Nous aussi on fait du « $\log^*$ » ![†]

Stéphane Devismes[1], David Ilcinkas[2], Colette Johnen[2] et Frédéric Mazoit[2]

[1] *Université de Picardie Jules Verne, MIS, UR 4290, Amiens, France*
[2] *Univ. Bordeaux, CNRS, Bordeaux INP, LaBRI, UMR 5800, F-33400 Talence, France*

L'autostabilisation est une propriété garantissant le retour à un comportement correct d'un système distribué en temps fini après des fautes transitoires. Le temps maximum pour obtenir cette convergence est appelé temps de stabilisation. Ce dernier est généralement mesuré en rondes. Il est notoirement connu qu'obtenir un temps de stabilisation en rondes linéaire voire sous-linéaire en le diamètre D du réseau est difficile, qui plus est lorsque l'on souhaite contenir la charge de travail des processus, mesurée en nombre de mouvements, afin que celle-ci reste polynomial en le nombre n de processus. Dans cet article, nous montrons pourtant que ces complexités peuvent être atteintes, en transformant tout algorithme synchrone non-autostabilisant qui termine en algorithme asynchrone, autostabilisant, silencieux et efficace à la fois en rondes et en nombre de mouvements. En particulier, la borne de $O(\log^* n)$ rondes est obtenue par notre version autostabilisante de l'algorithme de coloriage d'un anneau par Cole et Vishkin [2].

Mots-clefs : Autostabilisation, transformateur, algorithmes totalement polynomiaux

1 Introduction

Après un nombre fini de *fautes transitoires* [‡], la configuration d'un système distribué peut être quelconque et ainsi violer les propriétés de sûreté du système. À partir d'une telle configuration, un algorithme *autostabilisant* permet au système de retrouver en temps fini une configuration dite *légitime* à partir de laquelle sa spécification est satisfaite [5]. L'autostabilisation est ainsi considérée comme une approche générale pour tolérer de telles fautes.

Le *temps de stabilisation* est la durée maximale de la période après les pannes transitoires qui dure jusqu'à ce que les propriétés de sûreté du système soit à nouveau satisfaites. Elle est généralement évaluée en nombre de *rondes*. Ces dernières capturent le temps d'exécution en fonction de la vitesse des processus les plus lents. Bien évidemment, il est souhaitable de minimiser le temps de stabilisation d'un algorithme autostabilisant. Or, puisqu'un calcul distribué peut nécessiter qu'un processus acquiert des informations stockées dans la mémoire de processus éloignés, le diamètre D du réseau est souvent une borne inférieure sur le temps de stabilisation en rondes. Ainsi, la recherche de solutions ayant des complexités en rondes polynomiales, voire linéaires en D, est naturelle.

Nous considérons ici *le modèle à états*, une abstraction du modèle à passage de messages dans laquelle les échanges d'informations sont réalisés via des *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. Dans ce modèle, l'exécution d'un algorithme est composée d'une succession de pas de calcul atomiques. 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. Ensuite, si au moins un processus est activable, un adversaire, le *démon*, choisit un sous-ensemble non vide de processus activables. En un pas atomique, les processus choisis sont activés et mettent à jour leurs variables. Nous considérons ici le démon le plus général : le démon distribué inéquitable, qui active les processus de manière asynchrone sans aucune hypothèse particulière sur l'équité.

Dans le modèle à états, une autre unité de mesure permet d'évaluer la charge de travail des processus : le nombre de mouvements, c'est-à-dire, le nombre de changements d'état. Minimiser le nombre de mouvements permet à un algorithme autostabilisant d'effectuer moins d'opérations de communication. En effet, comme expliqué dans [6], pour implémenter une solution du modèle à états en pratique (c'est-à-dire, en passage

[†] Ce travail a été partiellement financé par les projets ANR SkyData (ANR-22-CE25-0008) et ANR TEMPOGRAL (ANR-22-CE48-0001).

‡. Des perturbations temporaires et rares du fonctionnement de certains des composants du système (liens ou processus).

de messages), les processus doivent vérifier régulièrement s'ils doivent changer d'état en raison de la modification de l'état de certain voisins. Ainsi, chaque processus doit envoyer régulièrement des informations à ses voisins et stocker le dernier état connu de chacun de ses voisins. Cependant, au lieu d'envoyer régulièrement des messages potentiellement lourds contenant l'état local complet du processus, on peut adopter l'approche légère proposée dans [6] : les processus envoient régulièrement à leurs voisins une preuve de leur état ; ensuite, un processus demande l'état local complet d'un voisin seulement après qu'une preuve reçue montre une modification d'état. Une telle preuve peut être très petite par rapport à l'état local : le processus peut simplement générer aléatoirement un entier x, saler un hachage de son état local avec cet entier, et enfin envoyer un « petit » message contenant le hachage h et l'entier x. Le processus récepteur fait la vérification en comparant h au hachage salé avec x du dernier état (complet) reçu du processus émetteur. De cette façon, le nombre de messages « lourds » (c'est-à-dire, ceux contenant des états complets) dépend du nombre de mouvements, et la minimisation du nombre de mouvements permet de réduire significativement le volume de données échangées. Cela est particulièrement vrai lorsque le système autostabilisant est silencieux [6], c'est-à-dire, lorsqu'il finit par atteindre une configuration terminale où les états des processus sont fixes. En effet, dans ce cas, il n'y a plus de mouvements à partir de la configuration terminale, et à partir de là, seuls des « petits » messages sont envoyées.

Cournier *et al.* [3] abordent l'existence d'algorithmes autostabilisants non-triviaux, appelés *totalement polynomiaux*, dont le temps de stabilisation en rondes et en nombre de mouvements sont respectivement bornés par des polynômes en D et n. Ils démontrent l'existence de tels algorithmes autostabilisants totalement polynomiaux en proposant le seul algorithme autostabilisant totalement polynomial à ce jour : un algorithme silencieux qui calcule un arbre couvrant en largeur dans un réseau connexe enraciné qui stabilise en $O(D^2)$ rondes et $O(n^6)$ mouvements. Dans cet article, ils énoncent également plusieurs questions ouvertes, parmi lesquelles : (1) Existe-t-il un algorithme autostabilisant asynchrone totalement polynomial pour le calcul de l'arbre couvrant en largeur qui stabilise en $O(D)$ rondes ? (2) D'autres problèmes fondamentaux, tels que l'élection de leader, admettent-il des solutions autostabilisantes totalement polynomiales ?

Contribution. Cet article est un résumé de notre rapport technique disponible sur HAL (`https://hal.science/hal-04159863`) dans lequel nous répondons notamment aux deux questions mentionnées ci-dessus par l'affirmative. En effet, nos résultats montrent que tout algorithme synchrone qui termine en $O(D)$ rondes peut être transformé en un algorithme autostabilisant, silencieux et asynchrone qui stabilise en $O(D)$ rondes et $O(n^3)$ mouvements. Or, il est bien connu que des problèmes classiques tels que la construction d'arbres couvrants et l'élection de leader admettent de telles solutions synchrones (voir la section 3 pour un exemple et notre rapport technique pour plus de détails).

En fait, nous proposons un schéma d'algorithme général, appelé *transformateur*, qui simule dans le modèle à états, sous l'hypothèse d'un démon distribué inéquitable, tout algorithme synchrone $AlgI$ qui termine. En plus de $AlgI$, il nécessite les entrées suivantes : (1) une borne supérieure B sur le temps d'exécution de $AlgI$, qui peut être fixée à $+\infty$ si elle est inconnue et (2) un paramètre booléen indiquant si le transformateur fonctionne en mode « greedy » ou « lazy » : en mode « greedy », notre transformateur simule systématiquement T rondes de $AlgI$, alors qu'en mode « lazy », il simule la $i^{\text{ème}}$ ronde de $AlgI$ seulement si c'est nécessaire (par exemple, si $AlgI$ termine en au moins i rondes). Nous notons $Trans(AlgI)$ l'algorithme simulant $AlgI$ obtenu avec notre transformateur. Les différentes bornes de complexité de $Trans(AlgI)$, en fonction du mode utilisé, sont synthétisées dans la table 1 ci-dessous.

	Mouvements	Rondes
mode Lazy	$O(\min(n^3+nT, n^2B))$	$O(D+T)$
mode Greedy	$O(\min(n^3+nB, n^2B))$	$O(B)$
Complexité en espace	$O(B \cdot S)$	

— T et S sont respectivement les complexités en temps et en espace de $AlgI$.
— $B \in [T, +\infty)$.

TABLE 1 : Complexités de $Trans(AlgI)$.

Notez que notre méthode est compatible avec la plupart des variantes du modèle à états. En particulier, notre transformateur n'utilise ni les identifiants des processus ni les numéros de canaux. Ainsi, il peut être utilisé dans des modèles forts avec des identifiants comme le modèle *LOCAL* [8], mais aussi dans des modèles faibles tels que le modèle *Stone Age* [7]. Enfin, nos résultats ont les trois implications suivantes :

1. En mode « lazy », $Trans(AlgI)$ est totalement polynomial si $AlgI$ termine en un nombre de rondes synchrones polynomial en D, indépendamment de la valeur de B. Par conséquent, tout problème qui peut être résolu dans le modèle *LOCAL* admet une solution asynchrone, autostabilisante (silencieuse) et totalement polynomiale.
2. Tout problème dont la complexité optimale est en $\Omega(D)$ rondes sous l'hypothèse synchrone et qui admet une solution qui termine, admet également une solution autostabilisante (silencieuse) asynchrone avec la même complexité en rondes et une complexité en $O(n^3)$ mouvements. Par conséquent, nous obtenons des solutions autostabilisantes asynchrones totalement polynomiales pour de nombreux problèmes fondamentaux, tels que des constructions d'arbre couvrant et l'élection de leader.
3. Si la borne B fournie est linéaire en T, notre transformateur en mode « greedy » réalise une simulation en $O(T)$ rondes, même si T est sous-linéaire en D. Ainsi, nous obtenons des solutions autostabilisantes extrêmement rapides. Par exemple, la borne de $O(\log^* n)$ rondes est atteinte par notre version autostabilisante de l'algorithme de coloriage d'un anneau initialement proposé par Cole et Vishkin [2].

Plan. Dans la section suivante, nous présentons succinctement les idées principales derrière notre transformateur. Dans la section 3, nous montrons que notre transformateur est un outil puissant de simplification en décrivant une entrée possible de notre transformateur qui permet d'obtenir un algorithme autostabilisant silencieux d'élection de leader totalement polynomial.

2 Notre transformateur

Concevoir des algorithmes autostabilisants asynchrones est complexe en raison du grand nombre d'exécutions possibles. Ceci est exacerbé par le fait que nous supposons un démon inéquitable, pour lequel un processus peut être bloqué par le démon tant qu'un autre peut être activé. Vouloir que ces algorithmes soient de plus à la fois rapides en rondes et économes en mouvements est donc encore plus exigeant. En effet, pour que l'algorithme soit rapide, les processus doivent exploiter le parallélisme autant que possible et ainsi effectuer leurs calculs le plus tôt possible. Mais dans ce cas, un processus p très rapide peut faire beaucoup de calculs qui s'avéreront inutiles car p n'a pas reçu une information cruciale venant d'un processus lent. Ainsi, les algorithmes rapides ont tendance à être médiocre en nombre de mouvements ; voir par exemple [4]. Une idée naturelle est alors de ralentir les processus en supprimant en partie le parallélisme pour les empêcher de faire des calculs lorsque les informations disponibles ne sont pas certaines. Mais, cela peut conduire à un ralentissement de l'algorithme. Ainsi, il faut garder assez de parallélisme pour être rapide en rondes tout en attendant les processus lents pour être efficace. Réaliser cet équilibre entre ces deux objectifs est délicat.

Une approche classique consiste à utiliser un mécanisme de correction d'erreur dans lequel les processus qui détectent des « erreurs majeures » (états localement incohérents) lancent une réinitialisation des processus possiblement influencés par ces erreurs. Une telle réinitialisation partielle peut être effectuée de plusieurs façons. Si cette réinitialisation permet aux processus de reprendre leurs calculs trop tôt, le problème des mouvements superflues persiste. Une manière courante d'effectuer une réinitialisation « lente » et précise est d'utiliser au moins deux phases : une phase de diffusion dans laquelle les processus impliqués sont gelés, et une phase de retour dans laquelle les processus impliqués se réinitialisent et confirment la bonne propagation du gel. Le calcul reprend ensuite après cette seconde phase. Ces deux phases organisent les processus qui se réinitialisent en graphes orientés acycliques (*DAG*), voire en arbres. Puisque le gel se propage le long de tels DAG, la complexité en rondes de la phase de retour dépend de leur profondeur, qui peut être $\Omega(n)$. On peut alors vouloir permettre aux processus de restructurer les DAG d'erreur pour obtenir une profondeur finalement en $O(D)$. Mais ce mécanisme de raccourcissement peut être lui-même coûteux. Ainsi, cela suggère une fois de plus qu'être à la fois rapide en rondes et économe en mouvements semble intrinsèquement contradictoire.

Notre algorithme est une variante du *Rollback Compiler* d'Awerbuch et Varghese [1]. Le principe de ce dernier est de stocker toute l'exécution synchrone de $AlgI$. Chaque processus p possède donc une liste L telle que pour toute cellule i, on a ultimement $p.L[i] = \mathtt{st}_p^i$, où $\mathtt{st}_p^i$ est l'état de p à la ronde i de l'exécution synchrone de $AlgI$ ($p.L[0] = \mathtt{st}_p^0$ étant l'état initial de p dans $AlgI$). Puisque la cellule $p.L[i+1]$ est calculée à partir des cellules $q.L[i]$ des processus q du voisinage fermé de p (c'est-à-dire, p et ses voisins), on dit que $p.L[i+1]$ *dépend* des cellules $q.L[i]$. Lorsque le processus p est activé, si certaines des cellules de sa liste sont incorrectes, il les corrige toutes. De plus, si $p.L[i]$ n'existe pas mais est nécessaire (*e.g.*,

l'exécution synchrone de $AlgI$ contient au moins i rondes) et que toutes ses dépendances existent, alors il crée $p.L[i]$. Cela donne un algorithme simple qui a une bonne complexité en rondes mais qui a une complexité exponentielle en mouvements (un exemple d'exécution exponentielle en mouvements est donné dans notre rapport technique).

Pour être efficace en mouvements, nous utilisons un mécanisme de correction d'erreur qui parvient à la fois à geler les processus impliqués et à raccourcir les DAG d'erreur. Chaque fois qu'un processus p détecte une erreur majeure, il efface entièrement sa liste $p.L$ et passe en état d'erreur. Cela crée généralement des dépendances manquantes dans les cellules voisines. Les processus avec de telles cellules les suppriment ensuite (et uniquement celles-ci), et passent également en état d'erreur. Cela crée des DAG d'erreur qui sont raccourcis chaque fois qu'un processus supprime davantage de cellules à cause d'un autre voisin, conduisant ainsi l'algorithme à être efficace en rondes. En supprimant uniquement les cellules ayant des dépendances manquantes, on obtient des tailles de liste strictement décroissantes le long de chaque chemin du DAG partant d'une feuille jusqu'à une racine. Une fois que le DAG a atteint sa taille maximale, ses nœuds repassent en état normal à partir des feuilles. Cependant, grâce aux tailles de liste décroissantes, ils ne reprennent la simulation d'$AlgI$ qu'après leurs prédécesseurs dans le DAG. Ainsi, les nœuds impliqués dans un DAG d'erreur ne reprennent la simulation qu'après la complète réinitialisation de celui-ci.

Un nouveau « segment » démarre chaque fois qu'un processus ayant détecté une erreur majeure termine sa réinitialisation. Notre mécanisme de correction garantit qu'aucune erreur majeure ne peut être créée. Il y a donc au plus $n+1$ segments. Or, nous avons vu qu'un processus impliqué dans des DAG d'erreur ne peut pas appliquer d'étape de simulation avant la complète réinitialisation de ces derniers et donc avant la fin du segment en cours. L'efficacité de notre algorithme en nombre de mouvements se déduit de cette propriété.

3 Un exemple d'instance

Pour illustrer la puissance de notre méthode, considérons le problème de l'élection de leader dans un réseau identifié connexe. Nous devons simplement définir un algorithme non-autostabilisant, donc avec une initialisation bien choisie, qui termine sous l'hypothèse synchrone.

Pour chaque processus p, l'algorithme consiste simplement à calculer dans une variable $p.Best$ l'identifiant minimum du réseau. Tout d'abord, chaque processus p initialise $p.Best$ avec sa propre identité. Puis, à chaque ronde synchrone, p évalue la valeur minimale parmi les variables $Best$ dans son voisinage fermé et met à jour $p.Best$ si nécessaire. Ainsi, après chaque ronde, chaque processus connaît l'identifiant minimum des processus un saut plus loin. Après au plus D rondes, le calcul se termine et la variable $Best$ de chaque processus contient pour toujours l'identifiant minimum du réseau.

En utilisant notre transformateur en mode « lazy », nous obtenons un algorithme d'élection de leader autostabilisant silencieux totalement polynomial qui stabilise en $O(D)$ rondes et $O(n^3)$ mouvements. De plus, en donnant une borne supérieure B sur D en entrée du transformateur, on obtient une solution à mémoire bornée réalisant des complexités temporelles équivalentes. Ainsi, si on fait l'hypothèse habituelle selon laquelle les identifiants sont stockés dans $O(\log\ n)$ bits, nous obtenons un encombrement mémoire en $O(B \cdot \log\ n)$ bits par processus. À notre connaissance, notre solution est la première solution asynchrone autostabilisante totalement polynomiale pour l'élection de leader.

Références

[1] B. Awerbuch and G. Varghese. Distributed program checking : a paradigm for building self-stabilizing distributed protocols. In *FOCS'91*.

[2] R. Cole and U. Vishkin. Approximate and exact parallel scheduling with applications to list, tree and graph problems. In *FOCS'86*.

[3] A. Cournier, S. Rovedakis, and V. Villain. The first fully polynomial stabilizing algorithm for bfs tree construction. *Information and Computation*, 2019.

[4] S. Devismes and C. Johnen. Silent self-stabilizing BFS tree algorithms revisited. *Journal on Parallel Distributed Computing*, 2016.

[5] E. W. Dijkstra. Self-stabilization in spite of distributed control. *Comm. of the ACM*, 1974.

[6] S. Dolev, M. G. Gouda, and M. Schneider. Memory requirements for silent stabilization. *Acta Informatica*, 1999.

[7] Y. Emek and R. Wattenhofer. Stone age distributed computing. In *PODC'13*.

Nous aussi on fait du "$\log^*$*" !*

[8] N. Linial. Locality in distributed graph algorithms. *SIAM Journal on Computing*, 1992.