L-Exclusion autostabilisante revisitée

Fabienne Carrier, Ajoy Datta, Stéphane Devismes, Lawrence Larmore

ℓ-Exclusion autostabilisante revisitée [†]

Fabienne Carrier[1], Ajoy K. Datta[2], Stéphane Devismes[1] et Lawrence L. Larmore[2]
[1] *VERIMAG UMR 5104, Université Grenoble Alpes, France*
[2] *School of Computer Science, UNLV, Las Vegas, USA*

La ℓ-exclusion a pour but de partager ℓ ressources identiques. Elle est définie par trois propriétés : l'*équité*, la *sûreté* et une propriété d'efficacité appelée *évitement de ℓ-interblocage*. Nous montrons que tout algorithme réalisant ces trois propriétés a un temps d'attente en $\Omega(n-\ell)$ rondes dans un anneau de n processus. Donc, quand n est grand comparé à ℓ, le gain d'avoir ℓ ressources identiques au lieu d'une seule devient négligeable. Pour contourner ce problème, nous reformulons la définition de ℓ-exclusion en remplaçant la propriété d'évitement de ℓ-interblocage par une contrainte sur le temps d'attente. Nous illustrons cette nouvelle version du problème avec des algorithmes autostabilisants dont le temps d'attente est en $O(\frac{n}{\ell})$ rondes, la borne asymptotique optimale.

Keywords: autostabilisation, ℓ-exclusion, circulation de ℓ jetons, temps d'attente.

1 Introduction

L'un des objectifs majeurs des systèmes distribués est de partager un petit nombre de ressources parmi un grand nombre de processus. Par exemple, le problème classique de l'*exclusion mutuelle* consiste à assurer à tout processus demandeur l'accès exclusif en temps fini à une unique ressource, *e.g.*, une imprimante. L'accès à cette ressource est géré par une section particulière du code, appelée *section critique*. Le temps nécessaire à l'exécution d'une section critique est supposé fini, mais non borné. Ainsi, l'exclusion mutuelle consiste à assurer que tout processus qui demande la ressource exécute sa section critique en temps fini tout en interdisant l'exécution concurrente de plusieurs sections critiques. Le problème de la ℓ-exclusion [4] est une généralisation de l'exclusion mutuelle, où jusqu'à $\ell \geq 1$ sections critiques peuvent être exécutées concurremment. Résoudre ce problème permet, par exemple, de gérer un ensemble de ℓ imprimantes. Dans [4], la ℓ-exclusion est définie par les trois propriétés suivantes :
Sûreté : Au plus ℓ processus exécutent leur section critique concurremment.
Vivacité : Tout processus demandeur entre en section critique en temps fini.
Évitement de ℓ-interblocage : Si moins de ℓ processus sont en section critique et qu'il y a au moins un processus demandeur, alors un processus demandeur finira par entrer en section critique, même si aucun processus ne quitte sa section critique entre-temps.

Pour l'illustrer en quoi consiste une solution ne réalisant pas l'évitement de ℓ-interblocage, Fisher *et al* [4] proposent l'allégorie suivante. Dans une banque, il y a ℓ guichets offrant des services identiques, mais une seule file d'attente pour l'accès aux guichets. Uniquement la personne en tête de file peut accéder à un guichet. Donc, on peut considérer que la section critique commence par le déplacement de la tête de file jusqu'au guichet et termine lorsque l'on quitte le guichet. Une personne peut alors bloquer l'accès à tous les guichets. Supposons en effet que la personne en tête de file soit absorbée par la lecture de son journal, elle bloque l'accès des autres personnes jusqu'à ce qu'elle relève la tête. Si elle ne le fait jamais (et meurt en lisant son journal ...), les autres personnes attendront pour toujours, même si plusieurs guichets sont libres !

Les performances des algorithmes de ℓ-exclusion sont principalement évaluées avec le *temps d'attente* : le temps maximum pour qu'un processus demandeur entre en section critique. Ce temps est généralement mesuré en nombre de *rondes (asynchrones)* : informellement, une ronde est un facteur d'exécution qui termine une fois que le processus le plus lent a agi (*e.g.*, en exécutant sa section critique) et que le message le plus ancien a été livré. La propriété d'évitement de ℓ-interblocage est importante car elle permet d'exclure des solutions triviales inefficaces [4], *e.g.*, on peut remarquer que tout algorithme d'exclusion mutuelle vérifie les deux premières propriétés de la ℓ-exclusion. Notez, de plus, que l'évitement de ℓ-interblocage est

[†] Il a été financé en partie par les projets ANR DESCARTES (ANR-16-CE40-0023) et ESTATE (ANR-16-CE25-0009).

très intéressant dans les systèmes où le temps d'exécution des sections critiques est drastiquement supérieur au temps de communication. En effet, dans de telles situations, l'évitement de ℓ-interblocage permet de maximiser le taux d'utilisation des ressources.

Contribution. Cet article est un résumé étendu de [2]. Dans ce dernier, nous montrons tout d'abord qu'il existe des cas dans lesquels la propriété d'évitement de ℓ-interblocage implique une perte d'efficacité. Formellement, nous montrons le théorème suivant :

Théorème 1 *Tout algorithme réalisant la ℓ-exclusion avec la propriété d'évitement de ℓ-interblocage et $\ell < n$ a un temps d'attente en $\Omega(n-\ell)$ rondes dans un anneau asynchrone de n processus.*

Ce théorème apparaît contre-intuitif avec l'idée naïve qu'en utilisant ℓ ressources identiques, on est en droit d'espérer un temps d'attente en $O(\frac{n}{\ell})$ rondes. Or, le théorème précédent établit clairement que lorsque ℓ est très petit par rapport à n, obtenir un temps d'attente en $O(\frac{n}{\ell})$ rondes est en fait impossible. Au vu de ce résultat négatif, nous avons défini une variante simple du problème où nous remplaçons la propriété d'évitement de ℓ-interblocage par une propriété, appelée ici *temps d'attente efficace*, qui impose un temps d'attente en $O(\frac{n}{\ell})$ rondes, la borne asymptotique optimale. Ainsi, nous appelons *ℓ-exclusion avec temps d'attente efficace* la variante du problème de ℓ-exclusion qui en découle. Notez que la *ℓ-exclusion avec temps d'attente efficace* exclut aussi les solutions triviales inefficaces : par exemple, lorsque ℓ est grand, il est impossible d'avoir un temps d'attente en $O(\frac{n}{\ell})$ rondes en utilisant un algorithme d'exclusion mutuelle. Nous avons ensuite illustré cette nouvelle variante du problème en proposant deux algorithmes distribués asynchrones. Le premier suppose un réseau en anneau orienté et enraciné. Le second, une généralisation du premier, fonctionne dans tout réseau connexe identifié. Ces deux algorithmes sont autostabilisants. L'*autostabilisation* [3] est un paradigme permettant de concevoir des systèmes distribués tolérants aux *fautes transitoires*. Les fautes transitoires sont rares, de durée finie et affectent le contenu du composant du réseau où elles surviennent, *e.g.*, la corruption de la mémoire locale d'un processus ou du contenu d'un message en transit sont deux exemples de fautes transitoires. Après un nombre fini de fautes transitoires et en supposant que ces fautes n'aient pas affecté le code de l'algorithme, un système autostabilisant retrouve de lui-même et en temps fini un comportement correct, c'est-à-dire conforme à sa spécification. On appelle *phase de stabilisation* la période nécessaire au système pour qu'il retrouve un comportement correct. La durée maximale de cette phase est appelée *temps de stabilisation*. Notre solution pour anneaux a un temps de stabilisation en $O(n)$ rondes et nécessite $\Theta(\log n)$ bits de mémoire par processus. Notre solution pour réseaux connexes stabilise aussi en $O(n)$ rondes. Cependant, elle nécessite $\Theta(\Delta \log n)$ bits de mémoire par processus, où Δ est le degré du réseau. En outre, les deux solutions ne supposent pas de connaissances initiales sur le réseau (telles que sa taille ou son diamètre, par exemple). Nos deux solutions sont basées sur une circulation autostabilisante de ℓ jetons, chaque jeton accordant le droit d'accès à la section critique au processus qui le détient. Ainsi, dans nos algorithmes, nous assurons qu'une fois que le système a stabilisé, chaque processus reçoit un jeton au moins tous les $O(\frac{n}{\ell})$ rondes. D'où, une fois stabilisé, la propriété de *temps d'attente efficace* est trivialement réalisée. À notre connaissance, il n'existait jusqu'alors aucun algorithme autostabilisant asynchrone de ℓ-exclusion ayant un temps d'attente en $O(\frac{n}{\ell})$ rondes.

Plan. Dans la section 2, nous décrivons le modèle de calcul utilisé pour nos algorithmes. Dans la section 3, nous présentons uniquement notre solution autostabilisante pour anneaux enracinés et orientés. Dans la section 4 nous discutons des conséquences de nos résultats dans une perspective plus large.

2 Modèle à états

Le modèle à états est une abstraction du modèle à passage de messages, communément utilisé dans le domaine de l'autostabilisation, dans lequel les échanges d'informations sont émulés par le biais de *variables localement partagées* : chaque processus détient un nombre fini de variables dans lesquelles il peut lire et écrire ; de plus, il peut lire les variables de ses voisins dans le réseau. Nous noterons $p.x$ la variable x du processus p. L'*état* d'un processus est défini par la valeur de ses variables. Une *configuration* du système est définie par l'état de chacun de ses processus. L'exécution d'un algorithme est composée d'une succession de *pas atomiques* de calcul. À chaque pas de calcul, chaque processus détermine en fonction de son état et de celui de ses voisins s'il est *activable*, c'est-à-dire s'il doit modifier ses variables. Un adversaire, le *démon*,

choisit ensuite un sous-ensemble non vide de processus activables. En un pas atomique, les processus choisis sont alors *activés* et mettent à jour leurs variables. Nous supposons ici que les processus sont activés de manière asynchrone en supposant un démon *faiblement équitable*, *i.e.*, tout processus *continûment* activable est activé en temps fini. Dans ce modèle, la notion de *ronde* est reformulée comme suit : la première ronde d'une exécution termine dès lors que tous les processus continûment activable depuis le début de l'exécution ont exécuté au moins une action ; la seconde ronde commence à la fin de la première, *etc.*

3 Algorithme pour anneaux enracinés et orientés

Nous dénotons par $p_0, \ldots, p_{n-1}$ les n processus. Le réseau est un anneau orienté et enraciné, c'est-à-dire :

— p_0 ‡ est la *racine*.

— Pour tout $i \in [0..n-1]$, $p_{(i-1) \bmod n}$ est le *prédécesseur* de p_i. Il est aussi noté $Pred(i)$.

— Pour tout $i \in [0..n-1]$, $p_{(i+1) \bmod n}$ est le *successeur* de p_i. Il est aussi noté $Succ(i)$.

Notre algorithme consiste en deux tâches exécutées en parallèle :

— Nous divisons l'anneau en ℓ segments de taille $O(\frac{n}{\ell})$. Plus précisément, soit $x = n \bmod \ell$, nous construisons x segments de taille $\lceil \frac{n}{\ell} \rceil$ et $\ell - x$ de taille $\lfloor \frac{n}{\ell} \rfloor$.

— Nous réalisons une circulation d'un unique jeton à l'intérieur de chaque segment. Une fois que l'algorithme a stabilisé, chaque jeton parcourt tout son segment en $O(\frac{n}{\ell})$ rondes.

Cependant, notez que la stabilisation de la seconde tâche commence effectivement après que la première a stabilisé. Notez aussi que, conformément au théorème 1, notre algorithme ne satisfait pas la propriété d'évitement de ℓ-interblocage car la circulation de chaque jeton est restreinte à un segment : si dans le même segment, le jeton est en cours d'utilisation et qu'il y a d'autres processus demandeurs, les demandes de ces processus ne peuvent être satisfaites tant que le jeton n'est pas libéré et ce même si d'autres jetons (dans d'autres segments) sont inutilisés. Nous détaillons maintenant chacune des tâches.

Construction des segments. Cette construction est aussi gérée par deux sous-tâches parallèles :

— Chaque processus p_i calcule dans la variable $p_i.d$ sa distance à la racine en suivant le sens des liens successeurs. Une fois toutes les variables d correctement évaluées, la racine p_0 peut évaluer facilement la taille de l'anneau. En effet, cette taille est simplement égale $Pred(0).d + 1$.

— Chaque processus p_i évalue son rang dans son segment. Ce rang permettra à p_i de décider localement s'il est l'extrémité initiale ou finale ou un nœud interne de son segment.

La stabilisation de la seconde sous-tâche commence aussi effectivement après que la première a stabilisé.

La racine étant à distance 0 d'elle même, $p_0.d$ doit être affecté à 0. Ensuite, chaque processus non-racine p_i doit maintenir $p_i.d$ pour que $p_i.d$ soit égal à $Pred(i).d + 1$. Ainsi, en $O(n)$ rondes, toutes les variables d sont correctement assignées et, à partir de là, la racine connaît la taille de l'anneau. Nous utilisons ensuite ℓ (l'entrée du problème) et la taille de l'anneau pour calculer le rang des processus dans leur segment. Pour cela, nous utilisons trois variables supplémentaires pour chaque processus p_i :

— Dans $p_i.S$ nous propageons la connaissance de taille de l'anneau.

— Dans $p_i.X$ nous propageons le nombre de segments de taille $\lceil \frac{n}{\ell} \rceil$ qu'il reste à construire à partir de p_i.

— Dans $p_i.R$ nous stockons le rang de p_i.

Pour propager la taille de l'anneau à l'ensemble du réseau, p_0 maintient $p_0.S$ égal à $Pred(0).d + 1$. Les processus non-racines maintiennent leur variable S pour qu'elle soit égale à celle de leur prédécesseur. La racine maintient $p_0.X$ égal à $p_i.S \bmod \ell$, le nombre total de segments de taille $\lceil \frac{n}{\ell} \rceil$ à construire. Son rang $p_0.R$ doit être égal à 1. Chaque processus nonracine p_i calcule $p_i.X$ et $p_i.R$ en fonction de l'état de son prédécesseur. Si $Pred(i).X > 0$, alors $p_i.R$ doit être égal à $Pred(i).R \bmod \lceil \frac{p_i.S}{\ell} \rceil + 1$. Sinon, $p_i.R$ doit être égal à $Pred(i).R \bmod \lfloor \frac{p_i.S}{\ell} \rfloor + 1$. Ensuite, si $Pred(i).X > 0$ et $p_i.R$ a pris la valeur $\lceil \frac{p_i.S}{\ell} \rceil$, alors $p_i.X$ doit prendre la valeur $Pred(i).X - 1$ pour refléter qu'un nouveau segment de taille $\lceil \frac{p_i.S}{\ell} \rceil$ a été achevé. Sinon, $p_i.X$ doit prendre la valeur $Pred(i).X$. Ainsi, le système atteint en $O(n)$ rondes une configuration (*e.g.* figure 1), où chaque processus peut décider localement s'il est extrémité initiale ($p_i.R = 1$), ou finale (soit $P_i.R = \lceil \frac{p_i.S}{\ell} \rceil$, soit $P_i.R = \lfloor \frac{p_i.S}{\ell} \rfloor$ et $P_i.X = 0$), ou nœud interne (les autres cas) de son segment.

Circulation d'un jeton dans chaque segment. Dans un segment p_1, …, p_k, nous réalisons une circulation autostabilisante d'un unique jeton en utilisant une variable supplémentaire $p_i.J$ qui peut prendre

‡. *N.b.*, les indices sont uniquement utilisés pour raisonner : seul p_0 est distingué, les autres processus sont anonymes.

trois valeurs D, G, ou C [§]. Supposons que le système est dans une configuration où tous les processus vérifient $p_i.J = C$. L'extrémité initiale p_1 initie alors une circulation de jeton en changeant son état pour D : le jeton est propagé vers la droite. Séquentiellement l'état D est propagé vers la droite, jusqu'à ce que $p_k.J = D$. Dans ce cas, l'extrémité finale du segment p_k initie une propagation de l'état G vers la gauche en affectant $p_k.J$ à G. L'état G est propagé à gauche jusqu'à atteindre p_1. Dès que l'état G a atteint le prédécesseur de p_k, p_k peut initier une réinitialisation du segment en reprenant l'état C. Un nœud interne prend l'état C dès que son successeur a l'état C et que l'état G a été propagé à son prédécesseur. L'extrémité initiale reprend l'état C lorsque son successeur a l'état C. Dans ce cas, le système a retrouvé une configuration où tous les processus vérifient $p_i.J = C$ et p_1 peut ré-initier une circulation de jeton. Nous pouvons remarquer que les changements d'états de C à D et de D à G sont en exclusion mutuelle pour l'ensemble du segment. Ainsi, nous autorisons les processus à exécuter leur section critique durant ces actions uniquement. On remarque alors qu'un processus peut exécuter deux fois sa section critique par parcours et qu'un parcours est effectué en un nombre de rondes qui est fonction de la taille du segment, *i.e.*, $O(\frac{n}{\ell})$ rondes. D'après ce fonctionnement « normal », nous déduisons qu'après des fautes, au moins un processus $p_i \neq p_0$ du segment peut détecter localement une erreur en comparant son état à celui de son prédécesseur : p_i détecte une erreur lorsque $\langle Pred(i).J, p_i.J\rangle \in \{\langle C,D\rangle, \langle C,G\rangle, \langle G,D\rangle\}$. Dans ce cas, p_i réinitialise $p.J$ à C. Cela provoque une propagation de corrections vers la droite dans le segment. Ainsi, une fois la construction de segments stabilisée, chaque segment retrouve une configuration correcte en $O(\frac{n}{\ell})$ rondes.

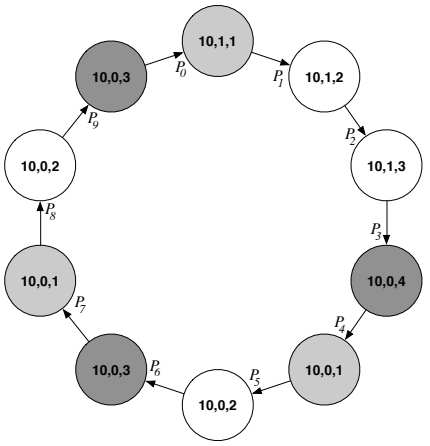

FIGURE 1: Anneau de $n = 10$ processus avec $\ell = 3$: l'étiquette à l'intérieur de chaque nœud p_i donne les valeurs de $p_i.S$, $p_i.X$ et $p_i.R$, dans cet ordre. Les couleurs donnent le statut de p_i dans son segment : gris clair pour extrémité initiale, blanc pour nœud interne et gris foncé pour extrémité finale.

Nous avons généralisé l'algorithme pour anneaux orientés enracinés afin qu'il fonctionne dans les réseaux connexes identifiés. Pour cela, nous utilisons une construction d'arbre couvrant autostabilisante. Puis, nous émulons l'algorithme pour anneaux en utilisant un anneau virtuel obtenu à partir du parcours en profondeur de l'arbre. Cet anneau virtuel a une taille de $2n-2$, ce qui nous permet de garder un temps d'attente en $O(\frac{n}{\ell})$ rondes. Cependant, l'émulation a un surcoût en mémoire : $O(\Delta \log n)$ bits par processus.

4 Discussion

L'*évitement de ℓ-interblocage* a été généralisée à la plupart des problèmes d'allocation de ressources sous le nom de *concurrence maximale* [1]. Nos résultats montrent que pour certains problèmes d'allocation de ressources, comme la ℓ-exclusion, la propriété de *concurrence maximale* est incompatible avec l'obtention d'un temps d'attente asymptotiquement optimal. Une question naturelle est alors : vaut-il mieux assurer la concurrence maximale ou préférer une solution avec un temps d'attente asymptotiquement optimal ? La réponse (normande) est que cela dépend du contexte ! En effet, dans le cas où les temps de passage en section critique sont (très) supérieurs aux temps de communication, il faut privilégier la concurrence maximale pour maximiser le taux d'utilisation des ressources, qui sont rarement disponibles. Dans le cas opposé, toutes les ressources sont rapidement rendues aux systèmes, donc il vaut mieux accélérer le temps d'accès aux ressources, en privilégiant une solution avec un temps d'attente « efficace ».

Références

[1] K. Altisen, S. Devismes, and A. Durand. Concurrency in snap-stabilizing local resource allocation. *J. Parallel Distrib. Comput.*, 102 :42–56, 2017.

[2] F. Carrier, A. K. Datta, S. Devismes, and L. L. Larmore. Self-stabilizing ℓ-exclusion revisited. In *ICDCN*, 2015.

[3] E. W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Commun. ACM*, 17(11) :643–644, 1974.

[4] M. J. Fischer, N. A. Lynch, J. E. Burns, and A. Borodin. Resource allocation with immunity to limited process failure (preliminary report). In *FOCS*, pages 234–254, 1979.

§. *N.b.* le troisième algorithme de Dijkstra [3] fonctionne aussi avec trois états sur une chaîne, mais sous un démon séquentiel !