K-parmi-L exclusion auto-stabilisante

Stéphane Desvimes, Florian Horn, Ajoy Datta, Lawrence Larmore

K-parmi-L exclusion auto-stabilisante[†]

A.K. Datta[1] S. Devismes[2] F. Horn[3] L.L. Larmore[1]

[1]*School of Computer Science, University of Nevada, Las Vegas*
[2]*CNRS, LRI, Université de Paris-Sud (Paris XI), Orsay*
[3]*LIAFA, CNRS UMR 7089, Université Denis Diderot, Paris VII*

Dans cet article, nous adoptons une démarche à la fois intuitive et modulaire afin de résoudre de manière auto-stabilisante le problème de la K-*parmi*-L *exclusion* dans un anneau unidirectionnel enraciné. Cette démarche nous permet d'obtenir un protocole élégant, plus simple que les solutions précédentes, sans pour autant détériorer l'efficacité.

Keywords: Tolérance aux pannes, K-*parmi*-L *exclusion*, auto-stabilisation

1 Introduction

Le problème de base d'allocation de ressources est la gestion de ressources *concurrentes*, comme une imprimante ou une variable partagée. L'usage de telles ressources par un agent affecte leur disponibilité pour les autres utilisateurs. Dans les deux cas précités, un agent au maximum doit avoir accès simultanément à la ressource via une *section critique* du code. Les protocoles associés doivent donc garantir la propriété d'*exclusion mutuelle* : la section critique ne peut être exécutée que par un processeur à la fois. La L-exclusion est une généralisation de cette propriété où L processeurs peuvent exécuter la section critique simultanément. Un nombre L d'unités d'une même ressource (par exemple, un pool d'adresses IP) peuvent ainsi être allouées. Ce modèle peut encore être généralisé pour considérer des demandes dont le coût est variable (par exemple, de la bande passante pour du *streaming* audio ou vidéo). La K-*parmi*-L *exclusion* permet de gérer des demandes qui utilisent jusqu'à K unités d'une même ressource simultanément (avec $1 \leq \mathtt{K} \leq \mathtt{L}$). Tout protocole de K-*parmi*-L *exclusion* vérifie les trois propriétés suivantes :

- **sûreté :** Chaque unité de la ressource partagée peut être utilisée par au plus un processeur à la fois et au plus L unités peuvent être utilisées simultanément.
- **vivacité :** Toute demande d'au plus K unités doit être satisfaite en un temps fini.
- **efficacité :** Le plus grand nombre possible de demandes doit être satisfait simultanément.

Travaux précédents. Deux types d'approches sont généralement utilisés pour résoudre les problèmes d'allocation de ressources : les *protocoles à permissions* et les *circulations de jetons*. Des mécanismes de permissions sont notamment utilisés dans [Ray91]. Deux protocoles (auto-stabilisants) de K-*parmi*-L *exclusion* sont proposés dans [DHV03b, DHV03a]. Ils fonctionnent dans un anneau unidirectionnel enraciné et utilisent une circulation de jetons.

Contributions. La contribution de cet article est double. D'une part, nous proposons un protocole de K-*parmi*-L *exclusion* fonctionnant dans un anneau unidirectionnel enraciné. Ce protocole est *auto-stabilisant* [Dij74] : confronté à une période de pannes transitoires susceptibles de modifier arbitrairement l'état des composants du réseau (mémoire des processeurs, contenu des messages...), il retrouve de lui-même et en un temps fini un comportement normal après que les pannes ont cessé. Le temps de stabilisation de notre protocole est meilleur que ceux des deux solutions précédentes : il stabilise en $4n$ rondes[‡] alors que les autres solutions de la littérature stabilisent en $8n$ rondes [DHV03b] et $5n$ rondes [DHV03a].

[†]Par manque de place, nous ne présentons pas la preuve formelle du protocole présenté dans cet article. Cette preuve est disponible en ligne (`http://www.lri.fr/~devismes/rapports/klexclu.pdf`).

[‡] Une ronde correspond au temps nécessaire pour que le processeur le plus lent agisse. Chaque action comporte trois étapes atomiques : (1) les messages présents dans les canaux sont reçus, (2) le processeur change son état et (3) envoie des messages si nécessaire.

D'autre part, notre méthodologie nous permet d'obtenir une solution élégante, plus simple que les solutions précédentes : nous présentons le protocole de manière intuitive et modulaire en présentant séparément un algorithme qui attribue l'usage des ressources de manière à assurer une K-*parmi*-L *exclusion* non auto-stabilisante, et un module d'auto-stabilisation pour gérer les pannes transitoires.

2 Le protocole

Dans la suite, n représente la taille de l'anneau et nous numérotons les processeurs de 0 à $n-1$. Cette numérotation est utilisée uniquement pour faciliter la présentation. Chaque processeur i connaît son *prédécesseur* $i-1 \pmod n$ et son *successeur* $i+1 \pmod n$ dans l'anneau. De plus, nous distinguons un processeur particulier, le processeur 0, comme la *racine* du réseau. Chaque processeur peut envoyer des messages à son successeur et en recevoir par son prédécesseur. Les liens de communication sont supposés FIFO, fiables (après la fin des pannes transitoires) et à capacité bornée ($\mathcal{C}_{\texttt{max}}$ étant la capacité maximale).

Une solution simple non auto-stabilisante. Le principe de base de notre protocole est de faire circuler L *jetons ressource* dans le réseau : le processeur i reçoit des jetons de son prédécesseur $(i-1)$ et en envoie à son successeur $(i+1)$. Les demandes sont représentées par les variables Need : si le processeur i a besoin de j unités de ressource, $\texttt{Need}_i$ est égal à j. Pour chaque processeur, il y a également une seconde variable (Cnt), qui compte les jetons « réservés ». Tant que $\texttt{Cnt}_i$ est inférieur à $\texttt{Need}_i$, le processeur i collecte les jetons qu'il reçoit en incrémentant $\texttt{Cnt}_i$. Lorsque $\texttt{Cnt}_i = \texttt{Need}_i$, il entre en section critique. Une fois que la section critique est terminée, le processeur i envoie les jetons qu'il détient à son successeur, puis remet à zéro $\texttt{Cnt}_i$ et $\texttt{Need}_i$. Lorsque le processeur i n'est pas demandeur (quand $\texttt{Cnt}_i = \texttt{Need}_i$), les jetons reçus sont transmis directement à son successeur $i+1$.

Malheureusement, ce protocole est trop simple pour garantir la vivacité. Dans le cas présenté dans la figure 1, il y a cinq processeurs et cinq unités de ressource ($\texttt{L} = 5$) avec des demandes de taille inférieure ou égale à trois ($\texttt{K} = 3$). Considérons le cas où chaque jeton est en transit dans un lien différent et chaque processeur demande deux jetons (partie gauche de la figure 1). Chaque processeur collecte le jeton provenant de son prédécesseur puis attend un second jeton (partie droite de la figure 1), sans jamais relâcher le sien : le système est en interblocage.

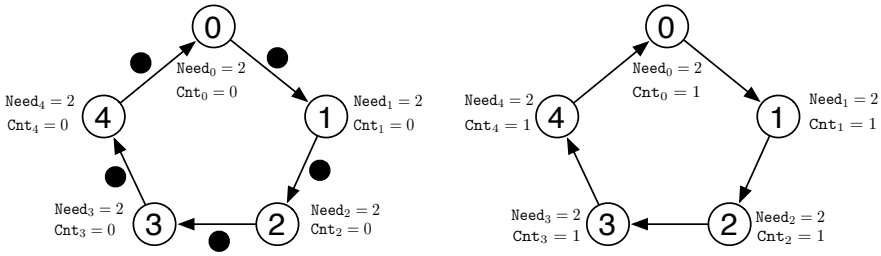

FIG. 1: Cas d'interblocage.

Nous corrigeons ce cas d'interblocage en ajoutant un jeton particulier appelé *jeton pousseur*. Ce jeton ne peut pas être utilisé comme une unité de ressource par les processeurs. Il circule en permanence dans l'anneau et empêche un processeur qui n'est pas en section critique de garder des jetons : lorsque le processeur i reçoit le *jeton pousseur*, s'il possède des jetons ressource sans être en section critique ($0 < \texttt{Cnt}_i < \texttt{Need}_i$), il les envoie immédiatement à son successeur $i+1$. Dans tous les cas, il envoie ensuite le jeton pousseur.

Le jeton pousseur empêche les cas d'interblocage globaux du système. Cependant, des cas locaux de *famine* peuvent encore se produire. La figure 2 présente un cas de famine, pour un protocole de 2-*parmi*-3 *exclusion* dans un anneau de trois processeurs. Dans la configuration (a), chaque processeur est demandeur : les processeurs 0 et 1 demandent un jeton ressource et le processeur 2 en demande deux. Il y a un jeton ressource dans chaque lien et le jeton pousseur est en transit entre le processeur 2 et le processeur 0 derrière un jeton ressource. Chaque processeur reçoit un jeton ressource et le conserve (configuration (b)). Si le jeton pousseur passe par les processeurs 0 et 1 avant qu'ils aient fini d'exécuter leur section critique, on obtient les configurations (c) puis (d). Le processeur 2 reçoit alors le jeton pousseur, qui le force à relâcher

son unique jeton ressource (configuration (e)). Les processeurs 0 et 1 peuvent alors terminer leur section critique, relâcher leur jeton, puis redevenir demandeurs d'un jeton. On arrive alors à la configuration (f) qui est identique à la configuration (a). Ce processus peut se répéter indéfiniment, sans que la demande du processeur 2 soit jamais satisfaite.

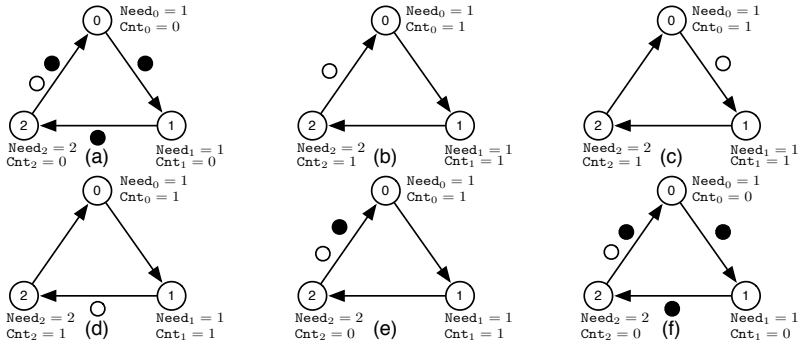

FIG. 2: Cas de famine (le jeton blanc correspond au jeton pousseur, les jetons noirs sont les jetons ressource).

Pour résoudre ce problème, nous ajoutons un *jeton priorité*, qui annule l'effet du jeton pousseur. Comme ce dernier, il ne peut pas être utilisé comme une unité de ressource. Les processeurs qui le reçoivent le retransmettent immédiatement à leur successeur à moins qu'ils n'aient une demande insatisfaite. Dans ce cas, le jeton priorité est conservé par le processeur jusqu'à satisfaction de la demande : il ne sera relâché que lorsque le processeur entrera en section critique. Un processeur qui possède le jeton priorité n'est pas tenu de relâcher ses jetons s'il reçoit le jeton pousseur : il renvoie uniquement le jeton pousseur à son successeur.

Avec ces trois types de jetons, nous obtenons un protocole simple mais non auto-stabilisant de K-*parmi*-L *exclusion*. Ci-dessous, nous présentons le module rendant ce protocole auto-stabilisant.

Un régulateur auto-stabilisant. Les pannes transitoires peuvent avoir deux types d'effets sur notre protocole non auto-stabilisant : (1) modifier les demandes et (2) modifier le nombre de jetons.

Le premier cas ne pose pas de problème : si un processeur demande plus de K jetons, cela est dû à une corruption de sa mémoire. Le processeur annule alors cette demande. De même, si sa demande est modifiée mais ne dépasse pas K, la perte de sûreté est temporaire : le système retrouvera un fonctionnement normal une fois ces « mauvaises » demandes traitées. Ce premier cas se règle donc en au plus 1 ronde.

Le deuxième cas est plus complexe : il faut ajouter un mécanisme auto-stabilisant permettant de réguler le nombre de jetons dans le réseau. Pour ce faire, nous utilisons le protocole auto-stabilisant de *counter flushing* de Varghese [Var00]. Ce protocole converge partant d'une configuration quelconque vers une configuration où un unique *jeton privilège* circule dans le réseau (ce protocole est présenté dans la dernière partie de l'article). Nous utilisons le jeton privilège comme point de repère : entre deux réceptions du privilège, le processeur racine compte le nombre de jetons de chaque type (ressource, pousseur et priorité) et ajuste ce nombre si celui-ci n'est pas adéquat.

Considérons le problème du comptage des jetons ressource (le comptage des autres types de jetons dérive de cette méthode). Lorsque la racine reçoit le privilège, elle initialise un compteur avec le nombre de jetons qu'elle détient. Ce compteur est ensuite incrémenté à chaque fois que la racine reçoit un jeton ressource jusqu'à ce que le jeton privilège ait fait un tour complet de l'anneau. Lorsque la racine reçoit le jeton privilège par son prédécesseur, le privilège a fait un tour complet. La racine ajoute alors la valeur de son compteur de jetons ressource au nombre de jetons ressource « doublés » par le privilège, ce nombre étant stocké directement sur le jeton privilège. Le jeton privilège « double » des jetons ressources dans la situation suivante : lorsqu'un processeur a réservé des jetons ressource et qu'il reçoit le jeton privilège, il garde ses jetons ressource mais renvoie le jeton privilège. Pour ne pas oublier ces jetons dans le décompte final, le processeur met à jour le champ « nombre de jetons doublés » du jeton privilège avant de le renvoyer.

Une fois le protocole de *counter flushing* stabilisé (après au plus $2n$ rondes [Var00]), le bon compte est connu de la racine après chaque tour complet du jeton privilège (n rondes). Trois cas sont alors possibles :

- *Le nombre de jetons est correct.* Dans ce cas, le système est stabilisé.

- *Le nombre de jetons est insuffisant.* Dans ce cas, la racine crée les jetons manquants à la fin du tour et le système est stabilisé.
- *Il y a trop de jetons.* Dans ce cas, on réinitialise complètement le réseau. Pour cela, le jeton privilège est marqué avec un drapeau spécial. La racine envoie le jeton privilège marqué, supprime ses jetons réservés ainsi que tous les jetons qu'elle reçoit jusqu'au retour du jeton privilège. A la réception du jeton privilège marqué, les autres processeurs détruisent leurs jetons réservés. Lorsque le privilège revient à la racine (n rondes), il n'y a donc plus de jeton dans le réseau, la racine crée alors exactement L jetons ressource et le système est stabilisé.

En étendant cette technique aux autres types de jetons, le système stabilise en au plus $4n$ rondes (*cf.* `http://www.lri.fr/~devismes/rapports/klexclu.pdf` pour la preuve).

Le protocole du jeton privilège. Le protocole de *counter flushing* proposé par Varghese simule la circulation d'un jeton unique via des messages contenant une variable entière. Les processeurs décident localement si un message est valide — c'est-à-dire qu'il représente effectivement la réception du jeton privilège — ou s'il ne l'est pas — auquel cas il est totalement ignoré. Le contrôle local se fait en comparant des valeurs variant de 0 à $n(\mathcal{C}_{\texttt{max}}+1)-1$. Chaque processeur i possède une variable C_i. Il envoie *périodiquement* à son successeur des messages contenant cette valeur. Les effets de la réception d'un message contenant la valeur c sont différents pour la racine et pour les autres processeurs :

- La racine considère qu'un message est un jeton privilège valide si $c = C_0$. Lorsqu'elle reçoit un tel message elle incrémente C_0 modulo $n(\mathcal{C}_{\texttt{max}}+1)$.
- Le processeur $i \neq 0$ considère qu'un message est un jeton privilège valide si $c \neq C_i$. Lorsqu'il reçoit un tel message, il donne à C_i la valeur c.

Ce protocole converge car le domaine des variables C_i est supérieur (d'une unité) au nombre de valeurs maximales initialement présentes dans le réseau. La racine génère donc en un temps fini une nouvelle valeur de message. Lorsque cette valeur revient à la racine après avoir traversé tous les processeurs, la racine est le seul processeur du système à détenir le jeton privilège.

3 Conclusion

Nous avons suivi une démarche intuitive pour construire un protocole auto-stabilisant de K-*parmi*-L *exclusion* à la fois simple et efficace. Nous avons obtenu cette simplicité en partant d'une solution naïve : une circulation de L jetons ressource. Nous avons ensuite ajouté deux jetons spéciaux — le jeton pousseur et le jeton priorité — pour lever les problèmes d'interblocages et de famines. Enfin, nous avons composé notre solution non auto-stabilisante avec un module auto-stabilisant régulant le nombre de jetons du système permettant ainsi de stabiliser l'ensemble. Le temps de stabilisation de notre solution (au plus $4n$ rondes) est meilleur que pour les solutions précédentes, ce qui montre que notre approche intuitive est efficace. Cette méthodologie peut être appliquée à d'autres types de problèmes. De plus, composé avec un protocole auto-stabilisant de parcours en profondeur, notre protocole peut être étendu aux réseaux de topologie quelconque. En effet, le parcours en profondeur permet de construire un anneau virtuel dans un réseau quelconque.

Références

[DHV03a] A.K. Datta, R. Hadid, and V. Villain. A new self-stabilizing k-out-of-l exclusion algorithm on rings. In *Self-Stabilizing Systems*, pages 113–128, 2003.

[DHV03b] A.K. Datta, R. Hadid, and V. Villain. A self-stabilizing token-based k-out-of-l exclusion algorithm. *Concurrency and Computation : Practice and Experience*, 15 :1069–1091, 2003.

[Dij74] Edsger W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Commun. ACM*, 17(11) :643–644, 1974.

[Ray91] M. Raynal. A distributed solution to the k-out of-m resources allocation problem. In *ICCI*, pages 599–609, 1991.

[Var00] G. Varghese. Self-stabilization by counter flushing. *SIAM J. Comput.*, 30(2) :486–510, 2000.