Temps, travail, espace : pourquoi choisir ? Un unisson autostabilisant et ses retombées

Stéphane Devismes, David Ilcinkas, Colette Johnen, Frédéric Mazoit

Temps, travail, espace : pourquoi choisir ? Un unisson autostabilisant et ses retombées[†]

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*

Nous présentons un algorithme autostabilisant pour le problème de l'unisson asynchrone qui fonctionne dans un modèle faible et qui est à la fois efficace en temps, en charge de travail et en espace.

Plus précisément, notre algorithme est défini dans le modèle à états et fonctionne dans des réseaux connectés asynchrones anonymes dans lesquels même les ports locaux ne sont pas étiquetés. Aucune hypothèse sur le démon n'est nécessaire. Ainsi, notre algorithme stabilise avec l'ordonnanceur le plus faible : le démon distribué inéquitable.

Dans un réseau de n nœuds, de diamètre D et en supposant la connaissance d'une borne $B \geq 2D + 2$, notre algorithme ne nécessite que $O(\log(B))$ bits par nœud et est *totalement polynomial* puisqu'il stabilise en au plus $2D + 2$ rondes et $O(\min(n^2B, n^3))$ mouvements. En particulier, il s'agit du premier unisson asynchrone autostabilisant pour des réseaux anonymes quelconques dont le temps de stabilisation en rondes est asymptotiquement optimal tout en utilisant une mémoire bornée à chaque nœud.

En outre, nous montrons que notre solution peut être utilisée pour simuler efficacement des algorithmes synchrones autostabilisants dans des environnements asynchrones. Par exemple, cette simulation nous permet de concevoir un nouvel algorithme autostabilisant silencieux qui résout à la fois l'élection de leader et la construction d'un arbre couvrant en largeur dans n'importe quel réseau connecté identifié et qui est, à notre connaissance, meilleur que toutes les solutions existantes de la littérature.

Mots-clefs : Autostabilisation, unisson, synchroniseur, algorithmes totalement polynomiaux

1 Introduction

Contexte. Dans un système réparti, les entités de calcul, ici appelés nœuds, sont autonomes et ont une vue partielle de la configuration globale du système, mais peuvent communiquer entre elles via un réseau d'interconnexion. Nous considérons des systèmes répartis pouvant faire l'objet de pannes transitoires [‡], et donc des solutions algorithmiques les prenant en compte. Plus précisément, après un nombre fini de telles fautes, 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* [11, 1] permet au système de retrouver en temps fini une configuration dite *légitime* à partir de laquelle sa spécification est satisfaite.

Dans le cas des systèmes asynchrones, avoir un outil qui permet de maintenir une synchronisation locale est très avantageux. L'*unisson* (asynchrone) [8] consiste à maintenir une horloge locale à chaque nœud. Le domaine de ces horloges est soit borné (comme les horloges de tous les jours), soit infini. La propriété de vivacité de ce problème stipule que chaque nœud doit incrémenter son horloge locale infiniment souvent. La propriété de sûreté impose que la différence entre les horloges de deux nœuds voisins dans le réseau soit toujours au plus d'un incrément.

Ses nombreuses applications font de l'unisson un outil fondamental en autostabilisation. En effet, il permet, entre autres, de simuler des systèmes synchrones dans un environnement asynchrone [9], d'affranchir un système asynchrone de son hypothèse d'équité [3], de faciliter la détection de terminaison [4], de partager localement des ressources [6], ou encore de calculer des infimums [5].

[†] Cet article est un résumé étendu d'un article présenté à STACS'2025 [10]. Il a été partiellement financé par les projets ANR SkyData (ANR-22-CE25-0008) et ANR ENEDISC (ANR-24-CE48-7768).

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

Contributions. Nous proposons un algorithme d'unisson asynchrone autostabilisant utilisant une mémoire bornée. Cet algorithme est défini dans une variante du modèle à états et fonctionne dans n'importe quel réseau anonyme connexe sous le démon distribué inéquitable (cf. Section 2). Il est *totalement polynomial* [7], c'est-à-dire que son temps de stabilisation en rondes est polynomial en le diamètre du réseau D, et son temps de stabilisation en mouvements est polynomial en le nombre de nœuds n. Précisément, il se stabilise en $2D+2$ rondes et $O(\min(n^2B, n^3))$ mouvements en utilisant $\lceil \log 2B \rceil + 1$ bits par nœud, où $B \geq 2D+2$ est une borne commune à l'ensemble des nœuds.

À notre connaissance, notre algorithme améliore significativement la littérature car les autres algorithmes d'unisson autostabilisants présentent au moins un des inconvénients suivants : utilisation d'une mémoire infinie (*e.g.*, [2]), une complexité en ronde en $\Omega(n)$ (*e.g.*, [6]), une condition forte sur le démon (synchrone ou équitable, *e.g.*, [12]). Par ailleurs, le modèle de calcul que nous utilisons est au moins aussi général que le modèle de « l'âge de pierre » d'Emek et Wattenhofer [13] : il ne nécessite aucun étiquetage des ports, ni la connaissance du degré d'un nœud.

En résumé, notre unisson est efficace en temps, charge de travail et espace, ce qui en fait également le premier algorithme autostabilisant totalement polynomial pour un problème dynamique. De plus, notre unisson a des retombées majeures pour de nombreux problèmes classiques du domaine, cf. Section 4.

2 Modèle

Nous considérons des réseaux bidirectionnels asynchrones que nous modélisons par des graphes connexes de n nœuds et de diamètre D où chaque nœud p peut communiquer directement avec le sous-ensemble $N(p)$ de ses *voisins*. Nous utilisons une variante du *modèle à états*. Dans ce modèle, chaque nœud détient un nombre fini de *variables localement partagées* qui définissent son *état*. Un nœud peut lire et modifier son état, il peut également accéder à l'ensemble contenant les états de ses voisins. La *configuration* du système est définie par l'état de chacun de ses nœuds.

Dans un *algorithme distribué*, chaque nœud applique continûment un même ensemble fini de règles de la forme $Etiquette : Garde \rightarrow Action$. L'étiquette identifie la règle, sa garde est un prédicat booléen portant sur l'état du nœud ainsi que l'ensemble contenant les états de ses voisins, et l'action est une suite d'affectations modifiant l'état du nœud.

Un nœud est *activable* s'il peut changer son état en exécutant la partie action d'une règle *activable*, i.e. d'une règle dont la garde est vraie. L'exécution d'un algorithme est composée d'une succession de *pas atomiques* de calcul. À chaque pas, un adversaire, le *démon*, choisit un sous-ensemble non vide de nœuds activables. Chacun de ces nœuds effectue un *mouvement* en exécutant atomiquement une de ses règles activables. S'il n'y a pas de nœud activable, l'exécution se termine et la dernière configuration est dite *terminale*. Un algorithme dont toutes les exécutions terminent est *silencieux*. Nous ne faisons ici aucune hypothèse sur le démon (le démon est *distribué inéquitable*). En particulier, s'il en a la possibilité, il peut « affamer » un nœud donné et ne jamais l'activer.

Le *temps de stabilisation* d'un algorithme autostabilisant est le temps maximal pour qu'une exécution atteigne une configuration légitime. Celui-ci est généralement exprimé avec deux types de mesures : le nombre de *mouvements* et de *rondes*. La première ronde d'une exécution termine dès lors que tous les nœuds qui étaient initialement activables ont soit exécuté une règle, soit été désactivés suite au mouvement d'un de leurs voisins. La seconde ronde commence à la fin de la première, *etc.* Notons que le nombre de mouvements est plus une mesure de travail qu'une mesure de temps.

3 L'algorithme

Soit $B \geq 2D+2$ un entier. Chaque nœud p a un *statut* $p.s \in \{C, E\}$ (Correct ou Erreur) et une *horloge* $p.c \in [-B, B[$ tels que si $p.s = E$, alors $p.c < 0$.[§] Nous définissons l'incrément $\oplus_B 1$ par $p.c \oplus_B 1 = p.c+1$ si $p.c \neq B-1$ et $p.c \oplus_B 1 = 0$ sinon. Deux horloges sont *synchronisées* si elles diffèrent d'au plus un incrément.

Bien qu'un nœud p ne puisse accéder qu'à l'ensemble contenant les états de ses voisins, nous écrivons des prédicats de la forme $\exists q \in N(p),\ \mathrm{Pred}(q.s, q.c)$ car leur sémantique est précisément $\exists (s, c) \in \{(q.s, q.c) \mid$

§. Cette contrainte peut facilement être implémentée avec un type union dans lequel le type de $p.c$ dépend de la valeur de $p.s$.

$q \in N(p)\}$, Pred(s, c). Nous faisons de même avec des prédicats universels.

Voici les prédicats que notre algorithme utilise :

$$\begin{aligned}
root(p) &:= \big(p.s = E \wedge \neg(\exists q \in N(p),\ q.s = E \wedge q.c < p.c)\big) \vee \\
&\qquad \big(p.s = C \wedge \exists q \in N(p),\ (q.c \geq p.c + 2) \wedge \neg(p.c = 0 \wedge q.c = B - 1)\big) \\
activeRoot(p) &:= root(p) \wedge (p.c \neq -B \vee p.s = C) \\
errorPropag(p, i) &:= i < 0 \wedge \exists q \in N(p),\ q.s = E \wedge q.c < i < p.c \\
canClearE(p) &:= p.s = E \wedge \forall q \in N(p),\ \big(|q.c - p.c| \leq 1 \wedge (q.c \leq p.c \vee q.s = C)\big) \\
updatable(p) &:= p.s = C \wedge \forall q \in N(p),\ q.c \in \{p.c, p.c \oplus_B 1\}
\end{aligned}$$

À partir de ces prédicats, les règles de notre unisson sont :

$$\begin{array}{ll|ll}
R_R : activeRoot(p) & \to (p.s, p.c) := (E, -B) & R_C : canClearE(p) & \to p.s := C \\
R_P(i) : errorPropag(p, i) & \to (p.s, p.c) := (E, i) & R_U : updatable(p) & \to p.c := p.c \oplus_B 1
\end{array}$$

La règle R_R a la plus haute priorité et $R_P(i)$ est plus prioritaire que $R_P(i')$ pour $i < i'$. Un nœud p tel que $root(p)$ est une *racine*. Il est en *erreur* si $p.s = E$ et est *correct* sinon. Deux voisins p et q dont les horloges sont désynchronisées forment une *falaise*. Si $p.c < q.c$, alors p en est son *pied* et q en est son *sommet*.

Une configuration est *légitime* si elle n'a pas de racine ou, ce qu'on prouve être équivalent, si tous les nœuds sont corrects et leurs horloges sont synchronisées. Comme les nœuds en erreur ont une horloge négative, on peut intuitivement se dire que pour que l'algorithme stabilise, il faut que toutes les horloges soient positives, mais c'est faux. Par exemple, la configuration dans laquelle tous les nœuds sont corrects et toutes les horloges valent $-B$, est légitime.

Intuition du fonctionnement de l'algorithme. Remarquons d'abord que dans une configuration légitime, seule la règle R_U est applicable et que les configurations légitimes sont stables par l'application de cette règle. Supposons maintenant que tous les nœuds sont corrects mais que la configuration contient une unique falaise dont r est le pied et p le sommet. Dans ces conditions, r est une racine et peut appliquer la règle R_R. Une fois cela fait, r est en erreur et son horloge vaut $-B$. Le nœud p peut alors appliquer la règle R_P pour se resynchroniser avec r : il devient en erreur et son horloge prend la valeur $-B + 1$. Ce faisant, il est possible que p crée une nouvelle falaise dont un autre nœud q est le sommet. Si c'est le cas, q peut à son tour appliquer la règle R_P pour se resynchroniser avec p, et ainsi de suite. Au final, le sommet r a lancé une diffusion d'erreur pour resynchroniser les horloges. Cette diffusion prend la forme d'un dag (directed acyclic graph) : tout nœud p en erreur a pour parent tout voisin q en erreur tel que $q.c < p.c$. Le dag d'erreurs peut être momentanément de profondeur $\Omega(n)$. Cependant, rien n'empêche un nœud d'un dag d'appliquer à nouveau la règle R_P s'il le peut. Ainsi, le dag se « raccourcit » et sa hauteur finit toujours par être au plus D.

Ensuite, pour retrouver une configuration légitime, il faut que les nœuds en erreur redeviennent corrects. Pour cela, on remarque qu'un nœud synchronisé en erreur n'ayant pas de descendant dans le dag peut appliquer la règle R_C. Donc après la diffusion d'erreur, il y a une phase de *retour* au terme de laquelle le nœud r applique la règle R_C. Il perd ainsi son statut de racine et la configuration devient légitime. Par construction, aucun nœud dont l'horloge est en avance sur un de ses voisins ne peut appliquer la règle R_U. Ainsi, parmi les nœuds ayant été impliqués dans le dag d'erreurs enraciné en r, le nœud r est le premier qui peut reprendre les incrémentations de son horloge, c'est-à-dire appliquer la règle R_U.

Évidemment, une situation initiale peut être plus complexe. Dans la configuration précédente, chaque nœud q en erreur à part le nœud r a au moins un parent. Nous considérons donc que, outre les pieds de falaise corrects, tout nœud en erreur sans parent est aussi une racine. De plus, on peut avoir plusieurs racines donc plusieurs diffusions simultanées. C'est pourquoi un nœud peut changer de dag ou appartenir à plusieurs dag d'erreurs.

Intuition des preuves. Nous prouvons qu'en ne permettant pas aux nœuds en erreur d'avoir des valeurs d'horloge positives, aucune racine n'apparaît. Nous découpons donc naturellement une exécution en au plus $n + 1$ *segments* à la fin desquels au moins une racine disparaît. Un nœud en erreur ne peut pas appliquer

la règle R_U tant que les racines de ses dags n'ont pas disparu, même s'il est devenu correct entre-temps. Comme les racines ne peuvent pas non plus appliquer cette règle, dans un segment ayant une racine, la synchronie requise par la règle R_U implique qu'un nœud quelconque ne peut appliquer cette règle que $2D$ fois au plus, après quoi il applique la règle R_P au plus $2B$ fois. Comme, entre deux applications de la règle R_C, un nœud applique une règle d'erreur (R_P ou R_R), l'algorithme converge en temps fini. Une analyse fine donne une convergence en $O(\min(n^3, Bn^2))$ mouvements.

Pour ce qui est des rondes, comme on ne peut pas créer de racine, toutes les applications de la règle R_R se font lors de la première ronde. À la fin de cette ronde, toutes les racines sont donc en erreur avec une horloge à $-B$. À la fin de la seconde ronde, tous les nœuds à distance 1 d'une racine ont pu appliquer la règle R_P et ont une horloge à $-B+1$. De proche en proche, à la fin de la ronde $D+1$, les falaises ont disparu et chaque nœud p en erreur a une horloge au plus $-B + d(p,r) \leq -B + D$ où r est une racine. Ensuite, parmi les nœuds en erreur, ceux dont l'horloge est maximale appliquent la règle R_C avant la fin de la ronde courante. Donc après chaque nouvelle ronde, la valeur maximale d'une horloge en erreur diminue. Toutes les racines disparaissent finalement et la configuration est légitime après au plus $2D+2$ rondes.

4 Conséquences

Comme indiqué dans l'introduction, une application majeure de l'unisson est de simuler un algorithme autostabilisant synchrone $\mathcal{A}$ dans un environnement asynchrone. Pour cela, nous utilisons une variante d'un algorithme du folklore. En plus des variables d'unisson, chaque nœud p stocke deux états de $\mathcal{A}$ dans les variables $p.pred$ et $p.curr$. Ces deux variables contiennent ultimement les deux derniers états de p dans une exécution synchrone de $\mathcal{A}$. Lorsque p incrémente son horloge, il fait un pas de simulation. Pour cela, il utilise la fonction $\widehat{\mathcal{A}}$ qui applique $\mathcal{A}$ dans le contexte où l'état de p est $p.curr$ et où l'état d'un voisin q de p est $q.curr$ si $q.c = p.c$ et $q.pred$ sinon (si $q.c = p.c \oplus_B 1$). Nous modifions aussi la règle R_U en :

$$R_U : updatable(p) \wedge P_{\text{paresseux}}(p) \longrightarrow (p.pred, p.curr) := \big(p.curr, \widehat{\mathcal{A}}(p)\big);\ p.c := p.c \oplus_B 1$$

dans lequel $P_{\text{paresseux}}(p) := (\widehat{\mathcal{A}}(p) \neq p.curr) \vee (\exists q \in N(p), q.c = p.c \oplus_B 1)$.

La garde de R_U implique que si $\mathcal{A}$ est silencieux, alors la simulation $Sym(\mathcal{A})$ l'est aussi. En effet, outre la régulation par l'unisson, pour que p fasse un pas de calcul, il faut soit que $\widehat{\mathcal{A}}(p) \neq p.curr$, donc le calcul n'est localement pas fini, soit que $\exists q \in N(p), q.c = p.c \oplus_B 1$, ce qui suggère qu'un nœud dans le réseau a eu besoin d'avancer pour poursuivre son calcul. Pour finir, une spécification est *statique* si elle demande de calculer un objet donné en un temps fini. Nous prouvons que si $\mathcal{A}$ satisfait une telle spécification, alors $Sym(\mathcal{A})$ est autostabilisant pour la même spécification.

Cette simulation est très efficace car elle nous permet d'obtenir des algorithmes autostabilisants silencieux asynchrones (1) totalement polynomiaux, (2) asymptotiquement optimaux en $(O(D))$ rondes et (3) utilisant une mémoire raisonnable. Par exemple (cf. Table 1), dans un réseau connexe identifié, nous obtenons un algorithme autostabilisant silencieux efficace d'élection de leader qui construit également un arbre couvrant en largeur enraciné au leader : cet algorithme stabilise en $O(D)$ rondes et $O(n^3)$ mouvements en utilisant $\Theta(\log N)$ bits par nœud où N est une borne supérieure sur n. Nous obtenons également des solutions autostabilisantes silencieuses efficaces pour, entre autres, le clustering et la construction d'un arbre en largeur dans un réseau enraciné semi-anonyme.

Problème	Mouvements	Rondes	Espace	
Arbre BFS dans un réseau enraciné	$O(n^3)$	$O(D)$	$\Theta(\log B + \log \Delta)$	
Arbre BFS dans un réseau identifié	$O(n^3)$	$O(D)$	$\Theta(\log N)$	$B \geq 2D+2$
Élection de leader	$O(n^3)$	$O(D)$	$\Theta(\log N)$	$N \geq n$
$O(\frac{n}{k})$-clustering	$O(n^3)$	$O(D)$	$\Theta(\log k + \log N)$	

TABLE 1 : Complexités de $Sym(\mathcal{A})$ en fonction de $\mathcal{A}$.

Références

[1] K. Altisen, S. Devismes, S. Dubois, and F. Petit. *Introduction to Distributed Self-Stabilizing Algorithms*. Synthesis Lectures on Distributed Computing Theory. Morgan & Claypool, 2019.

[2] B. Awerbuch, S. Kutten, Y. Mansour, B. Patt-Shamir, and G. Varghese. Time optimal self-stabilizing synchronization. In *25th Annual Symposium on Theory of Computing, (STOC'93)*, pages 652–661, 1993.

[3] J. Beauquier, M. Gradinariu, and C. Johnen. Cross-over composition - enforcement of fairness under unfair adversary. In *5th International Workshop on Self-Stabilizing Systems, (WSS'01)*, pages 19–34, 2001.

[4] L. Blin, C. Johnen, G. Le Bouder, and F. Petit. Silent anonymous snap-stabilizing termination detection. In *41st International Symposium on Reliable Distributed Systems, (SRDS'22)*, pages 156–165, 2022.

[5] C. Boulinier and F. Petit. Self-stabilizing wavelets and rho-hops coordination. In *22nd IEEE International Symposium on Parallel and Distributed Processing, (IPDPS'08)*, pages 1–8, 2008.

[6] C. Boulinier, F. Petit, and V. Villain. When graph theory helps self-stabilization. In *23rd Annual Symposium on Principles of Distributed Computing, (PODC'04)*, pages 150–159, 2004.

[7] A. Cournier, S. Rovedakis, and V. Villain. The first fully polynomial stabilizing algorithm for BFS tree construction. *Information and Computation*, 265 :26–56, 2019.

[8] J.-M. Couvreur, N. Francez, and M. G. Gouda. Asynchronous unison (extended abstract). In *12th International Conference on Distributed Computing Systems, (ICDCS'92)*, pages 486–493, 1992.

[9] A. K. Datta, S. Devismes, and L. L. Larmore. A silent self-stabilizing algorithm for the generalized minimal k-dominating set problem. *Theoretical Computer Science*, 753 :35–63, 2019.

[10] S. Devismes, D. Ilcinkas, C. Johnen, and F. Mazoit. Being efficient in time, space, and workload : a self-stabilizing unison and its consequences. In *(The 42nd International Symposium on Theoretical Aspects of Computer Science (STACS 2025)*, volume 327, pages 30 :1–30 :18, 2025.

[11] E. W. Dijkstra. Self-stabilization in spite of distributed control. *Communications of the ACM*, 17(11) :643–644, 1974.

[12] Y. Emek and E. Keren. A thin self-stabilizing asynchronous unison algorithm with applications to fault tolerant biological networks. In *40nd Symposium on Principles of Distributed Computing, (PODC'21)*, pages 93–102, 2021.

[13] Y. Emek and R. Wattenhofer. Stone age distributed computing. In *32nd Symposium on Principles of Distributed Computing, (PODC'13)*, pages 137–146, 2013.