Autour de l'autostabilisation

Partie I : Techniques généralisant l'approche

Stéphane Devismes* — **Franck Petit**** — **Vincent Villain*****

* *VERIMAG, Université Joseph Fourier, Grenoble*
** *LiP6, Université Pierre et Marie Curie Paris 6, Paris*
*** *MIS, Université de Picardie Jules Verne, Amiens*

RÉSUMÉ. *Cet article constitue la première partie d'un survol des techniques dérivées de l'autostabilisation. L'autostabilisation permet de tolérer les fautes transitoires. Elle n'offre cependant pas toujours la possibilité d'escamoter certains effets indésirables de ces fautes, par exemple la perte temporaire de sûreté. Des études ont été menées ces dernières années en vue de palier ces inconvénients conduisant, pour les unes, à affaiblir l'autostabilisation, pour les autres au contraire à la renforcer. Dans cette première partie, nous traitons des généralisations de l'autostabilisation. Il s'agit de relâcher une des deux propriétés caractérisant la stabilisation, à savoir la fermeture ou la convergence, permettant ainsi de résoudre certains problèmes ayant été prouvés difficiles à résoudre ou franchement insolubles de manière autostabilisante. Nous proposons ici un état de l'art détaillé de ces différentes techniques. Dans une seconde partie (Devismes* et al.*, 2009), également au sommaire de ce numéro, nous traitons des propriétés visant à renforcer la stabilisation.*

ABSTRACT. *This paper constitues the first part of a survey on techniques derived from self-stabilization. Self-stabilisation is a versatile technique to design distributed algorithms that withstand transient faults. However, it also includes some drawbacks such as the temporary loss of safety. Recent research has been made to overcome these drawbacks leading to propose either to weaken or to enhance the stabilization. In this first part, we present the main weakened forms of self-stabilization. They are obtained by relaxing either the closure or the convergence property of self-stabilization, allowing to solve problems that are proven to be of high complexity or even impossible to solve in a self-stabilizing manner. In this paper, we present a detailed state-of-the-art of these techniques. In a second part, also published in this issue (Devismes* et al.*, 2009), we deal about strenghtened forms of self-stablization.*

MOTS-CLÉS : *Systèmes répartis, tolérance aux fautes, fautes transitoires, algorithmes distribués, autostabilisation*

KEYWORDS: *Distributed systems, fault-tolerance, transient faults, distributed algorithms, self-stabilization*

RSTI - TSI – 30/2011. Algorithmique distribuée, pages 873 à 894

1. Introduction

1.1. *Autostabilisation*

Le concept d'*autostabilisation* a été formulé pour la première fois par (Dijkstra, 1974). Un algorithme distribué est dit *autostabilisant* dans un système donné si à partir d'une configuration quelconque[1] du système, toute exécution de l'algorithme atteint en un temps fini (et sans intervention extérieure) une configuration – dite *légitime*[2] – à partir de laquelle tous les suffixes d'exécution possibles sont corrects, c'est-à-dire qu'ils vérifient tous la spécification du problème pour lequel l'algorithme a été conçu.

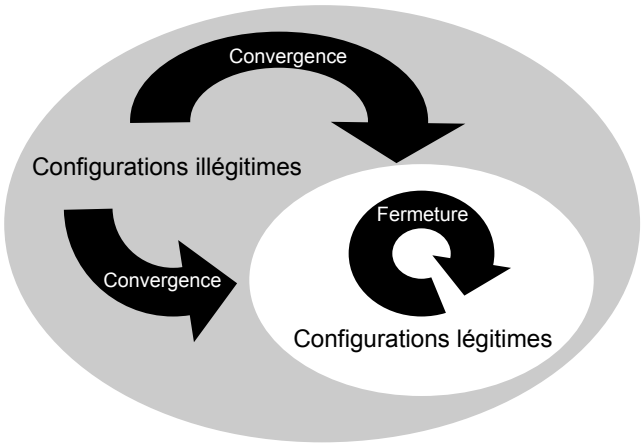

Figure 1. *Configurations du système*

Ainsi, tout algorithme distribué autostabilisant satisfait les deux propriétés suivantes (cf. figure 1) :

– *La convergence.* A partir de toute configuration initiale (en particulier illégitime), toute exécution de l'algorithme atteint une configuration légitime en temps fini. Cette phase de convergence est aussi appelée *phase de stabilisation* ;

1. C'est-à-dire une configuration où les variables des processeurs ont des valeurs quelconques et les canaux de communication contiennent un nombre arbitraire de messages transportant des valeurs quelconques.
2. Les configurations qui ne sont pas légitimes sont dites *illégitimes*.

– *La fermeture.* Toute configuration accessible à partir d'une configuration légitime est aussi une configuration légitime.

Dans cet article, nous parlons parfois d'autostabilisation déterministe pour différencier l'autostabilisation « classique » (définie ci-avant) de sa variante probabiliste définie dans la section 3.

1.2. *Exemple*

Le problème considéré dans les trois premiers algorithmes autostabilisants proposés dans (Dijkstra, 1974) est la circulation d'un jeton unique dans un anneau unidirectionnel avec racine. Dans un tel système, les configurations illégitimes peuvent contenir plusieurs jetons[3]. L'autostabilisation garantit alors qu'à partir d'une telle configuration illégitime, le système retrouve en temps fini une configuration à partir de laquelle un seul jeton circule.

1.3. *Les avantages de l'autostabilisation*

L'intérêt fondamental de l'autostabilisation est qu'elle permet une approche générale de la conception d'algorithmes distribués tolérants aux *fautes transitoires*. Une faute transitoire est une faute non définitive qui altère le contenu du composant du réseau (processeur ou canal de communication) où elle se produit. Une autre caractéristique importante des fautes transitoires est qu'elles sont supposées rares par opposition aux fautes intermittentes qui sont aussi non définitives mais supposées fréquentes[4]. Par exemple, une perte de message ou la corruption d'une partie de la mémoire locale d'un processeur peuvent être considérées comme des fautes transitoires. En supposant que les fautes transitoires n'altèrent pas le code de l'algorithme, pour être efficace, un algorithme autostabilisant doit donc être suffisamment rapide pour retrouver un comportement correct entre deux perturbations transitoires du système (cf. figure 2), ceci indépendamment du type de fautes transitoires qu'il subit. Ainsi, le *temps de stabilisation*, c'est-à-dire le temps maximal pour retrouver une configuration légitime à partir de n'importe quelle configuration illégitime, est l'une des métriques les plus importantes pour juger de l'efficacité d'un algorithme autostabilisant.

Comme expliqué dans (Dolev, 2000; Tel, 2001), l'autostabilisation comporte d'autres avantages. Par exemple, un algorithme autostabilisant ne nécessite pas d'initialisation. Or, dans un système distribué, cette phase est critique, en particulier lorsque le réseau est un système à large-échelle où des milliers de nœuds peuvent être géographiquement distants.

---

3. Nb., dans ces algorithmes, les configurations sont définies de telles sortes qu'elles contiennent toujours au moins un jeton.
4. Par *fréquente* nous entendons qu'il n'existe pas de borne inférieure sur le temps entre deux occurences de fautes.

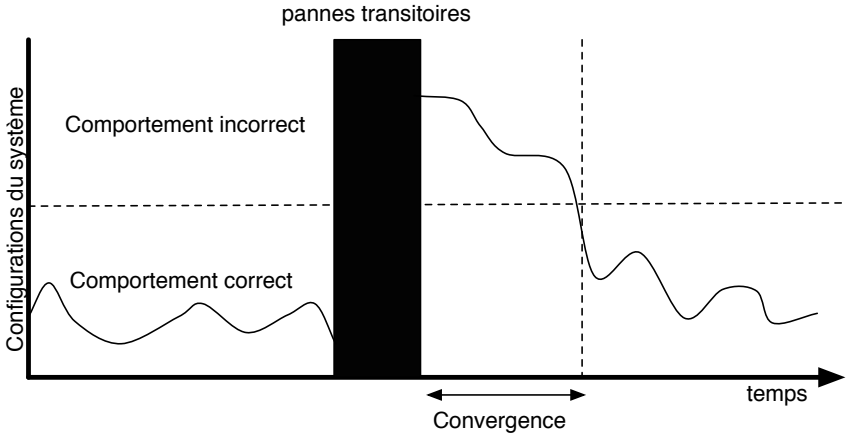

Figure 2. *Tolérance aux fautes transitoires*

En outre, de nombreux algorithmes autostabilisants, notamment les algorithmes de calcul de tables de routages ou d'arbres couvrants, tolèrent une certaine dynamicité (c'est-à-dire des modifications topologiques) du réseau au cours de l'exécution, à condition que cette dynamicité soit non silencieuse et de fréquence peu élevée. La dynamicité se définit en termes d'ajouts et/ou suppressions de nœuds et/ou de canaux de communications. Par dynamicité non silencieuse, nous entendons que les noeuds affectés par un changement topologique sont (ou finissent par être) informés de cette modification (par un protocole sous-jacent, par exemple). Par dynamicité de fréquence peu élevée, nous supposons l'existence de périodes sans changement topologique suffisamment longues pour permettre au système de stabiliser.

1.4. *Les inconvénients de l'autostabilisation*

Contrairement aux algorithmes distribués robustes, les algorithmes autostabilisants ne masquent pas l'effet des fautes qu'ils subissent. Ainsi, les fautes transitoires provoquent une perte de sûreté temporaire : aucune garantie n'est *a priori* donnée sur les calculs effectués durant la phase de stabilisation. De plus, dans la plupart des cas, les algorithmes autostabilisants ne permettent pas aux processeurs de détecter la fin de la phase de stabilisation.

Par définition, tout algorithme autostabilisant tolère les fautes transitoires. En revanche, la plupart des algorithmes autostabilisants ne sont pas conçus pour tolérer d'autres types de fautes, notamment lorsque les fautes sont définitives (arrêt, comportement byzantin) ou intermittentes. De ce fait, la plupart des algorithmes autostabilisants deviennent totalement inopérants en présence de telles fautes.

Il faut aussi noter que l'autostabilisation a un coût. Le surcoût[5] occasionné par l'autostabilisation est observable à la fois au niveau du temps d'exécution, de l'occupation mémoire et du nombre de messages échangés. Le fait que pour certains problèmes, l'autostabilisation nécessite l'ajout d'hypothèses supplémentaires sur le système constitue aussi un surcoût.

Enfin, il existe des problèmes pour lesquels il n'existe pas de solutions autostabilisantes. Grâce aux travaux de (Herman, 1990; Israeli et Jalfon, 1990b; Gouda et Multari, 1991; Katz et Perry, 1993), certaines limites de l'autostabilisation sont connues. Par exemple, dans les réseaux anonymes, beaucoup de problèmes – par exemple la circulation de jeton autostabilisante dans un anneau unidirectionnel anonyme (cf., (Herman, 1990; Israeli et Jalfon, 1990b)) – n'ont pas de solutions autostabilisantes. (Gouda et Multari, 1991) caractérisent une classe de problèmes incluant le « bit alterné » n'ayant aucune solution autostabilisante dans un système à passage de messages où les capacités des canaux de communications sont non bornées et où les mémoires locales des processeurs sont bornées. (Katz et Perry, 1993) démontrent que certains problèmes spécifiés en termes de configurations souhaitées et configurations exclues ne peuvent pas être résolus de manière autostabilisante.

1.5. *Techniques dérivées*

Plusieurs propriétés dérivées de l'autostabilisation ont été introduites pour contourner les inconvénients présentés ci-avant. Ces propriétés peuvent être divisées en deux catégories principales résumées dans la figure 3 :

– les *propriétés plus faibles* – ou *généralisations* – que l'autostabilisation ;

– les *propriétés plus fortes* – ou *spécialisations* – que l'autostabilisation.

Nous proposons ici d'étudier des *généralisations* de l'autostabilisation[6], c'est-à-dire, des propriétés vérifiées par tout algorithme autostabilisant mais aussi par des algorithmes non autostabilisants. Les propriétés présentées sont obtenues en affaiblissant la définition de la fermeture ou de la convergence. Elles ont été introduites pour deux raisons majeures : elles permettent de résoudre des problèmes qui n'ont pas de solutions autostabilisantes et/ou d'écrire des algorithmes moins coûteux que des solutions autostabilisantes.

Parmi les généralisations proposées dans la littérature, nous allons présenter la *pseudostabilisation* (section 2), la *stabilisation probabiliste* (section 3), la *stabilisation faible* (section 4), et la k*-stabilisation* (section 5).

Plus précisément, dans chacune des sections, une généralisation est définie et positionnée par rapport à l'autostabilisation. Un rappel des principaux résultats la concer-

5. Cette notion de surcoût est discutée en détail dans (Cournier, 2009).
6. Les *spécialisations* de l'autostabilisation sont abordées dans (Devismes *et al.*, 2009).

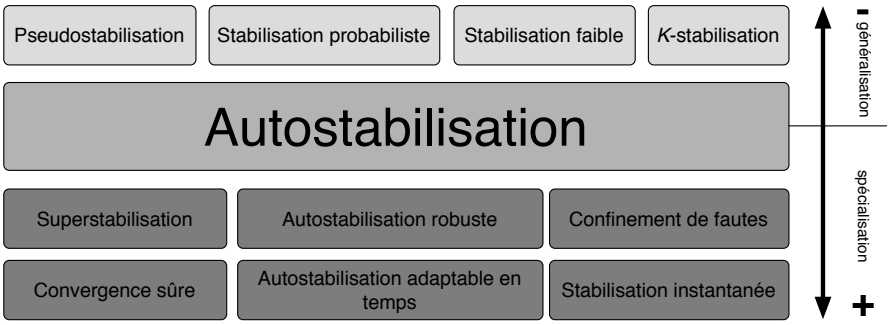

Figure 3. *Autour de l'autostabilisation*

nant est proposé. Enfin, la généralisation sera illustrée par des algorithmes l'implantant.

Il faut noter que deux principaux modèles sont utilisés par ces algorithmes : le *modèle à passage de messages* et le *modèle à états* (une abstraction du modèle à passage de message). Cependant, chacun des algorithmes présentés supposent des hypothèses différentes qui rendent difficile l'aggrégation des modèles dans une unique section. Par souci de clarté, nous avons donc choisi de présenter ces modèles de manière informelle et au fil de l'eau. Une présentation formelle et détaillée de ces modèles peut être trouvée dans (Dolev, 2000; Tixeuil, 2009), par exemple.

2. Pseudostabilisation

2.1. *Définition*

La *pseudostabilisation* a été introduite dans (Burns *et al.*, 1993). Cette propriété est très proche de l'autostabilisation. La différence essentielle tient dans le fait qu'un algorithme autostabilisant atteint en un temps fini une configuration (légitime) à partir de laquelle le système ne peut plus dévier de sa spécification, alors qu'un algorithme pseudostabilisant finit par ne plus dévier de sa spécification. En d'autres termes, un algorithme autostabilisant a l'assurance d'atteindre une configuration légitime et toutes les exécutions (ou suffixes d'exécution) possibles à partir de cette configuration sont toujours correctes. Au contraire, un algorithme pseudostabilisant (non autostabilisant) peut atteindre une configuration *pseudolégitime*[7] à partir de laquelle le comportement de l'algorithme peut être correct pendant un temps non borné puis dévier vers une configuration illégitime puis reconverger à nouveau. Cependant le nombre de fois où

7. Nb. l'ensemble des configurations pseudolégitimes est un sur-ensemble des configurations légitimes.

l'algorithme dévie d'une configuration pseudolégitime vers une configuration illégitime est fini : après un nombre fini de déviations l'exécution atteint une configuration légitime. De ce fait, toute exécution d'un algorithme pseudostabilisant comporte un suffixe d'exécution non vide correct. Ainsi la pseudostabilisation se définit comme suit :

– *convergence :* à partir de toute configuration, le système converge en un temps fini vers une configuration pseudolégitime ;

– *fermeture (faible) :* toute exécution ne dévie qu'un nombre fini de fois d'une configuration pseudolégitime vers une configuration illégitime.

Il est important de noter que dans le cas général, un système pseudostabilisant peut rester un temps non borné dans les configurations pseudolégitimes avant de dévier vers une configuration illégitime. Il existe alors des exécutions qui ne convergent jamais vers une configuration légitime mais qui contiennent un suffixe correct uniquement constitué de configurations pseudolégitimes. De ce fait, le temps maximal pour atteindre un suffixe d'exécution où le système ne dévie plus jamais de sa spécification n'est pas bornable : la notion de temps de stabilisation n'a plus de sens en pseudostabilisation.

La figure 4 représente l'ensemble des configurations d'un algorithme pseudostabilisant non autostabilisant. Dans ce schéma, les flèches représentent les transitions possibles entre états. Il est clair que cet algorithme est pseudostabilisant car il est impossible de construire une exécution (infinie) qui comporte un suffixe composé de configurations illégitimes : le système converge vers une configuration pseudolégitime quelle que soit la configuration initiale, ensuite il dévie au plus deux fois d'une configuration pseudolégitime vers une configuration illégitime. Nous pouvons aussi remarquer que l'algorithme n'est pas autostabilisant car il existe des exécutions possibles où le système reste pour toujours dans les configurations pseudolégitimes sans jamais atteindre une configuration légitime (pour cela, il suffit que l'exécution suive pour toujours le cycle constitué de configurations pseudolégitimes « non légitimes »). Dans ce cas, l'exécution atteint un suffixe correct mais il n'est jamais garanti que le système ne déviera plus : cela est dû à l'existence d'un cycle et de transitions non déterministes permettant de quitter ce cycle. Ces transitions s'expliquent par l'ordonnancement asynchrone des actions exécutées par des processeurs, qui est lui aussi non déterministe.

2.2. *État de l'art*

A notre connaissance actuellement seulement deux articles proposent des algorithmes pseudostabilisants : (Burns *et al.*, 1993; Delporte-Gallet *et al.*, 2010).

Dans l'article originel (Burns *et al.*, 1993), les auteurs démontrent que la pseudostabilisation est une généralisation de l'autostabilisation, c'est-à-dire, tout algorithme autostabilisant est aussi pseudostabilisant mais il existe des problèmes n'admettant pas de solutions autostabilisantes qui peuvent être résolus par un algorithme pseudos-

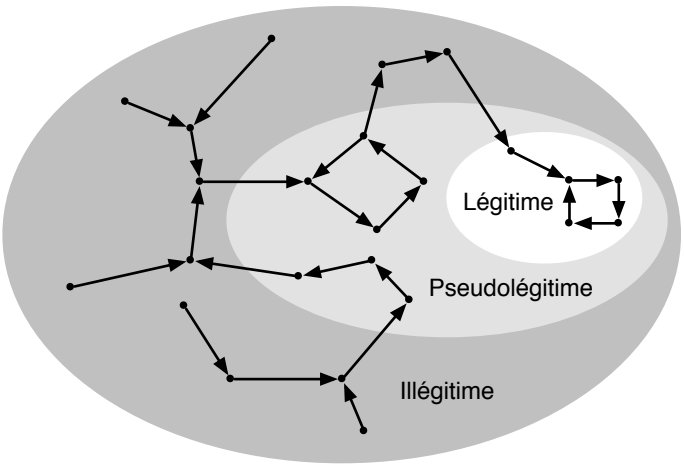

Figure 4. *Pseudostabilisation*

tabilisant. Cette preuve est constructive : les auteurs proposent un algorithme pseudostabilisant de *bit alterné* dans un modèle à passage de messages où les liens de communication sont non fiables (avec perte équitable), la capacité des liens est non bornée, mais où la mémoire locale des processeurs est bornée. Ce problème a été précédemment démontré sans solution autostabilisante dans (Gouda et Multari, 1991).

(Delporte-Gallet *et al.*, 2010) considèrent l'élection de leader dans des systèmes à passage de messages partiellement synchrones munis d'un réseau d'interconnexion complet où un nombre fini mais quelconque de pannes (pannes transitoires ou pannes définitives de processeurs) peut survenir. Cet article montre que pour obtenir des solutions autostabilisantes dans ce type de systèmes, notamment pour le problème d'élection, il est nécessaire de supposer que les pannes définitives sont statiques, c'est-à-dire, qu'il existe un temps à partir duquel il n'y a plus de pannes définitives. De plus, même en présence de pannes définitives statiques, il devient rapidement impossible de trouver des algorithmes d'élection autostabilisants dès lors que l'on relâche les hypothèses de synchronisme de tels systèmes. En revanche, il est possible d'écrire des algorithmes d'élection pseudostabilisants fonctionnant en dépit de pannes définitives dynamiques (c'est-à-dire non statiques) et sous des hypothèses de synchronisme très faibles. En fait, ces systèmes correspondent aux systèmes où l'élection robuste (c'est-à-dire, tolérante aux pannes définitives uniquement) peut être résolue.

2.3. *Exemple d'algorithme*

Nous allons maintenant illustrer la notion de pseudostabilisation avec l'algorithme de *bit alterné* proposé dans (Burns *et al.*, 1993). Le but de cet algorithme est d'établir

des communications fiables entre deux processeurs, c'est-à-dire qu'en dépit du caractère non fiable du lien de communication, l'algorithme donne accès à deux primitives `diffuse` et `délivre` telles que : si un processeur p appelle `diffuse`(m,q) où m est un message et q un autre processeur, alors q finit par exécuter une fois `délivre`(m,p). Dans ce cas, on dit que p diffuse le message m à q et que q délivre le message m provenant de p.

Dans cet algorithme, on considère, sans perte de généralité, deux processeurs à mémoire locale finie p et q où seulement p diffuse des messages de données à q. Ces deux processeurs sont reliés par un canal de communication FIFO, bidirectionnel, asynchrone, de capacité non bornée. Les communications sont effectuées par envoi et réception de messages. Le canal de communication est supposé *non fiable* mais *intègre*, c'est-à-dire que le canal peut perdre des messages mais il n'y a ni corruption ni duplication de message. Enfin, les pertes de messages sont supposées *équitables* : si l'un des processeurs envoie une infinité de messages alors l'autre en recevra une infinité.

Le principe de l'algorithme est le suivant : pour diffuser un message de donnée m, p envoie régulièrement un message contenant m et une estampille (0 ou 1) à q jusqu'à ce qu'il recoive un message d'accusé de réception de q. Pour cela, chaque processeur dispose d'une variable booléenne « estampille » initialement égale à 0 (lorsque le système est correctement initialisé). p démarre une diffusion de la donnée m en envoyant un message contenant m et la valeur de son estampille. Il attend ensuite la réception d'un accusé de réception de la part de q. Si au bout d'un certain temps, p n'a pas reçu d'accusé de réception, il conclut sans se tromper que le message qu'il a envoyé ou son accusé de réception s'est perdu. Pour cela, p utilise un mechanisme d'attente (*temporisateur*) et renvoie son message à l'expiration du délai d'attente. Ainsi, q peut recevoir plusieurs duplicatas du même message si ses accusés de réception sont perdus. L'estampille de q permet alors d'éviter de délivrer plusieurs fois le même message. Lorsque q reçoit la première copie du message, l'estampille du message est égale à son estampille locale, q délivre le message, change la valeur de son estampille et envoie un accusé de réception à p. Toutes les autres copies du message reçues ne sont pas délivrées car leur estampille sera différente de celle de q. Cependant, q accuse la réception de ces messages car la réception d'une copie du message de p signifie que l'accusé de réception précédent a été perdu. Lorsque p reçoit l'accusé de réception attendu, il change la valeur de son estampille (ainsi, les valeurs d'estampille de p et q sont à nouveau égales). Le canal de communication étant vide (p est supposé ne jamais se tromper lorsqu'il ré-émet un message), le système est dans un état similaire à l'état initial et p prêt à diffuser à nouveau un message.

Si le système démarre d'une configuration quelconque, plusieurs diffusions peuvent échouer sans que p ne le détecte. En effet, le canal de communication peut initialement contenir plusieurs accusés de réception et plusieurs messages dont la réception par q provoquera l'envoi d'un accusé de réception. Dans ce cas, si le délai d'attente n'est jamais dépassé, p peut initier une diffusion alors que le canal n'est pas vide. p peut alors recevoir un accusé de réception ne correspondant pas au message

diffusé et ainsi, p ne détecte pas l'éventuelle perte du message. Sans perte de messages, le système fonctionne tout de même mais le canal ne se vide pas. Donc, il n'est pas possible de garantir la convergence du système vers une configuration à partir de laquelle les diffusions sont toujours fiables. Cependant, le système ne peut perdre qu'un nombre fini de messages car chaque échec de diffusion résulte d'une perte de messages et d'une réception d'un mauvais accusé de réception : dans ce cas, le nombre de messages dans le canal diminue et après un nombre fini d'échecs, le système se retrouve dans une configuration où il démarre une nouvelle diffusion alors que le canal est vide. Ainsi, ce protocole, qui n'est pas autostabilisant, est bien pseudostabilisant.

3. Autostabilisation probabiliste

3.1. *Définition*

L'*autostabilisation probabiliste* a été introduite dans (Israeli et Jalfon, 1990b; Israeli et Jalfon, 1990a; Herman, 1990; Afek et Brown, 1993). L'idée consiste à relâcher la notion de convergence en passant d'une convergence *déterministe certaine* à une convergence *probabiliste* (incertaine). La propriété de fermeture reste la même que pour l'autostabilisation déterministe.

La convergence probabiliste est définie comme suit : à partir de toute configuration initiale, toute exécution de l'algorithme converge vers une configuration légitime *avec probabilité 1*. Ainsi, pour un algorithme autostabilisant probabiliste, il existe des exécutions (infinies) qui ne convergent pas. Cependant, à partir de toute configuration, il existe toujours au moins une exécution possible qui converge et plus le temps passe plus les chances de suivre une exécution qui ne converge pas diminuent. Ainsi la probabilité que l'exécution ne converge pas est nulle. En ce sens, un algorithme autostabilisant probabiliste est un algorithme de *Las Vegas* : il converge en un temps qui peut devenir très grand avec faible probabilité.

Il faut noter que pour obtenir des algorithmes autostabilisants probabilistes, deux principes sont généralement utilisés :

– l'exécution de certaines actions est guidée par des tirages aléatoires, comme dans (Gradinariu et Tixeuil, 2000), et/ou

– les valeurs de certaines variables de sortie sont déterminées en partie par des tirages aléatoires, comme dans (Bernard *et al.*, 2009).

3.2. *État de l'art*

L'autostabilisation probabiliste est utilisée principalement pour deux raisons : résoudre des problèmes qui n'ont pas de solutions autostabilisantes et écrire des algorithmes moins coûteux.

Tout d'abord, il existe plusieurs problèmes – parmi lesquels le coloriage de sommets, la circulation d'un jeton unique et l'élection – qui n'ont pas de solutions autostabilisantes déterministes dans un réseau anonyme et uniforme[8]. Ces résultats d'impossibilité sont liés aux symétries pouvant exister dans une configuration illégitime (Angluin, 1980) : si la configuration initiale est une configuration illégitime comportant des symétries, alors il est impossible de garantir la suppression de toutes ces symétries quelle que soit l'exécution. Par exemple, un algorithme autostabilisant de coloriage de sommets doit assurer qu'à partir d'une configuration initiale quelconque, le système atteint en temps fini une configuration dans laquelle chaque nœud a une couleur (un état) qui diffère de celles de ses voisins. Considérons un réseau de deux processeurs anonymes voisins et une configuration initiale où les deux nœuds ont le même état. A partir de cette configuration, au moins l'un des deux doit changer sa couleur. Or, les deux nœuds sont indistinguables, donc les deux nœuds peuvent changer leurs couleurs. Si le protocole de coloriage est déterministe, alors la nouvelle couleur choisie sera la même pour les deux nœuds. Ainsi, dans une exécution complètement synchrone, les deux nœuds ne peuvent arriver à se distinguer. D'où le résultat d'impossibilité. Plusieurs méthodes sont utilisées pour contourner ces résultats d'impossibilité : considérer des réseaux semi-anonymes (en particularisant un des nœuds, appelé *racine*), supposer que les nœuds agissent de manière séquentielle, ou encore utiliser l'autostabilisation probabiliste.

Le problème du 2-coloriage de sommets autostabilisant est démontré impossible à résoudre de manière déterministe pour plusieurs classes de réseaux anonymes biparties dans (Shukla *et al.*, 1995). Dans (Bernard *et al.*, 2009), il est démontré que le problème du coloriage de sommets n'a pas de solutions autostabilisantes déterministes dans les réseaux anonymes unidirectionnels. Ce résultat s'étend aux réseaux bidirectionnels. Un algorithme autostabilisant probabiliste pour réseaux bidirectionnels anonymes quelconques est proposé dans (Gradinariu et Tixeuil, 2000). (Herman, 1990) reprend les arguments développés dans (Angluin, 1980) pour établir que la circulation de jeton n'a pas de solution autostabilisante déterministe dans un anneau unidirectionnel anonyme. Or, plusieurs solutions autostabilisantes probabilistes existent, par exemple dans (Herman, 1990; Israeli et Jalfon, 1990b). De même, l'élection autostabilisante déterministe dans un arbre anonyme est démontré impossible dans (Devismes *et al.*, 2008), alors qu'une solution autostabilisante probabiliste peut trivialement être déduite de l'algorithme déterministe de calcul de centre dans un arbre anonyme proposé dans (Bruell *et al.*, 1999).

L'autostabilisation probabiliste a aussi été utilisée dans le but de réduire les ressources liées aux communications comme la taille des messages et les variables qui leur sont dédiées (registres partagés ou buffers d'émission/réception des messages). Par exemple, la circulation de jeton pour anneaux semi-anonymes unidirectionnels proposée dans (Herman, 1992) s'inspire directement du premier protocole détermi-

8. Dans de tels réseaux, les nœuds sont indistingables, c'est-à-dire qu'ils sont dépourvus d'identités et ont tous le même programme. Seul le nombre de liens de communication incidents (le degré) peut différer d'un nœud à l'autre.

niste de (Dijkstra, 1974). Là où ce dernier nécessite que les nœuds voisins partagent des registres d'au moins n états (ou $\log n$ bits, n étant le nombre de nœuds de l'anneau), la solution proposée dans (Herman, 1992) ne requiert que des registres partagés de 3 états (ou 2 bits) seulement. Notons cependant que cette performance est obtenue en supposant que la racine puisse accèder à une séquence (et donc une mémoire) infinie et apériodique de lettres dans l'alphabet $\{0, 1, 2\}$.

3.3. *Exemples d'algorithmes*

Nous allons présenter deux algorithmes autostabilisants probabilistes. Le premier est l'algorithme de coloriage de nœuds pour réseaux bidirectionnels quelconques de (Gradinariu et Tixeuil, 2000). Le second est un algorithme de circulation de jeton pour anneaux unidirectionnels enracinés provenant de (Herman, 1992). L'algorithme de (Gradinariu et Tixeuil, 2000) illustre le fait que l'autostabilisation probabiliste permet de résoudre des problèmes n'ayant pas de solution déterministe. L'algorithme de (Herman, 1992) illustre le fait que l'autostabilisation probabiliste permet d'obtenir des solutions moins coûteuses en ressources que l'autostabilisation déterministe. Les deux algorithmes présentés fonctionnent dans un modèle à mémoire partagée appelé *modèle à états*. Ce modèle est décrit brièvement ci-après.

3.3.1. *Le modèle à états*

Dans le modèle à états, les nœuds (ou processeurs) communiquent par le biais de variables localement partagées : chaque nœud dispose de variables qui peuvent être lues par ses voisins mais où il est le seul à pouvoir écrire. Le programme de chaque nœud consiste en un ensemble d'actions gardées. La garde d'une action est un prédicat défini sur les variables du nœud et de ses voisins. Une action peut être exécutée seulement si sa garde est vraie. Dans ce cas, l'action est dite *activable*. Un nœud est dit *activable* lorsqu'au moins une de ses actions est activable. L'évaluation des gardes et l'exécution des actions sont supposées atomiques (on parle alors d'atomicité composite en lecture et écriture). Les actions sont exécutées en fonction d'un *démon* responsable du degré de synchronisme du système. Lorsque celui-ci est *asynchrone*[9], tant qu'il existe des nœuds activables, ce dernier sélectionne un sous-ensemble non vide de nœuds activables à chaque étape (ou pas) de calcul. Les nœuds sélectionnés exécutent atomiquement une de leurs actions activables et puis l'étape de calcul suivante est exécutée, etc. Lorsque le démon sélectionne l'ensemble des nœuds à chaque pas de calcul, le système est dit *synchrone*.

3.3.2. *Coloriage des sommets dans un réseau bidirectionnel anonyme*

L'algorithme de (Gradinariu et Tixeuil, 2000) fonctionne sous l'hypothèse d'un *démon distribué inéquitable*. Cette hypothèse est la plus faible du modèle à états. En

9. Par opposition à d'autres modèles n'imposant pas l'atomicité de l'évaluation des gardes et de l'exécution des actions, on parle aussi de système *semi-synchrone*.

effet, les choix effectués par un démon distribué inéquitable ne sont pas contraints, excepté le fait qu'au moins un nœud doit être activé à chaque étape tant qu'il existe des nœuds activables.

Dans cet algorithme, chaque nœud dispose d'une variable *couleur* de domaine $\{1 \ldots \Delta + 1\}$ où Δ est le degré maximum des processeurs du réseau. Le programme d'un nœud consiste en une unique action : lorsqu'un nœud détecte qu'il a une couleur identique à celle d'un de ses voisins, il tire uniformément une valeur aléatoire parmi 0 et 1, puis change sa couleur seulement si la valeur tirée est 1. Lorqu'un nœud change sa valeur, il la change pour la valeur la plus grande qui n'est pas utilisée par l'un de ses voisins.

Dans cet algorithme, le tirage aléatoire uniforme permet de briser les symétries avec probabilité 1 en dépit des choix du démon. En effet, même si l'exécution est synchrone, c'est-à-dire, même si le démon choisit tous les nœuds activables à chaque étape, les changements de couleurs s'effectuent de manière asynchrone grâce aux tirages aléatoires. De plus, dès que la couleur d'un nœud n'est plus en conflit avec celles de ses voisins, le nœud devient inactivable pour toujours. De ce fait, le système converge avec probabilité 1.

3.3.3. *Circulation d'un jeton unique dans un anneau unidirectionnel*

L'algorithme présenté dans (Herman, 1992) reprend les principes du premier algorithme de circulation de jeton de (Dijkstra, 1974). Dans l'algorithme de Dijkstra, chaque nœud dispose de $O(n)$ états. De plus, l'algorithme nécessite que chaque nœud connaisse une borne supérieure de la taille de l'anneau. A l'inverse, l'algorithme probabiliste de (Herman, 1992) permet, d'une part, de réduire le nombre d'états nécessaire par nœud à 3. D'autre part, il n'est plus nécessaire de connaître une borne sur n. Cependant, il fonctionne sous l'hypothèse d'un démon distribué aléatoire uniforme, c'est-à-dire qu'à chaque étape le démon choisit un sous-ensemble non vide de nœuds activables de manière aléatoire uniforme.

Dans cet algorithme, chaque nœud dispose d'une variable partagée à 3 états possibles 0, 1 et 2 lisible uniquement par son successeur. La racine de l'anneau détient un jeton lorsque sa variable a la même valeur que celle de son prédécesseur dans l'anneau. Les autres nœuds détiennent un jeton lorsque leur variable n'a pas la même valeur que celle de leur prédécesseur. Lorsqu'un nœud détient un jeton, il change (ou essaie de changer) sa valeur afin de passer le jeton à son successeur.

Dans un tel système, il existe toujours au moins un nœud qui détient un jeton. Le but est alors de faire converger le système vers une configuration à partir de laquelle il existe exactement un jeton dans le réseau.

Lorsqu'il existe plus d'un jeton dans le réseau, l'algorithme va retarder la circulation d'un jeton pendant une période aléatoire. Ce jeton sera alors rattrapé par un autre jeton et fusionnera avec lui. Ainsi, le nombre global de jetons diminuera, d'où la convergence probabiliste.

Les jetons peuvent être retardés aléatoirement uniquement au niveau par la racine : lorsque la racine détient un jeton, elle tire uniformément une valeur aléatoire parmi 0, 1 et 2. Si elle tire son ancienne valeur, le jeton est retardé. Sinon, si elle choisit la même valeur que celle de son successeur le nombre de jetons diminue. Si elle choisit une valeur différente de celles de son prédécesseur et de son successeur, le jeton est passé au successeur. Lorsqu'un nœud non racine détient un jeton, il change sa valeur pour celle de son prédécesseur. Soit la nouvelle valeur est identique à celle de son successeur et dans ce cas le nombre global de jetons diminue. Soit la nouvelle valeur est différente de celle de son successeur : le jeton est passé au successeur.

Nous pouvons remarquer que l'amélioration de l'occupation mémoire se fait au prix d'un surcoût en temps : une fois dans une configuration légitime, la racine ne pouvant pas détecter la stabilisation, continue à ralentir le jeton.

4. Stabilisation faible

4.1. *Définition*

La *stabilisation faible* a été définie dans (Gouda, 2001). Cette propriété garde la même définition de fermeture que l'autostabilisation mais sa définition de convergence est plus faible. En effet, un algorithme faiblement stabilisant assure seulement une *possibilité de convergence déterministe* alors qu'un algorithme autostabilisant (déterministe) assure une *convergence déterministe certaine*.

La propriété de convergence de la stabilisation faible garantit qu'à partir de n'importe quel état du système *il existe* au moins une exécution qui converge. Ainsi, un algorithme faiblement stabilisant peut ne jamais converger bien qu'en tout point de l'exécution il a au moins une possibilité de converger. En fait, la convergence dépend de l'ordre dans lequel les nœuds exécutent le code de l'algorithme. En supposant que l'ordonnancement est géré par un adversaire, l'algorithme ne converge jamais. Au contraire, si l'ordonnancement est « amical » l'algorithme converge. Ainsi, le caractère non déterministe de la propriété de convergence est dû à l'ordonnancement des tâches uniquement et non aux tâches elles-mêmes, contrairement à l'autostabilisation probabiliste où l'ordonnancement et les actions exécutées sont non déterministes.

4.2. *État de l'art*

En l'état actuel de nos connaissances, seulement trois articles traitent de la stabilisation faible (Gouda, 2001; Devismes *et al.*, 2008; Cohen *et al.*, 2008).

(Gouda, 2001) introduit la notion de stabilisation faible. De plus, il propose des règles de composition d'algorithmes permettant de conserver la propriété de stabilisation faible.

(Devismes *et al.*, 2008) s'intéressent aux puissances d'expression relatives de la stabilisation faible, de l'autostabilisation déterministe et de l'autostabilisation probabiliste. Ils établissent que la stabilisation faible a un pouvoir d'expression plus fort que l'autostabilisation déterministe : c'est-à-dire, qu'elle permet de résoudre plus de problèmes que l'autostabilisation déterministe. Ils démontrent ce fait en proposant un algorithme de circulation de jeton pour anneau unidirectionnel anonyme et deux algorithmes d'élection pour arbres anonymes. Ces deux problèmes sont prouvés insolubles de manière autostabilisante déterministe dans (Herman, 1990; Devismes *et al.*, 2008). Ils affinent ensuite des résultats antérieurs de (Gouda, 2001) pour prouver que dans le modèle à états (défini dans la section 3.3.1), un protocole faiblement stabilisant déterministe peut être transformé en un protocole autostabilisant probabiliste si les nœuds utilisent un nombre fini d'états et si le démon est aléatoire uniforme (défini dans la section 3.3.3).

(Cohen *et al.*, 2008) considèrent des problèmes de stabilisation *égoïste*. Dans ce type de problème, les nœuds du réseau souhaitent établir une structure globale (un arbre couvrant, par exemple). La construction de cette structure a un coût pour chaque nœud. Chaque nœud applique alors une stratégie « égoïste » afin de minimiser son propre coût. Le problème est alors d'arriver à un équilibre global où tout nœud augmenterait son coût en changeant unilatéralement de stratégie. Le problème devient complexe à partir du moment où les fonctions de calcul de coût sont différentes. Si on considère le problème de l'arbre couvrant avec un seul type de fonction de coût, calculer un équilibre revient à calculer un arbre couvrant de coût minimum. Cohen *et al.* démontrent que dans le cas où le nombre de fonctions de coût est supérieur ou égal à trois, ce problème devient NP-difficile. Enfin, ils proposent un algorithme faiblement stabilisant de calcul d'arbre équilibré dans le cas où le nombre de fonctions différentes est égal à deux.

4.3. *Exemple d'algorithme*

Nous présentons maintenant l'un des deux algorithmes d'élection dans un arbre anonyme présenté dans (Devismes *et al.*, 2008). Cet algorithme est écrit dans le modèle à états. Chaque nœud p distingue localement ses voisins avec un numéro entre 0 et $\delta_p - 1$ où δ_p est le degré de p et partage une unique variable « pointeur » Par_p avec ses voisins. Le domaine de Par_p est $\{0 \dots \delta_p - 1\} \cup \{\perp\}$. $Par_p = \perp$ signifie que p pense être le leader. Lorsque $Par_p \neq \perp$, Par_p désigne comme *père* le voisin q que p pense être le plus proche du leader (dans ce cas, p est dit *fils* de q). L'algorithme est composé de trois règles afin de toujours assurer une possibilité de convergence.

1) Lorsqu'un nœud p est désigné par les pointeurs de tous ses voisins, cela signifie que tous ses voisins le considère comme leader. En conséquence, p affecte Par_p à $\perp$. Cette règle permet d'assurer que le système ne peut pas rester bloqué dans une configuration sans leader. En effet, dans une configuration sans leader, il y a toujours au moins un nœud activable pour cette règle car le réseau ne contient pas de cycle ;

2) Lorsqu'un nœud p a un voisin q qui n'est ni son père ni son fils, cela signifie que p et q ne s'accordent pas sur le même leader. Dans ce cas, p change de père simplement en incrémentant Par_p modulo δ_p. Ainsi, à partir de n'importe quelle configuration, il existe toujours une exécution possible permettant à chaque nœud non leader de s'accorder avec ses voisins non leader sur le même leader ;

3) La dernière règle permet d'éliminer des leaders, lorsqu'il n'y a pas de leader unique. Si un nœud p vérifie $Par_p = \perp$ mais n'est pas le père de tous ses voisins, alors il perd son status de leader en désignant avec Par_p l'un de ses voisins non fils.

En fait, cet algorithme calcule de manière faiblement stabilisante une orientation consistante des arêtes de l'arbre. Lorsque cette orientation est consistante, un leader unique existe et le système reste stable.

5. k-stabilisation

5.1. *Définition*

La *k-stabilisation* a été introduite dans (Beauquier *et al.*, 1998). Cette propriété est plus faible que l'autostabilisation car elle n'assure la convergence qu'à partir d'un sous-ensemble des configurations possibles. L'idée est de considérer des systèmes où une borne supérieure k sur le nombre de fautes est connue. Ainsi, un algorithme k-stabilisant garantit la convergence vers une configuration légitime seulement à partir d'une configuration obtenue après l'application d'au plus k fautes (transitoires) sur une configuration légitime.

Pour modéliser les fautes, on utilise la *distance de Hamming* entre les configurations (Dolev et Herman, 1997) : il s'agit du nombre de nœuds dont les états locaux sont différents entre deux configurations γ et γ'. Le nombre de fautes dans une configuration est alors égale à la distance (de Hamming) minimale entre la configuration et une configuration légitime. Notons que le nombre maximal de fautes dans une configuration est égal au nombre de nœuds, n. Par conséquent, l'autostabilisation est un cas particulier de la k-stabilisation où $k = n$.

Comme illustré dans la figure 5, on peut ainsi définir formellement la k-stabilisation avec les distances de Hamming : la convergence est seulement assurée à partir des configurations qui sont à une distance inférieure ou égale à k d'une configuration légitime.

5.2. *État de l'art*

L'utilisation de solutions k-stabilisantes a été motivée par deux aspects principaux :

– certains algorithmes k-stabilisants convergent plus vite que leurs équivalents autostabilisants (bien sûr lorsque le nombre de fautes est inférieur ou égal à k) ;

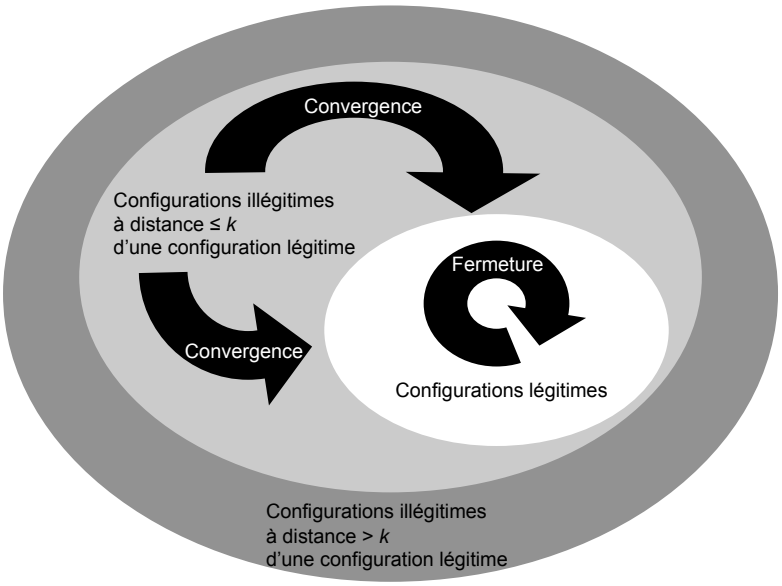

Figure 5. *k-stabilisation*

– la k-stabilisation permet de résoudre des problèmes n'ayant pas de solution autostabilisante.

Comme suggéré dans plusieurs articles, par exemple (Ghosh *et al.*, 2007), le temps de convergence de la plupart des algorithmes autostabilisants ne dépend pas du nombre de fautes. Ainsi, un algorithme autostabilisant peut dans certain cas converger à partir d'une simple faute en un temps dépendant de la taille du réseau. C'est le cas, par exemple avec le premier algorithme de circulation de jeton proposé dans (Dijkstra, 1974) : une seule faute peut provoquer une convergence en n rondes[10] où n est la taille du réseau. Ce type de comportement compromet l'utilisation de ces solutions dans un réseau à grande-échelle. (Beauquier *et al.*, 1998; Beauquier *et al.*, 1999) utilisent la k-stabilisation pour contourner ce problème. Ils proposent des solutions k-stabilisantes pour la circulation de jeton dans un anneau enraciné orienté dont le temps de convergence en rondes est fonction du nombre de fautes. Ce résultat est obtenu en supposant que $k < \sqrt{n}-2$ (où n est la taille de l'anneau) et que le démon est séquentiel (à chaque étape de calcul, un seul processeur exécute une action).

10. Une ronde est une unité de mesure de complexité en temps. Informellement, à partir d'une configuration donnée γ, une ronde correspond au temps nécessaire aux processeurs activables les plus lents pour qu'ils finissent par agir ou que leur environnement ne leur permette plus de le faire.

(Genolini et Tixeuil, 2002) considèrent la k-stabilisation de problèmes dynamiques, comme la circulation de jeton, dans le modèle à états avec un démon distribué[11]. Ils démontrent l'existence d'une borne inférieure sur le nombre total d'actions devant être exécutées pour converger de l'ordre du diamètre du réseau. Cette borne est vérifiée indépendamment du nombre de fautes. Cependant, ce résultat ne remet pas en cause celui de (Beauquier *et al.*, 1998; Beauquier *et al.*, 1999) où la complexité est évaluée en nombre de rondes. En fait, il montre plutôt que l'on peut obtenir un temps de convergence proportionnel au nombre de fautes seulement si on mesure la complexité en termes de rondes ou si le système est synchrone.

(Genolini et Tixeuil, 2001) prouvent que la k-stabilisation permet de résoudre plus de problèmes que l'autostabilisation. En effet, ils proposent un algorithme k-stabilisant de circulation de jeton dans un anneau unidirectionnel uniforme fonctionnant sous l'hypothèse d'un démon synchrone. Ce problème est connu pour n'avoir aucune solution autostabilisante : dans une configuration où il y a deux jetons, les deux jetons sont à égale distance l'un de l'autre et décrivent un axe de symétrie entre les états des processeurs, il est impossible de supprimer cette symétrie à cause du caractère synchrone du démon.

5.3. *Exemple d'algorithme*

A titre d'exemple, nous présentons maintenant un algorithme k-stabilisant de circulation de jeton dans un anneau orienté enraciné. Cet algorithme est proposé dans (Beauquier *et al.*, 1999). Il est écrit dans le modèle à états et considère un démon central (un seul processeur exécute une action à chaque étape de calcul).

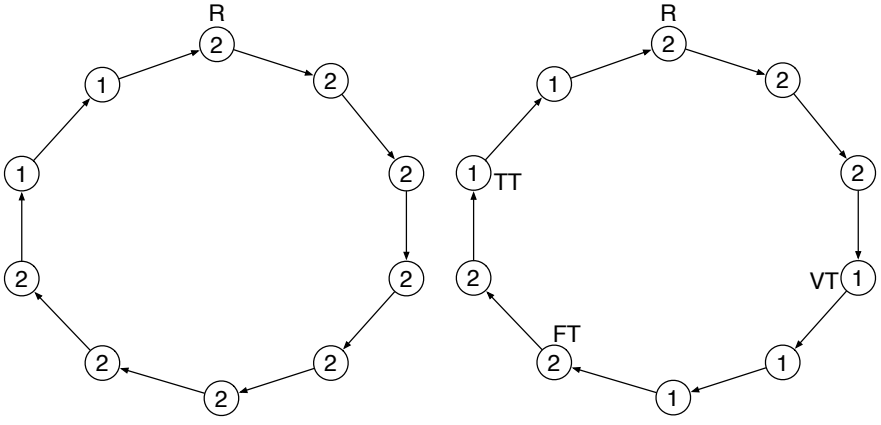

Figure 6. *Configurations de l'algorithme de Dijkstra*

11. Un démon distribué peut activer plusieurs processeurs à chaque étape de calcul.

Cet algorithme est basé sur le premier algorithme autostabilisant de (Dijkstra, 1974). Dans la figure 6, la partie gauche montre une configuration légitime et la partie droite une configuration illégitime de l'algorithme de Dijkstra. Dans cet algorithme, chaque processeur est muni d'une variable dont les valeurs varient de 0 à N où N est une borne supérieure sur la taille de l'anneau. La racine (R) détient le jeton lorsque sa variable est égale à celle de son prédécesseur. Au contraire, un processeur non racine détient un jeton lorsque la valeur de sa variable diffère de celle de son prédécesseur. Lorsqu'un processeur détient le jeton, il passe le jeton à son successeur en changeant son état : la racine incrémente sa variable (modulo $N + 1$) pour passer le jeton ; un processeur non racine recopie la valeur de son prédécesseur pour passer le jeton.

L'algorithme de (Beauquier *et al.*, 1999) part du constat suivant : après des fautes, l'anneau peut contenir 3 types de jetons (nb., toute configuration illégitime contient plusieurs jetons) :

– le *vrai* jeton (TT dans la figure),

– des jetons *virtuels* où un processeur corrompu a un prédécesseur non corrompu (VT dans la figure) et

– des *faux* jetons où un processeur a un prédécesseur corrompu (FT dans la figure).

L'idée principale de l'algorithme est alors la suivante : bloquer les faux jetons. Ainsi lorsqu'un faux jeton se fait « rattrapper » par un jeton virtuel ou le vrai jeton, le nombre global de jeton décroît. Pour ce faire, on modifie le code du protocole de Dijkstra pour qu'un processeur passe un jeton à son successeur seulement s'il détient un jeton et qu'une condition *TEST* est vérifiée. La valeur de la condition *TEST* dépend d'un mécanisme d'interrogation. Ce mécanisme permet d'évaluer si un processeur qui détient un jeton a un de ses $k + 1$ prédécesseurs qui lui aussi détient un jeton. Dans ce cas, la condition *TEST* est vraie et le processeur ne peut pas passer le jeton. Dans le cas contraire, la condition est fausse et le processeur peut passer le jeton. Il faut alors noter que cet algorithme n'est pas autostabilisant car il ne fonctionne que si le nombre de fautes k est strictement inférieur à $\sqrt{n} - 2$. En effet, dans le cas contraire, il existe des configurations où chaque processeur détenant un jeton a au moins un de ces $k+1$ prédécesseurs qui détient aussi un jeton et le système est en famine. Lorsque $k < \sqrt{n} - 2$, il existe au moins un jeton virtuel qui peut avancer. Lorsqu'un jeton virtuel avance, le nombre de processeurs corrompus diminue, d'où la convergence du système.

6. Conclusion

Dans cet article, nous avons présenté des avantages et inconvénients de l'autostabilisation comme approche à la tolérance aux fautes dans les systèmes distribués. Son principal atout est de tolérer toutes les formes de fautes pourvu qu'elles soient transitoires, malheureusement au prix de ne pas pouvoir escamoter certains effets indésirables occasionnés par ces fautes. Plusieurs techniques ont émané de cette approche

pour tenter d'éviter ses faiblesses et ses inconvénients. Certaines de ces techniques sont des *généralisations*, recherchant des propriétés plus faibles pour résoudre des problèmes n'ayant pas de solutions autostabilisantes ou pour obtenir des solutions moins coûteuses.

Parmi les généralisations proposées dans la littérature, nous avons décrit la pseudostabilisation, la stabilisation probabiliste, la stabilisation faible, et la k-stabilisation. Cette liste n'est pas exhaustive. En effet, d'autres approches de généralisation ont été proposées très récemment. Par exemple, dans (Lin *et al.*, 2009), les auteurs introduisent la notion de « quasi stabilisation » qui impose que le compteur ordinal de chaque processeur soit égal à zéro dans la configuration initiale. (Sudo *et al.*, 2010) introduisent les algorithmes « vaguement stabilisants » qui relâchent la notion de fermeture en autorisant que la sûreté soit perdue après un certain temps arbitrairement long. Il s'agit d'une notion plus générale que la pseudostabilisation car elle n'impose pas que le système finisse par ne plus dévier de sa spécification. Remarquons que nous n'avons pas décrit les spécialisations de l'autostabilisation, lesquelles sont traitées dans (Devismes *et al.*, 2009).

7. Bibliographie

Afek Y., Brown G. M., « Self-Stabilization Over Unreliable Communication Media », *Distributed Computing*, vol. 7, n° 1, p. 27-34, 1993.

Angluin D., « Local and Global Properties in Networks of Processors (Extended Abstract) », *STOC*, p. 82-93, 1980.

Beauquier J., Genolini C., Kutten S., « *k*-Stabilization of Reactive Tasks », *PODC*, p. 318, 1998.

Beauquier J., Genolini C., Kutten S., « Optimal Reactive *k*-Stabilization : The Case of Mutual Exclusion », *PODC*, p. 209-218, 1999.

Bernard S., Devismes S., Potop-Butucaru M. G., Tixeuil S., « Optimal Deterministic Self-stabilizing Vertex Coloring in Unidirectional Anonymous Networks », *Proceedings of the IEEE International Conference on Parallel and Distributed Processing Systems (IPDPS 2009)*, IEEE Press, Rome, Italy, p. 167-177, May, 2009.

Bruell S. C., Ghosh S., Karaata M. H., Pemmaraju S. V., « Self-Stabilizing Algorithms for Finding Centers and Medians of Trees », *SIAM J. Comput.*, vol. 29, n° 2, p. 600-614, 1999.

Burns J. E., Gouda M. G., Miller R. E., « Stabilization and Pseudo-Stabilization », *Distributed Computing*, vol. 7, n° 1, p. 35-42, 1993.

Cohen J., Dasgupta A., Ghosh S., Tixeuil S., « An exercise in selfish stabilization », *ACM Trans. Auton. Adapt. Syst.*, vol. 3, p. 15 :1-15 :12, December, 2008.

Cournier A., Graphe et algorithmique distribuée stabilisante, PhD thesis, Université de Picardie, Faculté de Math-Info, 33, Rue Saint Leu, 80 039 Amiens Cedex 1, décembre, 2009.

Delporte-Gallet C., Devismes S., Fauconnier H., « Stabilizing leader election in partial synchronous systems with crash failures », *J. Parallel Distrib. Comput.*, vol. 70, n° 1, p. 45-58, 2010.

Devismes S., Petit F., Villain V., « Autour de l'autostabilisation : Techniques spécialisant l'approche », *TSI*, 2009. Au sommaire de ce même numéro.

Devismes S., Tixeuil S., Yamashita M., « Weak vs. Self vs. Probabilistic Stabilization », *ICDCS*, p. 681-688, 2008.

Dijkstra E. W., « Self-stabilizing Systems in Spite of Distributed Control », *Commun. ACM*, vol. 17, n° 11, p. 643-644, 1974.

Dolev S., *Self-stabilization*, MIT Press, March, 2000.

Dolev S., Herman T., « Superstabilizing Protocols for Dynamic Distributed Systems », *Chicago J. Theor. Comput. Sci.*, 1997.

Genolini C., Tixeuil S., Reactive k-stabilization and time adaptivity : possibility and impossibility results, Technical Report n° 1276, Laboratoire de Recherche en Informatique, University of Paris Sud XI, 2001.

Genolini C., Tixeuil S., « A Lower Bound on Dynamic k-Stabilization in Asynchronous Systems », *SRDS*, p. 212-221, 2002.

Ghosh S., Gupta A., Herman T., Pemmaraju S. V., « Fault-containing self-stabilizing distributed protocols », *Distributed Computing*, vol. 20, n° 1, p. 53-73, 2007.

Gouda M. G., « The Theory of Weak Stabilization », *WSS*, p. 114-123, 2001.

Gouda M. G., Multari N. J., « Stabilizing Communication Protocols », *IEEE Trans. Computers*, vol. 40, n° 4, p. 448-458, 1991.

Gradinariu M., Tixeuil S., « Self-stabilizing Vertex Coloration and Arbitrary Graphs », *OPODIS*, p. 55-70, 2000.

Herman T., « Probabilistic Self-Stabilization », *Inf. Process. Lett.*, vol. 35, n° 2, p. 63-67, 1990.

Herman T., « Self-Stabilization : Randomness to Reduce Space », *Distributed Computing*, vol. 6, n° 2, p. 95-98, 1992.

Israeli A., Jalfon M., « Self-Stabilizing Ring Orientation », *WDAG*, p. 1-14, 1990a.

Israeli A., Jalfon M., « Token Management Schemes and Random Walks Yield Self-Stabilizing Mutual Exclusion », *PODC*, p. 119-131, 1990b.

Katz S., Perry K. J., « Self-Stabilizing Extensions for Message-Passing Systems », *Distributed Computing*, vol. 7, n° 1, p. 17-26, 1993.

Lin J.-C., Huang T. C., Yang C.-Z., Mou N., « Quasi-self-stabilization of a distributed system assuming read/write atomicity », *Comput. Math. Appl.*, vol. 57, n° 2, p. 184-194, 2009.

Shukla S., Rosenkrantz D., Ravi S., « Observations on self-stabilizing graph algorithms for anonymous networks », *Proceedings of the Second Workshop on Self-Stabilizing Systems*, p. 7.1-7.15, 1995.

Sudo Y., Nakamura J., Yamauchi Y., Ooshita F., Kakugawa H., Masuzawa T., « Loosely-stabilizing Leader Election in Population Protocol Model », *16th International Colloquium on Structural Information and Communication Complexity (SIROCCO 2009)*, vol. 5869 of *Lecture Notes in Computer Science*, p. 295-308, 2010.

Tel G., *Introduction to distributed algorithms*, Cambridge University Press, Cambridge, UK, 2001.

Tixeuil S., *Algorithms and Theory of Computation Handbook, Second Edition*, Chapman & Hall/CRC Applied Algorithms and Data Structures, CRC Press, Taylor & Francis Group, chapter Self-stabilizing Algorithms, p. 26.1-26.45, 2009.

Article reçu le 16 juin 2009
Accepté après révisions le 30 juin 2010

Stéphane Devismes *est maître de conférences à l'université Joseph Fourier de Grenoble (Grenoble I) et membre de l'équipe Synchrone du laboratoire VERIMAG de Grenoble. Ses activités de recherche couvrent les différents aspects de l'algorithmique distribuée et notamment, les problématiques liées à la tolérance aux fautes, en particulier l'autostabilisation.*

Franck Petit *est professeur à l'université Pierre et Marie Curie. Il effectue ses recherches au sein du Laboratoire d'Informatique de Paris 6 (LIP6). Celles-ci portent principalement sur l'algorithmique distribuée, notamment la tolérance aux fautes, l'autostabilisation, la stabilisation instantanée et les cohortes de robots mobiles.*

Vincent Villain *est professeur à l'université de Picardie Jules Verne. Il est responsable de l'équipe SDMA (Systèmes Distribués, Mots et Applications) au sein du laboratoire MIS (Modélisation, Information et Systèmes). Ses travaux portent essentiellement sur l'algorithmique distribuée et plus particulièrement sur la tolérance aux fautes, l'autostabilisation et la stabilisation instantanée.*

ANNEXE POUR LE SERVICE FABRICATION
A FOURNIR PAR LES AUTEURS AVEC UN EXEMPLAIRE PAPIER
DE LEUR ARTICLE ET LE COPYRIGHT SIGNE PAR COURRIER
LE FICHIER PDF CORRESPONDANT SERA ENVOYE PAR E-MAIL

1. ARTICLE POUR LA REVUE :

RSTI - TSI – 30/2011. Algorithmique distribuée

2. AUTEURS :

Stéphane Devismes[*] *— Franck Petit*[**] *— Vincent Villain*[***]

3. TITRE DE L'ARTICLE :

Autour de l'autostabilisation

4. TITRE ABRÉGÉ POUR LE HAUT DE PAGE MOINS DE 40 SIGNES :

Généralisation de l'autostabilisation

5. DATE DE CETTE VERSION :

23 juillet 2012

6. COORDONNÉES DES AUTEURS :

– adresse postale :
[*] VERIMAG, Université Joseph Fourier, Grenoble
[**] LiP6, Université Pierre et Marie Curie Paris 6, Paris
[***] MIS, Université de Picardie Jules Verne, Amiens
– téléphone : 04 56 52 03 68
– télécopie : 04 56 52 03 44
– e-mail : stephane.devismes@imag.fr

7. LOGICIEL UTILISÉ POUR LA PRÉPARATION DE CET ARTICLE :

LaTeX, avec le fichier de style `article-hermes2.cls`,
version 1.23 du 17/11/2005.

8. FORMULAIRE DE COPYRIGHT :

Retourner le formulaire de copyright signé par les auteurs, téléchargé sur :
`http://www.revuesonline.com`

SERVICE ÉDITORIAL – HERMES-LAVOISIER
14 rue de Provigny, F-94236 Cachan cedex
Tél. : 01-47-40-67-67
E-mail : revues@lavoisier.fr
Serveur web : http://www.revuesonline.com