Renommage et Splitters en présence d'une Majorité de Pannes

David Bonnin, Corentin Travers

Renommage et Splitters en présence d'une Majorité de Pannes

David Bonnin et Corentin Travers

LaBRI, Université de Bordeaux, France
prenom.nom@labri.fr

Les *splitters* de Moir et Anderson sont des objets partagés simples, implémentables à l'aide de registres à lecture/écriture, qui retournent une *direction* parmi $\{right, down, stop\}$. Tous les processus accédant à cet objet ne retournent pas la même direction, et au plus l'un d'entre eux obtient stop. Dans leur version *one-shot* autant que dans leur version *long-lived*, les splitters sont les briques de base d'élégants algorithmes de renommage en mémoire partagée. Dans un système par envoi de messages avec une stricte minorité de processus susceptible de tomber en panne, les splitters peuvent être implémentés en simulant d'abord des registres partagés. Cela n'est plus le cas si au moins la moitié des processus peuvent tomber en panne. Nous définissons et implémentons des splitters one-shot et des long-lived adaptés au modèle avec une majorité de pannes. Nos splitters généralisés conservent la plupart des propriétés des splitters classiques, excepté qu'on ne garantit qu'au plus $\lfloor \frac{n}{n-f} \rfloor$ processus retournent *stop*, où n est le nombre de processus et $f < n$ est une borne supérieure sur le nombre de pannes. Nous adaptons les grilles de splitters de Moir et Anderson pour résoudre des variantes du renommage one-shot et long-lived pour lesquelles au plus $k = \lfloor \frac{n}{n-f} \rfloor$ processus peuvent obtenir le même nom. Cet article est un résumé de [5], qui a été accepté et publié à la conférence ICDCN 2015.

Keywords: Tolérance aux pannes, Envoi de messages, Renommage, Splitter

1 Introduction

Contexte On considère un système de n processus asynchrones qui communiquent par envoi de messages. Bien que le système de communication soit supposé sûr, les processus peuvent tomber en panne (*crash*). Lorsque le nombre maximal de pannes f est borné par $\lfloor \frac{n-1}{2} \rfloor$, c'est-à-dire seule une stricte minorité des processus peut tomber en panne, on sait que ce modèle est aussi fort que le modèle par mémoire partagée, car des registres partagés peuvent être simulés dans ce premier modèle [3]. En particulier, cette simulation permet à n'importe quel algorithme en mémoire partagée d'être automatiquement implémenté en envoi de messages. Ce n'est plus le cas lorsqu'une majorité des processus peut tomber en panne. Dans ce contexte, trouver des alternatives utiles aux briques de base de la mémoire partagée est un défi, que nous relevons à travers deux abstractions simples que sont les *splitters* et le *renommage*.

Splitters et renommage Un *splitter* [9] est un objet partagé disposant d'une opération SPLITTER(). Cette opération retourne une *direction* parmi $\{right, down, stop\}$. Un splitter a un lien avec l'exclusion mutuelle [8], car au plus un des processus qui appellent ce splitter peut le *capturer*, c'est-à-dire obtenir *stop*. Comme son nom l'indique, un splitter garantit une séparation des processus en garantissant que toutes les invocations ne retournent pas la même direction *right* ou *down*. Les splitters ont été introduits par Moir et Anderson pour résoudre le renommage en mémoire partagée [9], s'inspirant de travaux de Lamport [8].

Le M-renommage [4] est un problème distribué fondamental, dans lequel des processus ayant des identifiants uniques dans un large ensemble cherchent à obtenir des noms distincts dans un domaine plus petit, de taille M. Un algorithme de renommage simple et élégant, basé sur les splitters a été présenté dans [9].

Les splitters et le renommage ont aussi une version *long-lived*. Dans le problème du *M-renommage long-lived*, les processus obtiennent et libèrent des noms en continu dans un ensemble de taille M de manière à ce qu'aucun nom ne soit simultanément capturé par plusieurs processus. De manière similaire, un *splitter long-lived* peut être invoqué à multiples reprises. Il dispose alors d'une opération supplémentaire RELEASE() qui permet à un processus ayant capturé le splitter de le libérer. À tout instant, un splitter long-lived est capturé par au plus un

processus, et lorsqu'il n'est pas occupé (c'est-à-dire qu'il n'est pas capturé et n'a pas d'opération en cours), il se comporte comme sa version one-shot.

Contributions Cet article définit et implémente des variantes one-shot et long-lived des splitters et du renommage adaptées pour un système asynchrone par envoi de messages avec une majorité de pannes potentielles. Les résultats sont divisés de la manière suivante : Nous définissons les *k-splitter* et le (M,k)-*renommage* en section 2. Une implémentation des k-splitters one-shot et long-lived, avec $k \geq \lfloor \frac{n}{n-f} \rfloor$, est disponible dans la section 3.Une solution du (M,k)-renommage one-shot, avec $k \geq \lfloor \frac{n}{n-f} \rfloor$, et pour $M = O(n^{3/2})$, est présentée dans la section 4. Enfin, la version long-lived du (M,k)-renommage est résolue avec $k \geq \lfloor \frac{n}{n-f} \rfloor$ et $M = O(n^8)$ dans la section 4.

Travaux connexes Le renommage est un problème distribué important, et de nombreux articles ont été dédiés à son étude, principalement dans des systèmes asynchrones par mémoire partagée et avec pannes, comme par exemple [1, 4, 6, 9]. Un algorithme asynchrone par envoi de message avec un espace de nom de taille $n+f$ apparaît dans [4]. Cet algorithme tolère $f < \frac{n}{2}$ pannes. Sous cette condition, les algorithmes de renommage en mémoire partagée peuvent être traduits en algorithmes par envoi de messages car les registres peuvent être simulés tant que $f < \frac{n}{2}$ [3].

Lorsque $f \geq \frac{n}{2}$, des problèmes de partitionnement peuvent survenir, car le système peut finir séparé en plusieurs groupes de $n-f$ processus dont les messages d'un groupe à l'autre sont arbitrairement ralentis, de sorte que chaque groupe pense que les autres sont tombés en panne, ce qui est possible. Or, d'après le théorème CAP [7], aucun algorithme asynchrone résistant au partitionnement ne peut garantir à la fois la cohérence (ici l'unicité des noms) et la terminaison (tout processus qui ne tombe pas en panne finit par obtenir un nouveau nom). On affaiblit alors la cohérence en autorisant les noms à être partagés entre jusqu'à $\lfloor \frac{n}{n-f} \rfloor$ processus. En un sens, cet article peut être vu comme partie d'une question plus large qui est de savoir jusqu'à quel point la cohérence doit être affaiblie quand un algorithme doit résister à un certain nombre de partitions.

Les splitters sont des briques de base en mémoire partagée permettant des algorithmes de renommage adaptatifs [1, 2, 9]. Un algorithme de renommage est adaptatif si sa complexité en temps/espace ainsi que la taille de son espace des noms dépend du nombre de processus essayant d'obtenir des nouveaux noms. Les algorithmes à base de splitters de cet article sont eux aussi adaptatifs dans la taille de l'espace des noms.

2 Modèle et définitions

Modèle de calcul On se place dans un système par envoi de message classique, tel que décrit dans [10] par exemple, et composé de n processus asynchrones $\{p_1, \ldots, p_n\}$. Les processus peuvent communiquer en s'échangeant des messages via un réseau complet de canaux de communication sûrs, *FIFO*, et asynchrones. Ce qui signifie que les messages envoyés par p_i à p_j seront tous reçus, dans le même ordre qu'ils ont été envoyés, en temps fini mais inconnu (et sans borne connue).

Les processus peuvent *tomber en panne*, c'est-à-dire spontanément arrêter d'exécuter leur algorithme. Au cours d'une exécution, un processus sera dit *fautif* s'il tombe en panne, et *correct* sinon. f désigne une borne supérieure sur le nombre maximum de processus susceptibles de tomber en panne au cours d'une exécution. Chaque processus connaît n et f, mais aucun ne connaît les indices i utilisés dans la notation p_i.

k-Splitters Comme évoqué dans l'introduction, un *k-splitter one-shot* dispose d'une opération SPLITTER() qui peut être invoquée au plus une fois par chaque processus. Elle prend comme paramètre l'identifiant du processus et retourne une valeur dans $\{right, down, stop\}$, en vérifiant les propriétés :

1. Chaque invocation de SPLITTER(id_i) par un processus p_i correct retourne.
2. Si un seul processus invoque SPLITTER(id), et que ce processus est correct, il obtient *stop*.
3. Si p processus invoquent SPLITTER(), au plus $p-1$ d'entre eux obtiennent *down* et au plus $p-1$ d'entre eux obtiennent *right*.
4. Parmi les processus invoquant SPLITTER(), au plus k obtiennent *stop*.

On dit qu'un processus qui obtient *stop* a *capturé* le splitter.

Un *k-splitter long-lived* dispose d'une opération supplémentaire RELEASE() qui permet aux processus de libérer le splitter après l'avoir capturé. On ne considère que des *exécutions bien formées* dans lesquelles (1) chaque

processus a au plus une opération en cours à tout instant et (2) chaque invocation de RELEASE() est précédée d'une invocation de SPLITTER() par le même processus et ayant retourné *stop*. À tout instant dans l'exécution d'un splitter, ce dernier est *occupé* si un processus p est *occupé avec* le splitter, c'est-à-dire si

— un processus p a invoqué SPLITTER() et n'a pas encore obtenu de réponse, ou,

— p a invoqué SPLITTER(), obtenu *stop*, et l'appel correspondant à RELEASE() n'a pas encore terminé.

Une *période occupée* est le plus grand intervale de temps durant lequel le splitter est *occupé*. Considérons un algorithme $\mathcal{S}$ implémentant un k-splitter long-lived, et composé de n algorithmes locaux $\mathcal{S}_1, ..., \mathcal{S}_n$, que l'on suppose à *information complète*, c'est-à-dire que chaque message envoyé contient l'état complet σ_i de $\mathcal{S}_i$. Pour chaque état σ_i, le splitter est considéré *occupé* ou non. Une période de temps est dite *vide* si durant cette période, chaque $\mathcal{S}_i$ considère le splitter comme non occupé, et si chaque σ_i contenu dans un message *utile* considère aussi le splitter comme non occupé. Un message *utile* au temps τ est un message envoyé mais non reçu tel que, si ce message venait à être reçu immédiatement après τ par p_i, ce message modifierait l'état actuel de $\mathcal{S}_i$. On peut noter qu'un splitter est initialement dans une période *vide*.

Un k-splitter long-lived a les propriétés suivantes.

1. Chaque invocation de SPLITTER(id_i) ou RELEASE() par un processus correct retourne.
2. À tout instant, le splitter est capturé par au plus k processus.
3. Durant toute période *occupée* immédiatement précédée par une période *vide*, toutes les invocations de SPLITTER() ne retournent pas la même direction *down* ou la même direction *right*.
4. Si un seul processus invoque SPLITTER() chacune de ces invocations retourne *stop*.
5. Chaque période de temps durant laquelle toute invocation de SPLITTER() retourne *down* est finie.

À noter que la propriété 3 implique que si, dans une période occupée immédiatement précédée par une période vide, un unique processus accède au splitter, l'appel à SPLITTER() retourne *stop*.

Renommage Un objet de (M,k)-*renommage one-shot* est accessible par une opération GET-NAME(id). Cette opération retourne un nouveau *nom* dans l'ensemble $[1..M]$, avec les propriétés suivantes :

— (Terminaison) Toute invocation de GET-NAME(id_i) par un processus correct retourne.

— (k-unicité) Pour tout nom $y \in [1..M]$, au plus k invocations de GET-NAME() retournent y.

Un objet de (M,k)-*renommage long-lived* dispose d'une opération supplémentaire RELEASE(). On ne considère que des exécutions *bien formées* dans lesquelles chaque processus alterne entre des appels à GET-NAME() et à RELEASE(), en commençant par GET-NAME(). On dit alors qu'un nom $y \in [1..M]$ est *capturé* par p_i si ce dernier a obtenu y d'un appel à GET-NAME() et que l'appel suivant à RELEASE() par p_i n'a pas encore retourné. Une implémentation de (M,k)-renommage long-lived a la même propriété de terminaison que la version one-shot, et doit en plus vérifier :

— (k-unicité long-lived) Pour tout nom $y \in [1..M]$ et à tout instant, y est capturé par au plus k processus.

3 k-splitters

Un algorithme implémentant un $\lfloor \frac{n}{n-f} \rfloor$-splitter one-shot est présenté dans la version longue [5]. Cet algorithme s'inspire de l'implémentation des splitters *classiques* de [9], qui fonctionnent approximativement de la façon suivante : Un processus qui entre dans le splitter commence par écrire son identifiant dans un registre partagé, puis vérifie si une *porte* (registre partagé booléen) est ouverte. Si la porte est fermée, le processus retourne *right*, sinon il ferme la porte, puis vérifie le registre des noms. Si son nom est encore présent, il retourne *stop*, sinon il retourne *down*. Notre algorithme est similaire, bien que n'utilisant pas à proprement parler de registres partagés (impossibles à implémenter en présence d'une majorité de pannes), et utilise $N = \lfloor \frac{n}{n-f} \rfloor + 1$ *portes* au lieu d'une seule. Une telle multiplication des portes permet de limiter le nombre de *stop* qui peuvent être retournés, en réduisant à chaque porte le nombre de processus pouvant arriver jusqu'à ce point tout en ayant leur identifiant suffisamment présent dans le système pour être considéré comme dans le registre virtuel de noms. La preuve qu'un tel algorithme implémente bien un $\lfloor \frac{n}{n-f} \rfloor$-splitter one-shot est elle aussi présente dans [5].

De même, un algorithme implémentant un $\lfloor \frac{n}{n-f} \rfloor$-splitter long-lived est décrit dans [5], s'inspirant encore une fois de l'algorithme classique des splitters long-lived de [9]. Cet algorithme est semblable à celui de la version one-shot, si ce n'est que les processus posent un *cadenas* personnel sur la porte plutôt que de la fermer de manière

anonyme. Ensuite, lorsqu'un processus retourne autre chose que *stop*, il enlève son cadenas de la porte, de manière à ce qu'elle s'ouvre à nouveau une fois tous les cadenas retirés. Enfin, un processus qui appelle l'opération RELEASE() enlève son cadenas. Une fois de plus, notre algorithme est similaire, en utilisant $N = \lfloor \frac{n}{n-f} \rfloor + 1$ portes, chacune fermée par les cadenas des processus, et tous les cadenas étant enlevés en même temps lors du retour d'un processus, ou de l'appel à RELEASE(). La preuve qu'un tel algorithme implémente correctement un $\lfloor \frac{n}{n-f} \rfloor$-splitter long-lived est disponible dans [5].

4 k-Renommage

Le $(M,1)$-renommage one-shot peut être implémenté à l'aide d'un réseau de 1-splitters, comme présenté dans [2, 9]. Chaque splitter d'une telle grille est implémenté indépendamment et associé à un entier unique dans $[1,M]$. Ensuite, l'unicité de la réponse *stop* pour chaque splitter permet à un processus ayant obtenu *stop* de retourner l'entier du splitter comme nouveau nom, garantissant une unicité de ce nom. Comme, entre les propriétés des k-splitters et des 1-splitters, seule l'unicité est modifiée en k-unicité, les grilles de 1-splitters utilisées pour du $(M,1)$-renommage peuvent être reproduites à l'aide de k-splitters, et implémentent ainsi du (M,k)-renommage one-shot. De cette manière, on peut donc résoudre un $(O(n^2), \lfloor \frac{n}{n-f} \rfloor)$-renommage via la grille de [9], et un $(O(n^{3/2}), \lfloor \frac{n}{n-f} \rfloor)$-renommage avec la grille décrite dans [2]. Enfin, on peut noter que l'algorithme de renommage de [4], si utilisé dans un modèle à majorité de pannes, implémente un $(n+f, \lfloor \frac{n}{n-f} \rfloor)$-renommage one-shot, mais cet algorithme ne fonctionne pas pour le cas long-lived.

De même que pour la version one-shot, une grille de splitters long-lived permet d'implémenter du $(O(n^2),1)$-renommage long-lived, comme présenté dans [9]. Malheureusement, une des propriétés des splitters long-lived classiques, qui limite le nombre de réponses *right* et *down*, ne peut pas être conservée en présence d'une majorité de pannes. Ainsi, les k-splitters long-lived ont une propriété semblable mais plus faible que leur version usuelle. Plus précisément, les périodes *occupées* ne limitent pas toutes le nombre de réponses non-*stop*, contrairement à la propriété habituelle, et seules ces périodes précédées de périodes *vides* le font. Cette affaiblissement d'une propriété fait que les grilles habituelles ne suffisent plus à implémenter du (M,k)-renommage avec des k-splitters.

Afin de résoudre ce problème via une nouvelle grille, et pour utiliser la faible propriété de ces objets, il convient d'utiliser une abstraction liant les implémentations individuelles des k-splitters qui composent la grille. Un tel lien entre les splitters permet de borner le nombre de splitters n'étant pas dans une période *vide*, comme expliqué en détails dans [5]. Enfin, à l'aide de cette abstraction, on peut montrer qu'une grille similaire à celle de [9] de taille suffisamment grande peut suffire à implémenter du (M,k)-renommage long-lived. [5] présente ainsi plus en profondeur une solution pour un $(O(n^8), \lfloor \frac{n}{n-f} \rfloor)$-renommage long-lived, à l'aide d'une grille de largeur $O(n^4)$.

Références

[1] Y. Afek, G. Stupp, and D. Touitou. Long lived adaptive splitter and applications. *Distributed Computing*, 15(2) :67–86, 2002.

[2] J. Aspnes. Slightly smaller splitter networks. *CoRR*, abs/1011.3170, 2010.

[3] H. Attiya, A. Bar-Noy, and D. Dolev. Sharing memory robustly in message-passing systems. *J. ACM*, 42(1) :124–142, 1995.

[4] H. Attiya, A. Bar-Noy, D. Dolev, D. Peleg, and R. Reischuk. Renaming in an asynchronous environment. *J. ACM*, 37(3) :524–548, July 1990.

[5] D. Bonnin and C. Travers. Splitting and renaming with a majority of faulty processes. In *Proceedings of the 2015 International Conference on Distributed Computing and Networking*, ICDCN '15, pages 26 :1–26 :10. ACM, 2015.

[6] A. Castañeda, S. Rajsbaum, and M. Raynal. The renaming problem in shared memory systems : An introduction. *Computer Science Review*, 5(3) :229–251, 2011.

[7] S. Gilbert and N. A. Lynch. Perspectives on the CAP theorem. *IEEE Computer*, 45(2) :30–36, 2012.

[8] L. Lamport. A fast mutual exclusion algorithm. *ACM Trans. Comput. Syst.*, 5(1) :1–11, Jan. 1987.

[9] M. Moir and J. H. Anderson. Wait-free algorithms for fast, long-lived renaming. *Science of Computer Programming*, 25(1) :1 – 39, 1995.

[10] M. Raynal. *Communication and Agreement Abstractions for Fault-Tolerant Distributed Systems*. Synthesis Lectures on Distributed Computing Theory. Morgan & Claypool Publishers, 2010.