Election Robuste, Auto-Stabilisante et Efficace

C. Delporte-Gallet[1] S. Devismes[2] H. Fauconnier[1]

[1]*LIAFA, CNRS UMR 7089, Université Denis Diderot, Paris VII*
[2]*CNRS, LRI, Université de Paris-Sud (Paris XI), Orsay*

Ici, nous présentons un des résultats de notre article [DGDF07] : un algorithme à la fois *robuste*, *auto-stabilisant* et *efficace* pour le problème de l'*élection de leader* dans un réseau complet synchrone. À notre connaissance, aucun algorithme vérifiant ces trois propriétés n'avait jusqu'alors été proposé.

Keywords: Tolérance aux pannes, élection de leader, auto-stabilisation, algorithme robuste

1 Introduction

Les algorithmes robustes abordent la tolérance aux pannes selon une approche *pessimiste* : les processeurs suspectent toutes les informations qu'ils reçoivent et chaque étape de calcul est garantie par un nombre suffisant de contrôles préalables. Cette approche est généralement utilisée dans des réseaux où les pannes sont *définitives*.

Les algorithmes auto-stabilisants [Dij74] abordent la tolérance aux pannes selon une approche *optimiste* : cette méthode autorise les processeurs non-défaillants à avoir un comportement incorrect mais garantit le retour du système à un comportement normal en un temps fini après que les pannes ont cessé. L'auto-stabilisation est donc adaptée aux systèmes pouvant subir des pannes *transitoires* (*i.e.*, une panne transitoire est une défaillance temporaire d'un des composants du réseau).

Travaux précédents. Quelques articles [GP93, AH93, BKM97] traitent d'une approche transversale : la *ftss* (*fault-tolerance and Self-Stabilisation*). Un algorithme *ftss* est auto-stabilisant dans un système où des pannes transitoires et/ou définitives sont possibles. Gopal et Perry [GP93] ont proposé un algorithme *ftss* de synchronisation de phases fonctionnant dans un réseau complet synchrone où au plus k processeurs peuvent *crasher*[†]. En utilisant cet algorithme, ils ont aussi proposé une méthode pour transformer des algorithmes robustes en algorithmes *ftss*. Anagnostou et Hadzilacos [AH93] démontrent qu'il n'y a pas d'algorithme *ftss* permettant de calculer la taille d'un réseau en anneau si le réseau est asynchrone et peut subir un crash au plus. Beauquier et Kekonnen-Moneta [BKM97] étendent ce résultat d'impossibilité mais démontrent aussi que dans un réseau synchrone, il devient possible de résoudre ce problème avec des détecteurs de pannes.

La plupart des solutions auto-stabilisantes ou *ftss* existantes utilisent le *heartbeat* : chaque processeur envoie régulièrement son état local à ses voisins, même lorsque le système est stabilisé. Cette méthode est coûteuse en terme de communications : dans un réseau complet de n processeurs, $\Theta(n^2)$ canaux unidirectionnels sont utilisés en permanence. L'*efficacité en communication* [LFA00] a été définie pour mettre en avant le fait qu'un algorithme *robuste* utilise un minimum de canaux de communication : un algorithme est dit *efficace en communication* s'il finit par ne plus utiliser que $n-1$ canaux de communication — cette borne étant prouvée optimale pour les algorithmes robustes.

Contributions. Dans cet article, nous proposons une solution à la fois robuste, auto-stabilisante et efficace (en communication) au problème de l'élection de leader dans un réseau complet synchrone où un nombre fini mais quelconque de pannes (pannes transitoires ou pannes définitives de processeurs) peuvent arriver. Bien que l'élection de leader ait été beaucoup étudiée dans le domaine de la tolérance aux pannes, aucun algorithme vérifiant ces trois propriétés à la fois n'a été proposé jusqu'à présent.

[†] Dans cet article, nous utiliserons l'anglicisme *crashé* pour signifier qu'un processeur est définitivement en panne.

Plan. Dans la section 2, nous présentons le modèle que nous utilisons. Dans la section 3, nous proposons notre algorithme. Par manque de place, la preuve formelle de l'algorithme n'est pas présentée dans cet article. En revanche, elle est disponible dans le rapport interne en ligne à l'adresse suivante : `http://hal.archives-ouvertes.fr/hal-00167935/fr/`. Nous concluons dans la section 4.

2 Modèle

Nous considérons des réseaux *complets* ayant un nombre *fini* de processeurs : pour chaque paire de processeurs distincts (p,q), il existe un canal de communication de p vers q et un autre canal de q vers p. Les processeurs communiquent entre eux par envoi et réception de messages. Nous ne faisons pas d'hypothèse sur l'ordre de délivrance des messages.

L'analyse d'un algorithme auto-stabilisant commence à partir de la configuration immédiatement postérieure à la fin de la dernière faute, cette configuration étant considérée comme la configuration initiale du système. Pendant la période où les fautes se produisent, nous supposons que :

- Les états locaux des processeurs peuvent être corrompus.
- Les processeurs peuvent tomber définitivement en panne.
- Les messages peuvent être corrompus, perdus et/ou retardés.

La configuration initiale du système peut donc être complètement quelconque[‡] : l'état local de chaque processeur peut être quelconque, les canaux de communication peuvent contenir un nombre fini de messages quelconques et des processeurs peuvent être en panne définitivement. Cependant, à partir de cette configuration initiale quelconque, nous supposons que les processeurs vivants[§] suivent leur algorithme local, que plus aucune panne ne survient[¶] et que le réseau est complètement synchrone, *i.e.*, les processeurs et les canaux de communication sont synchrones.

Le fait que les processeurs soient *synchrones* signifie que le temps d'exécution de chacun de leurs pas de calcul est borné. L'algorithme que nous proposons (l'algorithme 1) consiste en une boucle infinie contenant un nombre fini d'actions. Le temps d'exécution d'un tour de boucle est donc lui aussi borné. Pour simplifier, nous supposerons que chaque processeur exécute un tour de boucle en une unité de temps.

Le fait que les canaux de communication soient *synchrones* signifie que chaque message en transit dans un canal est délivré en un temps borné. Pour simplifier, nous supposerons que la borne sur le temps de délivrance des messages est une constante entière non nulle notée δ.

3 L' algorithme

Dans cette section, nous présentons un algorithme robuste, auto-stabilisant et efficace d'élection de leader. Notre algorithme — l'algorithme 1 — garantit qu'à partir d'une configuration initiale quelconque, le système atteint en un temps fini une configuration γ où :

(i) Chaque processeur vivant dans γ vérifie $\texttt{Elu} = \ell$ tel que ℓ est un processeur vivant.

(ii) Chaque processeur vivant vérifie $\texttt{Elu} = \ell$ dans toutes les configurations atteignables depuis γ.

Bien sûr, la variable `Elu` désigne le leader du réseau. Enfin, les configurations vérifiant les propriétés (i) et (ii) sont appelées les configurations *légitimes*. Inversement, les configurations ne vérifiant pas les propriétés (i) ou (ii) sont appelées configurations *illégitimes*.

Notre solution est basée sur quatre principes : le premier principe assure l'efficacité en communication, les trois autres la stabilisation du système.

Pour obtenir l'efficacité en communication, notre algorithme utilise un seul type de message et procède comme suit :

[‡] Comme tout algorithme auto-stabilisant retrouve en un temps fini un comportement correct à partir d'une configuration initiale quelconque, un tel algorithme ne nécessite pas de phase d'initialisation des variables.

[§] *i.e.*, les processeurs qui ne sont pas tombés en panne définitivement.

[¶] En fait, cela signifie que le protocole assure la convergence du système dès que le temps entre deux périodes de pannes est suffisamment long.

(1) Un processeur p envoie périodiquement des messages VIVANT contenant sa propre identité aux autres processeurs seulement si $\texttt{Elu}_p = p$, *i.e.*, seulement si p pense être l'élu (*i.e.*, le leader). Afin d'éviter de surcharger le réseau, un processeur pensant être le leader déclenche un envoi de messages VIVANT toutes les $k\delta$ unités de temps (avec $k \in \mathbb{N}^*$). Ce temps est nécessaire pour assurer que les messages VIVANT de l'envoi précédent ont été reçus.

En utilisant ce mécanisme, nous obtenons facilement l'efficacité en communication. En effet, lorsque le système a atteint une configuration légitime, il existe un seul processeur p vérifiant $\texttt{Elu}_p = p$. Ce processeur (le leader) est alors le seul à envoyer des messages. Il faut noter qu'il est nécessaire que le leader envoie périodiquement des messages VIVANT aux autres processeurs afin qu'il ne soit pas suspecté d'être crashé.

Algorithm 1 Code pour chaque processeur p

```
1: Variables : Elu_p : entier naturel ; TimerEnv_p ∈ {0...kδ} ; TimerRecep_p ∈ {0...8kδ}
2: Répéter Toujours
3:     Pour tout q ∈ V \ {p} faire
4:         Si Reçoit(VIVANT,q) alors
5:             Si (Elu_p ≠ p) ∨ (q < p) alors        /* Principes 2 et 3 */
6:                 Elu_p ← q
7:             Fin Si
8:             TimerRecep_p ← 0
9:         Fin Si
10:    Fin Pour
11:    TimerEnv_p ← TimerEnv_p + 1
12:    Si TimerEnv_p ≥ kδ alors
13:        Si Elu_p = p alors        /* Principe 1 */
14:            Envoie(VIVANT,p) à tous les autres processeurs
15:        Fin Si
16:        TimerEnv_p ← 0
17:    Fin Si
18:    TimerRecep_p ← TimerRecep_p + 1
19:    Si TimerRecep_p > 8kδ alors        /* Principe 4 */
20:        Si Elu_p ≠ p alors
21:            Elu_p ← p
22:        Fin Si
23:        TimerRecep_p ← 0
24:    Fin Si
25: Fin Répéter
```

Notre algorithme utilise trois autres principes pour stabiliser à partir d'une configuration initiale quelconque. La figure 1 présente trois configurations initiales possibles qui illustrent les problèmes à résoudre pour obtenir une élection de leader auto-stabilisante. L'état des variables Elu étant quelconque dans la configuration initiale, plusieurs processeurs — comme les processeurs 1 et 4 dans la configuration (a) — peuvent penser être le leader. Pour converger vers une configuration où un seul leader existe nous utilisons le principe suivant :

(2) Lorsqu'un processeur p tel que $\texttt{Elu}_p = p$ reçoit un message (VIVANT,q), le processeur q devient l'élu de p si $q < p$.

Ensuite, certaines variables Elu peuvent contenir des identités de processeurs qui n'existent pas ou plus dans le réseau. Par exemple, dans la configuration (b), les variables $\texttt{Elu}_2$ et $\texttt{Elu}_3$ contiennent les valeurs 5 et 4, or il n'y a pas de processeur 5 dans le réseau et le processeur 4 est crashé. Afin que tout les processeurs adoptent un leader valide, nous appliquons le principe suivant :

(3) Lorsqu'un processeur p tel que $\texttt{Elu}_p \neq p$ reçoit un message (VIVANT,q), le processeur q devient l'élu de p.

Enfin, dans une configuration initiale quelconque, il est possible qu'aucun des processeurs vivants ne se considère comme un leader (*cf.* configuration (c)). Pour palier à ce problème, nous appliquons le principe suivant :

(4) Si un processeur p ne reçoit pas de message VIVANT pendant une longue période, alors p suspecte son leader (*i.e.*, $\texttt{Elu}_p$) et devient un leader potentiel en affectant $\texttt{Elu}_p$ à p. Cette période d'attente doit être suffisamment longue pour que p soit sûr qu'il n'y a aucun leader dans le réseau. Dans [DGDF07], nous avons prouvé que cette période doit être d'au moins $8k\delta$ unités de temps.

Les principes (1) et (4) nécessitent l'utilisation de *timer*. Pour mettre en œuvre ces deux *timers*, nous utilisons simplement deux compteurs locaux `TimerEnv` et `TimerRecep` qui sont incrémentés à chaque tour de boucle. Chaque tour de boucle s'effectuant en 1 unité de temps, un processeur peut déduire facilement le temps écoulé à partir de la valeur de ses compteurs. Bien sûr, dans la configuration initiale, les compteurs peuvent avoir des valeurs quelconques et, par conséquent, le premier déclenchement du timer peut arriver trop tôt. Cependant, chaque compteur étant réinitialisé à 0 après chaque déclenchement, les déclenchements suivants se feront au moment prévu.

Il faut noter que pour simplifier l'algorithme, nous avons supposé que les processeurs connaissaient le temps d'exécution d'un tour de boucle et le temps maximal de délivrance d'un message. Cette hypothèse peut être levée en utilisant la méthode présentée dans [ADGFT03]. Cependant, cette méthode nécessite l'utilisation de compteurs non-bornés.

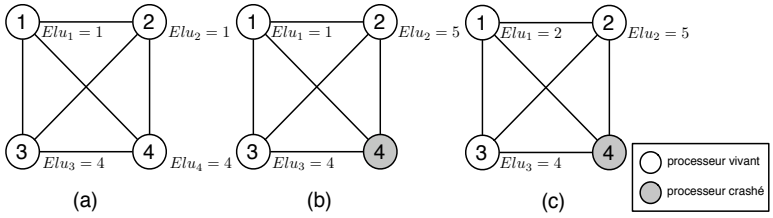

FIG. 1: Configurations initiales possibles.

4 Conclusion

Dans cet article, nous avons présenté le premier algorithme d'élection à la fois robuste, auto-stabilisant et efficace en communication. L'intérêt de l'approche robuste auto-stabilisante (*ftss*) est qu'elle permet d'obtenir des protocoles à la fois tolérants aux pannes définitives et transitoires. L'efficacité en communication est aussi essentielle. En effet, elle met en avant le fait qu'un algorithme est économe en envoi de messages. L'efficacité en communication n'est (pour le moment) que trop rarement prise en compte dans la conception des algorithmes auto-stabilisants.

Références

[ADGFT03] Marcos Kawazoe Aguilera, Carole Delporte-Gallet, Hugues Fauconnier, and Sam Toueg. On implementing omega with weak reliability and synchrony assumptions. In *PODC*, pages 306–314, 2003.

[AH93] Efthymios Anagnostou and Vassos Hadzilacos. Tolerating transient and permanent failures (extended abstract). In *WDAG*, pages 174–188, 1993.

[BKM97] Joffroy Beauquier and Synnöve Kekkonen-Moneta. Fault-tolerance and self-stabilization : impossibility results and solutions using self-stabilizing failure detectors. *Int. J. Systems Science*, 28(11) :1177–1187, 1997.

[CHT96] Tushar Deepak Chandra, Vassos Hadzilacos, and Sam Toueg. The weakest failure detector for solving consensus. *J. ACM*, 43(4) :685–722, 1996.

[DGDF07] Carole Delporte-Gallet, Stéphane Devismes, and Hugues Fauconnier. Robust stabilizing leader election. In *SSS*, pages 219–233, 2007.

[Dij74] EW Dijkstra. Self stabilizing systems in spite of distributed control. *Communications of the ACM*, 17 :643–644, 1974.

[GP93] Ajei S. Gopal and Kenneth J. Perry. Unifying self-stabilization and fault-tolerance (preliminary version). In *PODC*, pages 195–206, 1993.

[LFA00] Mikel Larrea, Antonio Fernández, and Sergio Arévalo. Optimal implementation of the weakest failure detector for solving consensus. In *SRDS*, pages 52–59, 2000.