On peut tromper mille personnes mille fois, mais pas plus

Lélia Blin[1], Anaïs Durand[2] et Sébastien Tixeuil[3]

[1] *Sorbonne Université, CNRS, Université d'Evry-Val-d'Essonne, LIP6, Paris*
[2] *LIMOS, Université Clermont Auvergne, Clermont-Ferrand*
[3] *Sorbonne Université, CNRS, LIP6, Paris*

Nous explorons la possibilité de concevoir des algorithmes auto-stabilisants pour des systèmes distribués à passage de messages où l'adversaire peut initialement placer un nombre arbitraire de messages corrompus (leur contenu est également déterminé par l'adversaire) dans les canaux de communications. Plus précisément, nous proposons un algorithme auto-stabilisant de $(\Delta+1)$-coloration pour un graphe arbitraire de degré maximum Δ dans le modèle à passage de messages asynchrone. Cet algorithme est déterministe et converge en $O(k\Delta n^2 \log n)$ échanges de messages, où k est une borne sur la capacité des canaux de communications et n est le nombre de processus. Notre algorithme utilise des messages de $O(\log\log n + \log\Delta)$ bits et une mémoire de $O(\Delta\log\Delta + \log\log n)$ bits par processus. La propriété fondamentale de notre algorithme est le fait que les processus ne nécessitent aucune connaissance sur le nombre de messages initialement présents dans les canaux (k est inconnu).

Pour concevoir notre protocole de coloration, nous utilisons une construction auto-stabilisante de graphe orienté acyclique couvrant, qui peut également servir d'outil pour résoudre d'autres tâches dans le même contexte.

Mots-clefs : Auto-stabilisation, canaux de communication non bornés, coloration

1 Introduction

L'*auto-stabilisation* [3] est une technique permettant à un système distribué de retrouver un comportement correct après avoir été frappé par des *pannes transitoires* qui peuvent toucher à la fois les processus participants et leur moyen de communication. Plus précisément, un algorithme auto-stabilisant permet au système de finir par retrouver une configuration correcte à partir d'une configuration initiale arbitraire, potentiellement entièrement corrompue. On peut même supposer que les valeurs des variables locales des processus ainsi que le contenu des canaux de communications peuvent être arbitrairement modifiés par un adversaire dans le but d'empêcher le retour à un comportement correct.

Assurer l'auto-stabilisation en présence de messages corrompus dans les canaux de communication est complexe. Il est en particulier difficile d'assurer la stabilisation si aucune borne n'est connue sur le nombre de messages (potentiellement corrompus) initialement en transit dans les canaux. Au contraire, si une telle borne est connue, il est possible de vider les canaux de communication, puis de réinitialiser les variables locales grâce à un protocole qui peut "faire confiance" aux messages reçus. Dans les réseaux "physiques", un faible (*i.e.* borné par une petite constante) nombre de messages peut être en transit entre deux machines directement connectées, mais dans les réseaux "logiques" (*e.g.* réseaux virtualisés ou pair-à-pair), une liaison entre deux machines peut contenir un nombre de messages en transit qui peut dépendre du nombre global de machines dans le réseau, les messages pouvant transiter par un grand nombre de machines physiques pour simuler une liaison entre deux machines logiques. Ainsi, pour un réseau de mille machines, de l'ordre de mille messages pourraient être en transit pour chaque connexion virtuelle vers cette machine. Or, les machines d'un réseau ignorent généralement la taille globale du réseau.

Si on ne connaît pas de borne sur le nombre de messages initialement présents dans un lien de communication, l'auto-stabilisation peut demander des ressources importantes. Typiquement, les solutions génériques comme les *data link protocols* [1] nécessitent des ressources non-bornées ce qui les rend irréalistes en pratique. Des solutions spécifiques existent pour certaines tâches (par exemple, pour construire un arbre)

mais elles nécessitent $O(n \log n)$ ou $O(\Delta \log n)$ bits de mémoire par processus, où n est les nombre de processus et Δ est le degré maximum du réseau, ce qui peut être un frein au passage à l'échelle.

Coloration. Dans le problème de la coloration des nœuds, chaque participant (ou nœud) dispose d'une couleur, et le but est d'atteindre une configuration où toute paire de nœuds voisins ont des couleurs différentes. Les algorithmes auto-stabilisants de coloration de nœuds existants dans le modèle à passage de messages n'offrent que des garanties probabilistes (*e.g.* [5]). De plus, nombre d'entre eux supposent des versions plus ou moins fortes du modèle synchrone. À notre connaissance, il n'existe aucun algorithme déterministe auto-stabilisant de coloration de nœuds pour un système asynchrone à passage de messages, où le nombre initial de messages transitant dans les canaux n'est pas connu par les processus.

Contributions. Nous proposons un algorithme auto-stabilisant de coloration de nœuds. Cet algorithme utilise $O(\log\log n + \Delta \log \Delta)$ bits de mémoire par processus et des messages de taille $O(\log\log n + \log \Delta)$ bits, où n est le nombre de processus et Δ est le degré maximum du réseau. Il ne nécessite aucune connaissance sur le nombre initial de messages dans les canaux de communications, ni sur le contenu des ces messages. Il converge après $O(k\,\Delta\, n^2 \log n)$ échanges de messages, où k est une borne sur la capacité des canaux de communications. (Cette borne k n'est utilisée que pour l'étude de complexité et n'est pas connue par les processus.)

L'outil principal utilisé pour concevoir ce protocole de coloration est un mécanisme servant à casser les symétries qui oriente localement chaque lien du réseau de façon à ce que, globalement, l'orientation soit acyclique. Autrement dit, ce mécanisme construit un graphe couvrant orienté acyclique. Cela permet de simplifier la conception d'algorithmes de plus haut niveau comme notre algorithme de coloration, mais qui peut également servir pour concevoir d'autres protocoles comme la construction d'un stable maximal.

Plan. Dans la section suivante, nous présentons brièvement le modèle de calcul considéré ici. Puis nous présentons les idées principales de notre mécanisme d'orientation d'arêtes dans la section 3 avant de présenter notre algorithme de coloration dans la section 4. Enfin, nous concluons dans la section 5.

2 Modèle de calcul

Le réseau est modélisé par un graphe $\mathcal{G} = (V, E)$ où les nœuds V représentent les processus et les arêtes E représentent les canaux de communication bidirectionnels. Les nœuds communiquent en s'échangeant des messages via des canaux de communications FIFOs (les messages envoyés sur un lien sont délivrés dans l'ordre de leur envoi). Nous notons $N(v)$ l'ensemble des *voisins* du processus v, autrement dit l'ensemble des nœuds ayant un lien de communication avec v. Le nombre de processus est noté n, le degré maximum du graphe est noté Δ et δ_v est le degré du nœud v.

Une *exécution* est une séquence maximale de *pas de calcul*. Pendant un pas de calcul, des processus (a) reçoivent des messages (au plus un par canal entrant), (b) font des calculs internes, et (c) envoient des messages (au plus un par canal sortant). Pour les besoins de l'analyse de complexité, nous définissons une *ronde* comme étant le plus petit fragment d'exécution tel que les deux conditions suivantes sont satisfaites : (a) tous les messages en transit au début de la ronde ont été reçus (et traités) par leur destinataire, et (b) tous les nœuds qui n'avaient aucun message en transit ont déclenché un *timeout* (et l'ont traité).

Identifiant. Chaque processus v dispose d'un identifiant unique ID_v mais il ne peut accéder qu'à un bit de son identifiant à la fois. La fonction $Bit_v(i)$ retourne la position (numérotée de droite à gauche) du $i^{\text{ème}}$ bit de poids fort égal à 1. Par exemple, si le nœud v a comme identifiant (en binaire) 1010, la fonction $Bit_v(i)$ peut être implémentée ainsi :

$$Bit_v(i) := \begin{cases} 4 & \text{if } i = 1 \\ 2 & \text{if } i = 2 \\ -1 & \text{if } i > 2 \end{cases}$$

Nous supposons que les identifiants s'écrivent sur $O(\log n)$ bits, par conséquent les valeurs retournées par $Bit_v(i)$ sont de taille $O(\log\log n)$. Notons qu'un nœud v ne connaît pas les identifiants de ses voisins, mais uniquement les numéros des ports de ses liens sortants. Par abus de langage, nous noterons ici u le numéro de port du lien (v,u) lorsqu'aucune ambiguité n'est possible.

On peut tromper mille personnes mille fois, mais pas plus

3 Orientation des arêtes

La première partie de notre solution est un protocole auto-stabilisant nommé DAG permettant de casser des symétries. Pour ce faire, le protocole construit un graphe couvrant orienté acyclique basé sur les identifiants des nœuds : une arête doit être orientée du nœud de plus petit identifiant vers le nœud de plus grand identifiant. Bien sûr, les nœuds ne connaissent pas les identifiants de leurs voisins et, pour atteindre nos objectifs en terme de taille de messages, les nœuds ne doivent pas s'échanger directement leur identifiant complet.

Chaque nœud v dispose d'une variable $ord_v(u)$ par lien sortant (v,u) qui indique l'orientation de l'arête (v,u). Plus précisément, $ord_v(u)$ doit être mis à 0 si $\text{ID}_v > \text{ID}_u$ et à 1 sinon. Pour mettre à jour $ord_v(u)$, les nœuds s'échangent constamment leur identifiant de façon compacte en utilisant $Bit_v(i)$ et une variable $cpt_v \in \{1,\ldots,\lceil \log \text{ID}_v \rceil\}$ de taille $O(\log\log n)$ bits.

Lorsque $cpt_v = 1$, v envoie un message $\langle 1 \rangle$ à tous ses voisins pour leur demander la position de leur 1$^{\text{er}}$ bit à 1. Lorsqu'un voisin u réponds par un message $\langle 1, B \rangle$, où B est la position du 1$^{\text{er}}$ bit à 1 de ID_u, trois cas sont possibles : (a) si $B > Bit_v(1)$, alors $\text{ID}_v < \text{ID}_u$ et $ord_v(u) := 1$, (b) si $B < Bit_v(1)$, alors $\text{ID}_v > \text{ID}_u$ et $ord_v(u) := 0$, (c) sinon, $B = Bit_v(1)$ et la comparaison bit à bit doit continuer. La variable $wait_v$ permet de stocker les ports vers les voisins n'ayant pas encore répondu lors de cette phase de la comparaison. Lorsque tous les voisins de v ont répondu, cpt_v est incrémenté et la comparaison continue avec les voisins pour lesquels l'orientation n'est pas encore déterminée, et ainsi de suite jusqu'à ce que tous les liens sortants de v soient orientés.

À cause de la configuration initiale arbitraire, les canaux de communications peuvent initialement contenir un nombre inconnu de messages corrompus. Le processus de comparaison doit donc se répéter indéfiniment. Pour éviter un interblocage dû à une absence de messages, un processus v fait une étape de comparaison (l'envoi d'un message $\langle cpt_v \rangle$ à u avec mise à jour préalable de cpt_v si nécessaire) dès lors qu'ils ne reçoivent aucun message sur un port u.

Théorème 1. DAG *est un algorithme auto-stabilisant qui construit un graphe couvrant orienté acyclique dans un graphe de n nœuds de degré maximum* Δ*. Il nécessite* $O(\log\log n + \Delta\log\Delta)$ *bits de mémoire par processus et des messages de taille* $O(\log\log n)$ *bits. Il stabilise en* $O(k\log n)$ *rondes avec* $O(k\Delta\log n)$ *messages échangés, où k est le nombre maximum de messages initialement dans les canaux de communication (k n'est pas connu par les processus).*

Par manque de place, les preuves de correction et de complexité de DAG ne sont pas détaillées ici, mais sont disponibles dans notre rapport technique [2].

4 Coloration

Dans cette section, nous présentons notre algorithme COLO de $(\Delta+1)$-coloration de nœuds qui utilise l'orientation construite par DAG. Chaque nœud dispose d'une variable de couleur $coul_v \in \{1,\ldots,\delta_v+1\}$. Pour chaque lien sortant vers un voisin u, v maintient la variable $coul_v(u)$ qui est la dernière couleur de u connue par v.

Infiniment souvent, v envoie sa couleur à ses voisins grâce à un message $\langle coul_v \rangle$. Si v a un voisin u ayant la même couleur que lui, autrement dit si v détecte un conflit, et si $\text{ID}_v < \text{ID}_u$ (plus précisément si $ord_v(u) = 1$), v change sa couleur et choisit la plus petite couleur non utilisée par ses voisins. En autorisant le changement de couleur uniquement lorsque l'identifiant d'un nœud est un minimum local parmi les nœuds en conflit, cela empêche (lorsque DAG est stabilisé) deux voisins de changer leurs couleurs simultanément.

Théorème 2. COLO *est un algorithme auto-stabilisant de* $(\Delta+1)$*-coloration de nœuds pour un graphe de n nœuds et de degré maximum* Δ*, en supposant un graphe couvrant orienté acyclique. Il nécessite* $O(\Delta\log\Delta)$ *bits de mémoire par nœuds et des messages de taille* $O(\log\Delta)$ *bits. Il stabilise après* $O(n)$ *rondes et* $O(\Delta k n)$ *messages échangés.*

Par manque de place, les preuves de correction et de complexité de COLO ne sont pas détaillées ici, mais sont disponibles dans notre rapport technique [2].

Pour éviter la famine d'un de nos algorithmes, DAG et COLO sont composés grâce à une *composition équitable* [4], autrement dit, chaque processus exécute alternativement un pas de calcul de DAG et un pas

de calcul de COLO. Comme démontré dans des travaux précédents [4], la composition équitable préserve la propriété d'auto-stabilisation.

Coloration minimale et stable maximal. Notez que notre algorithme COLO ne garantit pas une coloration localement minimale. En effet, si la configuration initiale est une coloration qui n'est pas localement minimale, aucun conflit n'est détecté et donc aucun changement de couleur n'est effectué. Cependant, une simple modification du protocole permet d'obtenir une coloration localement minimale : lorsqu'un nœud v ne détecte aucun conflit mais voudrait choisir une couleur c plus petite, il demande l'autorisation à ses voisins de plus grande identité. Un voisin u donne cette autorisation à v sauf s'il a la couleur c (mais v ne le savait pas) ou alors s'il veut lui-même changer de couleur et est en attente d'une autorisation. Grâce à ce mécanisme, aucun nouveau conflit n'est créé et la longueur d'une chaîne d'autorisation est bornée par la hauteur du graphe orienté acyclique.

Grâce à la coloration minimale obtenue, il est possible de résoudre le problème du stable maximal : chaque nœud de couleur 0 peut être vu comme faisant partie du stable maximal.

5 Conclusion

Nous avons proposé le premier algorithme déterministe auto-stabilisant pour des systèmes distribués asynchrones à passage de messages dont la capacité des liens de communication n'est pas connue et qui requière uniquement une mémoire et des messages de taille sous-logarithmique (en n, le nombre de nœuds) lorsque Δ (le degré maximum du graphe) est lui-même sous-logarithmique.

Notre approche est constructive et modulaire. En particulier, notre algorithme DAG résout une problématique fondamentale en auto-stabilisation : éviter les comportements cycliques. Nous pensons que cela peut être un outil intéressant pour résoudre d'autres problèmes dans des conditions similaires à celles considérées ici.

Nous pensons que nos algorithmes DAG et COLO peuvent être utiles pour la résolution d'autres tâches, comme par exemple le stable minimal, la coloration d'arêtes, ou le couplage maximal. Il serait également intéressant d'étudier leur utilité pour résoudre des tâches globales (comme la construction d'arbres ou l'élection de leader) et dans quelle mesure leur efficacité en terme de ressources nécessaires reste inchangée dans un tel contexte.

La seule propriété que nous avons conservée sur les liens de communication est leur comportement FIFO. Enlever cette hypothèse dans un contexte où la capacité des liens est inconnue va probablement générer de nombreux résultats d'impossibilité. Cela reste une question ouverte.

Références

[1] Yehuda Afek and Geoffrey M. Brown. Self-stabilization over unreliable communication media. *Distributed Computing*, 7(1) :27–34, 1993.

[2] Lélia Blin, Anaïs Durand, and Sébastien Tixeuil. Ressource efficient stabilization for local tasks despite unknown capacity links, 2020. https ://arxiv.org/abs/2002.05382.

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

[4] Shlomi Dolev. *Self-stabilization*. MIT Press, March 2000.

[5] Ted Herman and Sébastien Tixeuil. A distributed TDMA slot assignment algorithm for wireless sensor networks. In *AlgoSensors'04*, pages 45–58, 2004.