). – A distributed mutual exclusion algorithm. *ACM Trans. Comput. Syst.*, vol. 3, n4, 1985, pp. 344–349.
21. Weidman (E. B.), Page (I. P.) et Pervin (W. J.). – Explicit dynamic exclusion algorithm. – In *SPDP*, pp. 142–149, 1991.