Autour de l'autostabilisation

Partie II : Techniques spécialisant 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 seconde partie d'un survol des techniques dérivées de l'autostabilisation, la première étant également au sommaire de ce numéro (Devismes* et al.*, 2011). 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 à ces inconvénients conduisant, les unes, à affaiblir l'autostabilisation, les autres au contraire à la renforcer. Différentes généralisations sont présentées dans la première partie (Devismes* et al.*, 2011). Cette seconde partie traite des spécialisations de l'autostabilisation. Il s'agit de renforcer certaines propriétés dans le but d'accroître le spectre des aléas susceptibles d'être tolérés par le système ou encore de mieux le traiter. Nous nous intéressons ici aux propriétés dont la finalité est d'améliorer la dynamicité, de tolérer plus de fautes ou d'affermir la propriété de convergence.*

ABSTRACT. *This paper presents the second part of a survey on techniques derived from self-stabilization, the first being also published in this issue (Devismes* et al.*, 2011). 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. This second part deals with the specialization that seeks to enhance properties in order to increase the range of hazards the system can tolerate or to enhance its processing. We are interested here in properties whose purpose is to enhance the dynamic, to tolerate more failure patterns, or to consolidate the convergence property of self-stabilization.*

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 895 à 922

1. Introduction

Dans cet article, nous proposons la seconde partie d'un survol des techniques dérivées de l'autostabilisation. Dans la première partie (Devismes *et al.*, 2011), nous avons présenté le concept 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 et malheureusement au prix de ne pas pouvoir escamoter certains effets indésirables occasionnés par ces fautes. Deux types d'approches sont envisagées pour contourner cet inconvénient. La première, consiste à *généraliser* le concept de l'autostabilisation en *affaiblissant* certaines contraintes. La seconde vise à *spécialiser* le concept en *renforçant* certaines propriétés dans le but d'accroître le spectre des aléas susceptibles être tolérés par le système ou encore de mieux le traiter.

Certaines généralisations sont présentées dans la première partie (Devismes *et al.*, 2011). Dans cette seconde partie, nous traitons des spécialisations. Plus précisément, nous nous intéressons aux propriétés dont la finalité est d'améliorer la dynamicité, de tolérer plus de fautes ou encore d'affermir la propriété de convergence. Nous en présentons certaines, notamment la *superstabilisation* (section 2), le *confinement de fautes* (section 3), la *stabilisation adaptative en temps* (section 4), l'*autostabilisation robuste* (section 5), la *stabilisation sûre* (section 6) et la *stabilisation instantanée* (section 7). Cette dernière est présentée de manière différente car elle constitue une rupture par rapport aux autres approches considérées : alors que celles-ci proposent des solutions algorithmiques à un problème défini au préalable, celle-là part du problème pour en donner une spécification adaptée à l'environnement (configuration initiale quelconque) et seulement ensuite, proposer un algorithme.

2. Superstabilisation

2.1. *Définition*

A l'instar de la stabilisation adaptative en temps et du confinement de fautes (cf. sections 3 et 4), l'introduction de la *superstabilisation* a été motivée par la recherche de solutions efficaces lorsque le nombre de fautes est faible. En effet, la plupart des algorithmes autostabilisants ne permettent pas de maîtriser l'étendue des effets d'une seule faute, la défaillance d'un seul processeur pouvant impacter l'ensemble du système. La superstabilisation, introduite dans (Dolev et Herman, 1997), s'applique aux *systèmes dynamiques*, c'est-à-dire aux systèmes dans lesquels des changements topologiques peuvent survenir par l'*ajout* ou la *suppression* de processeurs ou de liens de communication[1]. Elle vise à circonscrire l'impact d'un changement topologique au voisinage de ce dernier. En d'autres termes, un algorithme est superstabilisant dès lors qu'il est autostabilisant et qu'en partant d'une configuration correcte à partir de laquelle un certain type de changement topologique intervient, le système retrouve

1. (Dolev et Herman, 1997) supposent qu'un seul changement topologique intervient à la fois.

« rapidement » une configuration correcte tout en garantissant qu'un prédicat, dit prédicat de *passage*, est préservé.

Le prédicat de passage est défini en fonction du type de changements topologiques. *A priori*, il doit permettre de pouvoir relâcher certaines conditions de bon fonctionnement de l'algorithme tout en étant suffisamment fort pour apporter un quelconque avantage.

Par exemple, dans un système dans lequel on cherche à désigner un processeur particulier, appelé leader, le prédicat de passage peut consister à garantir qu'à tout moment, le système contient au plus un leader. Pour qu'un algorithme conçu pour élire un leader soit superstabilisant, il doit d'abord garantir qu'en partant d'une configuration contenant plusieurs leaders, le système finit par ne plus contenir qu'un leader et en l'absence de changement topologique (et de faute transitoire), cette situation perdure pour toujours ; il doit également assurer qu'en cas de changement topologique alors qu'il n'existe qu'un seul leader, aucun nouveau leader ne peut être désigné, sauf si le système n'en contient aucun.

Outre les paramètres classiques de mesure de performance comme le temps de stabilisation, l'efficacité d'un algorithme superstabilisant peut se mesurer en « temps de superstabilisation » qui correspond au temps mis par l'algorithme pour ramener le système dans un état correct après qu'un changement topologique soit intervenu depuis une configuration stable.

2.2. *État de l'art*

Deux exemples d'algorithmes superstabilisants sont présentés dans (Dolev et Herman, 1997). Le premier résout le problème du coloriage et le second permet de construire un arbre couvrant. Il inclut également un schéma général pour transformer un algorithme autostabilisant $\mathcal{A}$ en son équivalent superstabilisant. Ce transformateur utilise un composant appelé « superstabilisateur » qui est adjoint à $\mathcal{A}$ et qui consiste principalement à spécifier une section d'exception devant être exécutée lors d'un changement topologique. Cette dernière est évidemment propre à $\mathcal{A}$.

(Herman, 2000) s'intéresse au problème de l'exclusion mutuelle dans un anneau. Il introduit un autre point de vue de la superstabilisation en ne traitant plus des changements topologiques, mais de fautes transitoires localisées qu'il définit comme étant n'importe quelle défaillance transitoire subie par *au plus* un nœud du système. De sorte que dans ce cadre, un protocole est superstabilisant si d'une part, il est autostabilisant lorsque qu'une faute transitoire touche plus d'un processeur et d'autre part, il vérifie le prédicat de passage lorsque la faute transitoire ne touche qu'un processeur. Cette approche se rapproche fortement du *confinement de fautes* discuté dans la section 3.

Les algorithmes d'exclusion mutuelle auxquels Herman s'intéresse sont ceux fondés sur une circulation de jeton. Le prédicat de passage spécifie que le système contient

au plus un jeton, sauf si un des jetons est détenu par le processeur subissant la faute transitoire. A partir de cette définition, il précise certaines conditions nécessaires à l'implantation d'un algorithme de circulation de jeton superstabilisant et donne les algorithmes *ad hoc*. Les travaux initiés par Herman sont poursuivis dans (Katayama *et al.*, 2002). Ses auteurs étudient la latence, une des mesures d'efficacité introduites par Herman pour la circulation de jeton. La latence permet de mesurer le surcoût de la stabilisation pour un algorithme de circulation de jeton. En effet, un algorithme de circulation de jeton est k-latent si à partir de toute configuration légitime, le jeton est retardé d'un facteur k durant chaque circulation complète. (Katayama *et al.*, 2002) démontrent une borne minimale pour laquelle il est possible d'envisager un algorithme superstabilisant de circulation de jeton et donne un algorithme optimal pour cette borne.

2.3. *Exemple d'algorithme*

Nous considérons le problème du *coloriage* qui consiste à faire en sorte que chaque processeur (c'est-à-dire chaque nœud du graphe) finisse par choisir une couleur une fois pour toutes sans que deux nœuds voisins aient la même couleur. L'algorithme que nous présentons s'inspire de (Dolev et Herman, 1997). Il est écrit dans *le modèle à états* dont les principales caractéristiques sont que (i) les processeurs communiquent par le biais de variables localement partagées, (ii) leur programme est constitué d'un ensemble d'actions gardées, (iii) exécutées atomiquement lorsque leur garde est vraie et que le processeur est activé par le démon, ce dernier modélisant le degré de synchronisme du système[2].

L'algorithme fonctionne sous l'hypothèse la plus faible du modèle à états : *le démon distribué inéquitable* pour lequel le choix n'est pas contraint, à l'exception du fait qu'il doit activer au moins un nœud à chaque étape tant qu'il existe des nœuds activables.

Chaque nœud dispose d'un *identifiant incorruptible unique* et d'une variable *couleur*. Le principe de base de l'algorithme est le suivant : lorsqu'un nœud détecte qu'il a une couleur identique à celle d'un de ses voisins ayant un identifiant plus grand que le sien, alors il choisit arbitrairement (par exemple, la plus grande) une nouvelle couleur dans $\{1 \ldots \Delta + 1\}$ qui n'a été choisie par aucun de ses voisins. Le fait que le processeur choisisse sa couleur parmi $\Delta + 1$ valeurs nous assure qu'il existe toujours une couleur disponible. De plus, l'ordre total induit par les identifiants rend toute symétrie impossible en interdisant que deux voisins monochromes choisissent une couleur en même temps. Il est facile de constater que le temps de convergence de cet algorithme est de l'ordre de n rondes[3].

---

2. Le lecteur pourra trouver une description plus détaillée du modèle à état en section 3.3.1 de la première partie de ce survol (Devismes *et al.*, 2011).

3. 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

Le système auquel nous nous intéressons étant dynamique, des liens de communication ou des nœuds peuvent être ajoutés ou supprimés, sans jamais néanmoins remettre en cause le fait que les nœuds aient au plus Δ voisins. Supposons que les changements de topologie soient restreints à l'ajout ou au retrait d'un seul lien de communication à la fois. Clairement, la suppression d'un lien n'a pas de conséquence sur la stabilité du système (même si sa connexité n'est pas préservée). Par contre, en partant d'une situation stable, l'ajout d'un lien peut contrarier la spécification lorsque ses deux extrémités sont de même couleur.

Supposons que chaque processeur p soit doté d'un dispositif d'interruption permettant de détecter s'il est lui-même ajouté au système – interruption $\texttt{recov}_p$ –, ou si un lien de communication lui est ajouté – interruption $\texttt{recov}_{pq}$. Notons qu'une interruption $\texttt{recov}_p$ ou $\texttt{recov}_{pq}$ déclenche systématiquement l'interruption $\texttt{recov}_{qp}$ sur tout q affecté par l'ajout de p ou du lien pq (dans ce dernier cas, q est évidemment l'unique processeur partageant le lien pq avec p). Notons aussi que les interruptions sont déclenchées simultanément avec leur changement topologique associé. Lorsque l'interruption $\texttt{recov}_p$ est déclenchée, p (ré-)initialise sa couleur avec une couleur spécifique, $\perp$. Il fait de même lorsqu'il s'agit d'une interruption $\texttt{recov}_{pq}$ et que son identifiant est plus petit que celui de q. Ainsi, si un changement topologique se produit alors que le système était stable, le prédicat de passage suivant est vérifié : deux nœuds voisins ont la même couleur si et seulement si leur couleur est $\perp$. Ensuite, pour revenir à une situation stable, chaque processeur de couleur $\perp$ change sa couleur pour une couleur dans $\{1 \dots \Delta + 1\}$ qui n'a été choisie par aucun de ses voisins. Un tel processeur agit uniquement lorsqu'il est le processeur de couleur $\perp$ avec l'identité la plus grande dans son voisinage. L'algorithme garantit alors que le temps de superstabilisation est d'au plus k rondes, k étant le nombre de processeurs affectés par le changement topologique ($k \leq \Delta$). C'est-à-dire, si à partir d'une situation stable, le système subit un unique changement topologique, alors en au plus k rondes le système retrouve une configuration où les processeurs sont coloriés correctement avec au plus $\Delta + 1$ couleurs.

3. Confinement de fautes

3.1. *Définition*

L'approche du *confinement de fautes* rejoint celles de la superstabilisation et de la stabilisation adaptative en temps (cf. sections 2 et 4). Elle s'applique à nouveau lorsque le nombre de fautes est réduit.

Introduite dans (Ghosh et He, 2000), son but est d'éviter la propagation de fautes localisées au-delà d'une certaine limite. Par « fautes localisées », les auteurs entendent des fautes transitoires qui affectent un petit nombre de composants physiquement

---

les plus lents pour qu'ils finissent par agir ou que leur environnement ne leur permette plus de le faire.

proches les uns des autres dans le système. Lorsqu'une faute localisée se produit dans le réseau, le nombre d'actions directement imputables au rétablissement du système doit être petit, de même que le nombre de composants affectés par les effets de la faute, ces derniers devant en outre être physiquement proches des composants défaillants. La limite à laquelle la propagation d'une faute localisée doit être stoppée est donc à la fois temporelle et spaciale. On parle respectivement de *temps* et de *rayon de confinement* – cf. figure 1.

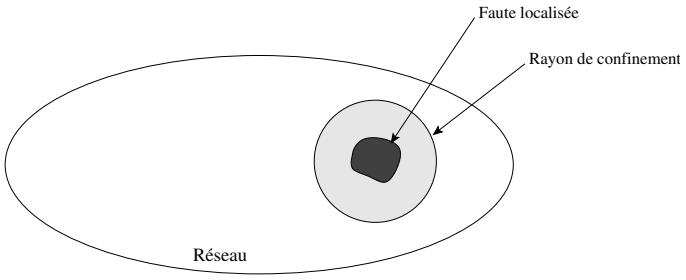


Figure 1. *Confinement de fautes localisées*

(Ghosh et He, 2000) ne traitent que du cas où une seule faute localisée intervient, elle n'affecte qu'un seul processeur. Ils parlent alors d'état 1-défaillant et affirment qu'un algorithme est un algorithme de confinement de fautes dès lors qu'il est autostabilisant et qu'en partant d'une configuration 1-défaillante, les temps et rayon de confinement sont de l'ordre de 1.

3.2. *État de l'art*

Le principal résultat de (Ghosh et He, 2000; Ghosh *et al.*, 2007) a trait aux algorithmes silencieux, c'est-à-dire des algorithmes qui atteignent en temps fini une configuration dans laquelle les valeurs des variables ne changent plus. Il s'agit d'un transformateur capable de rendre tout algorithme silencieux et autostabilisant en son équivalent confinant les défaillances. Deux algorithmes *ad hoc* pour les problèmes de l'élection de leader et de la construction d'arbre couvrant sont respectivement proposés dans (Ghosh et Gupta, 1996) et (Ghosh et He, 2000).

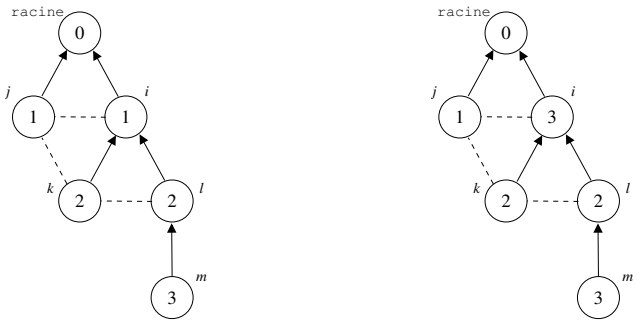
Une approche originale est présentée dans (Herman et Pemmaraju, 2000). Elle utilise un code de détection d'erreurs pour identifier avec une forte probabilité les corruptions de données. Les auteurs caractérisent la classe d'algorithmes autostabilisants pouvant utiliser un code de détection d'erreurs pour pouvoir confiner une faute avec une forte probabilité. La classe inclut tous les algorithmes silencieux et un certain type d'algorithmes non silencieux, notamment ceux pour lesquelles la condition suivante est vraie : pour toute configuration correcte γ, il existe une configuration γ' atteignable depuis γ en une étape telle que tous les processeurs sont bloqués dans γ'.

3.3. *Exemple d'algorithme*

Nous présentons l'algorithme de construction d'arbre couvrant de (Ghosh et He, 2000). Il se fonde sur l'algorithme autostabilisant de (Chen *et al.*, 1991) écrit dans le modèle à états pour un réseau contenant un processeur particulier r, appelé la racine. L'algorithme de (Chen *et al.*, 1991) utilise deux variables par nœud $p \neq r$: la première est un pointeur dénommé $\mathtt{P_p}$ désignant le voisin de p qui normalement est son père dans l'arbre ; la seconde, appelée N_p (comme « niveau ») mesure la distance séparant p de r dans l'arbre. Quant à la racine r, elle n'exécute pas d'algorithme et ne dispose pas de pointeur P mais d'une contante N, égale à 0. Lorsque l'arbre couvrant est formé, le niveau de chaque processeur $p \neq r$ doit être compris entre 1 et $n-1$ (n étant égal au nombre de processeurs du système) et être égal au niveau de son père incrémenté de 1.

Chaque processeur $p \neq r$ connaît n. L'algorithme stabilise de la manière suivante : si N_p est différent de $N_{P_p}+1$ et de n, alors N_p reçoit $N_{P_p}+1$, sauf si $N_{P_p} = n$ auquel cas N_p reçoit n. Lorsque $N_p = n$ et qu'il existe un voisin q tel que $N_q < n-1$, N_p et P_p sont respectivement affectés des valeurs $N_q + 1$ et q. En procédant ainsi, pour chaque chemin constitué à partir des pointeurs qui ne se terminent pas à la racine, au moins un processeur finit par avoir son niveau égal à n. Parallèlement à cela, un arbre couvrant se construit de proche en proche autour de la racine r, arbre qui finit par recouvrir l'ensemble du graphe par l'application des règles énoncées ci-avant.

La figure 2(a) montre un exemple de configuration terminale obtenu avec l'algorithme de (Chen *et al.*, 1991) – chaque flèche dirigée de p vers q représente P_p, la valeur de N_p étant l'entier inscrite à l'intérieur de chaque nœud p.



(a) Une configuration terminale correcte (b) Une configuration 1-défaillante

Figure 2. *Un exemple traitant de la construction d'un arbre couvrant*

Sur la figure 2(b), le processeur i a subit une défaillance. Il est facile de constater qu'en corrigeant la distance de i de 3 à 1, aucun autre nœud voisin de i n'a le temps d'être contaminé et que le dysfonctionnement est corrigé en $O(1)$ étape.

Malheureusement, l'algorithme de (Chen *et al.*, 1991) permet d'autres corrections en parallèle, par exemple en permettant à k et à l de « corriger » leur distance par rapport à leur père (i) en la changeant de 2 à 4. On voit bien que si k et l agissent avant ou en même temps que i, le processus peut continuer ainsi, jusqu'à contaminer l'ensemble du réseau dans le pire des cas.

L'idée de Ghosh et He consiste donc à faire en sorte que chaque processeur vérifie si les effets de la faute ont une chance d'être corrigés localement avant d'initier la stabilisation complète du système avec les règles de corrections de l'algorithme de (Chen *et al.*, 1991). Pour cela, lorsqu'un processeur p constate que son niveau n'est pas correct vis-à-vis de son voisinage, il commence par vérifier les deux conditions suivantes :

1) le niveau de tous ses voisins est strictement inférieur à n ;

2) il existe une valeur k telle que (i) le niveau des voisins de p pointant vers p (les fils de p, s'il existent) est égal à $k-1$ et (ii) il existe un autre voisin de p (n'étant pas un de ses fils) ayant son niveau égal à $k+1$ et (iii) soit $N_p \neq k$, soit $N_p \neq N_{P_p} + 1$ (le niveau de p est différent du niveau de son père incrémenté de 1).

Lorsqu'elles sont toutes deux vérifiées, alors p peut corriger son niveau en lui affectant la valeur k, après avoir le cas échéant changé de père en pointant sur le voisin ayant son niveau égal à $k-1$.

Cependant, cela n'empêche toujours pas les fils de p d'appliquer la correction conduisant à stabiliser le système dans son entier. Il reste donc à s'assurer que p exécute sa correction avant que ses fils ne puissent appliquer la leur. Ghosh et He parviennent à cela en surimposant un mécanisme de question/réponse entre voisins. Ce mécanisme est mis en œuvre lorsqu'un processeur doit exécuter une des règles de l'algorithme de (Chen *et al.*, 1991) : avant d'exécuter la règle, le processeur averti l'ensemble de ses voisins. Si l'un d'entre eux a la possiblité d'effectuer la correction locale telle qu'elle est décrite ci-avant, il l'effectue avant de répondre au demandeur. Le demandeur n'exécute la règle qu'il souhaitait exécuter que s'il reçoit une réponse positive de tous ses voisins.

Ce subtile mécanisme de gestion de priorités garantit la non prolifération d'une faute et sa correction en $O(1)$ étape. Mis en œuvre avec trois variables supplémentaires, il n'accroît pas la complexité en espace des processeurs ($O(\log n)$).

4. Autostabilisation adaptable en temps

4.1. *Définition*

Le but de l'*autostabilisation adaptable en temps* (Kutten et Patt-Shamir, 1997; Kutten et Patt-Shamir, 1999) est d'obtenir une convergence rapide lorsque le système subit peu de fautes. Elle a été proposée pour une classe particulière de problèmes : les tâches *non réactives*. Une tâche non réactive est un calcul réparti dont le résultat

est fonction de la donnée en entrée. Dans un algorithme résolvant une tâche non réactive, on distingue trois types de variables : les variables d'entrée (qui contiennent les données d'entrée), les variables de sortie (qui contiennent le résultat) et les variables internes (qui sont utilisées pour le calcul uniquement). On appelle *configuration d'entrée* (respectivement, *configuration de sortie*) la projection d'une configuration sur les variables d'entrée (respectivement sur les variables de sortie). La *spécification* d'une tâche non réactive est alors une relation entre les configurations d'entrée et les configurations de sortie. Une configuration légitime est alors une configuration dont les configurations d'entrée et de sortie associées sont conformes à la spécification. Un algorithme non réactif doit aussi être *silencieux* (Dolev *et al.*, 1999) : l'algorithme silencieux atteint en temps fini une *configuration silencieuse* où les valeurs de toutes les variables des processeurs ne changent plus.

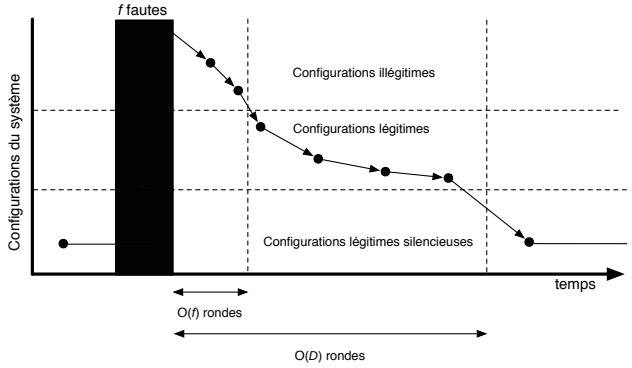


Figure 3. *Autostabilisation adaptable en temps*

L'*autostabilisation adaptable en temps* fait la distinction entre la convergence vers une configuration légitime et la convergence vers une configuration silencieuse. En effet, un algorithme autostabilisant k-adaptable en temps vérifie trois propriétés (cf., figure 3) :

1. il est autostabilisant ;

2. si la configuration initiale contient $f \leq k$ fautes alors l'algorithme converge vers une configuration légitime en $O(f)$ rondes, c'est-à-dire, en temps proportionnel au nombre de fautes ;

3. à partir de n'importe quelle configuration initiale, l'algorithme converge vers une configuration silencieuse en $O(D)$ rondes où D est le diamètre du réseau.

Ainsi, lorsque le système subit un faible nombre de fautes, un algorithme adaptable en temps retrouve rapidement un état légitime. Le prix de cette efficacité est un surcoût de calcul une fois retrouvée 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 confi-

guration est alors égal 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.

4.2. *État de l'art*

(Kutten et Patt-Shamir, 1999) proposent le premier algorithme autostabilisant k-adaptable en temps. Cet algorithme résout le problème du bit persistant dans un réseau identifié synchrone de topologie quelconque. Dans cet article, k, bien qu'inconnu des processeurs, est supposé strictement inférieur à $n/2$ où n est le nombre de processeurs, *i.e.*, l'algorithme est autostabilisant et la convergence rapide vers une configuration légitime n'est garantie que lorsque le nombre de fautes est strictement inférieur à $n/2$. La complexité en temps de cet algorithme est prouvée optimale. Dans le même article, les auteurs généralisent leur approche en proposant un algorithme qui transforme un algorithme non réactif ni autostabilisant ni adaptable en temps en algorithme autostabilisant k-adaptable en temps pour $k < n/2$.

Un autre algorithme autostabilisant et k-adaptable en temps de bit persistant pour réseaux identifiés synchrones de topologie quelconque est proposé dans (Herman, 1997). Cet algorithme est k-adaptable pour toute valeur de k.

L'algorithme autostabilisant k-adaptable en temps proposé dans (Burman *et al.*, 2005) résout un problème plus général que le bit persistant : *le consensus majoritaire*. Le modèle et les résultats obtenus sont comparables à ceux de (Kutten et Patt-Shamir, 1999).

Deux solutions générales (transformateurs) à l'autostabilisation adaptable en temps dans des réseaux asynchrones sont proposées dans (Dolev et Herman, 2007; Kutten et Patt-Shamir, 1998).

4.3. *Exemple d'algorithme*

Nous présentons maintenant un algorithme de *bit persistant* autostabilisant et adaptable en temps pour réseaux identifiés de topologie quelconque. Cet algorithme est écrit dans le modèle à états et suppose un démon synchrone. Il est inspiré de la solution proposée dans (Kutten et Patt-Shamir, 1999).

Dans le problème du bit persistant, chaque processeur possède un bit de « sortie » et on souhaite que le système converge vers une configuration où les bits de sortie de tous les processeurs aient la même valeur. Une fois que le système est dans une configuration légitime où tous les bits de sortie ont la même valeur $i \in \{0, 1\}$, l'algorithme doit garantir que si le système subit jusqu'à f fautes, le système converge à nouveau vers une configuration où tous les bits de sortie ont la valeur i. En outre, une décision triviale est interdite, c'est-à-dire, l'algorithme doit pouvoir décider 0 dans certains cas, et 1 dans d'autres cas.

Dans l'algorithme que nous étudions, f est supposé (strictement) inférieur à $n/2$ où n est le nombre de processeurs. Dans ce cas, il existe toujours une majorité de bits de sortie corrects juste après les fautes. L'idée est alors qu'un processeur change la valeur de son bit de sortie seulement s'il estime qu'il y a une majorité de processeurs qui ne sont pas d'accord avec lui. Chaque processeur stocke une estimation de la valeur du bit de sortie de chacun des autres processeurs pour prendre sa décision.

L'algorithme part du constat suivant : les processeurs corrompus doivent rapidement « réparer » leur bit de sortie mais il faut éviter la propagation de valeurs corrompues. Ces deux points sont contradictoires. En fait, pour aller vite, un processeur non corrompu peut se croire fautif et changer la valeur de son bit : dans ce cas, une erreur est propagée.

Pour résoudre ce problème, on introduit une nouvelle variable : un bit d'entrée. Dans une configuration légitime les bits d'entrée et de sortie sont tous égaux. C'est maintenant l'estimation des bits d'entrée de tous les processeurs qui définit la valeur du bit de sortie d'un processeur (en utilisant la règle de la majorité).

Après des fautes, ces deux bits peuvent être modifiés mais à des vitesses différentes : le bit de sortie peut changer de valeur rapidement afin de converger rapidement vers une configuration légitime ; au contraire, un bit d'entrée ne peut pas changer de valeur avant une période assez longue afin de ralentir la propagation des erreurs.

Le protocole est constitué de deux couches :

– la première suppose qu'un processeur non corrompu ne change jamais la valeur de son bit d'entrée et assure que les bits de sortie sont à nouveau corrects en $O(f)$ rondes ;

– la seconde assure qu'un processeur non corrompu ne change jamais la valeur de son bit d'entrée et corrige les erreurs dans les bits d'entrée en $O(D)$ rondes.

L'idée de la première couche est de propager la valeur de bit d'entrée de chaque processeur p le long d'un arbre couvrant en largeur enraciné en p. Chaque arbre couvrant est calculé à l'aide d'une variable de distance et d'un pointeur père (la méthode utilisée est celle de (Huang et Chen, 1992)).

Supposons qu'une faute affecte un des arbres couvrants. L'idée est alors de propager une valeur de reset ($\perp$) dans les variables dans le sous arbre du processeur corrompu. Cependant, avant qu'un processeur corrompu ne reçoive le reset, il peut propager l'erreur. Pour assurer que le reset « rattrape » l'erreur, on modifie l'algorithme afin qu'un processeur ne puisse changer ses variables pour une valeur autre que celle de reset ($\perp$) que tous les deux rondes. En $2 \times \min(d, f)$ rondes, les valeurs corrompues ont disparu où d est la hauteur de l'arbre. Les variables dont les valeurs sont égales à $\perp$ sont alors ignorées et sont à nouveau correctement assignées $\min(d, f)$ rondes plus tard (la valeur correcte est aussi propagée deux fois plus lentement que le reset).

En s'appuyant sur la première couche, on sait qu'un processeur p ne peut plus être atteint par une valeur corrompue après au plus $2 \times \min(dist_{max}(p), f)$ rondes où

$dist_{max}(p)$ est la distance maximale entre p et un autre processeur. Ensuite, comme le démon est synchrone, on sait aussi que p peut « compter » le temps. L'idée est alors la suivante : lorqu'une variable de la première couche change, p remet un compteur à zéro. Le compteur est ensuite incrémenté à chaque étape modulo $2 \times \min(D, f) + 2$, où D est le diamètre du réseau. Lorsqu'un compteur atteint $2 \times \min(D, f) + 1$, le processeur concerné affecte son bit d'entrée à la valeur de son bit de sortie. Ceci empêche un processeur correct p de corrompre la valeur de son bit d'entrée. En effet, un « mauvais » changement de bit de sortie provoque une remise à zéro du compteur et le bit de sortie est corrigé avant que le compteur de p n'atteigne la borne, *i.e.*, avant que p ne modifie son bit d'entrée. Cela permet aussi à un processeur corrompu de finir par corriger son bit d'entrée : le bit d'entrée est corrigé au plus $2 \times \min(D, f) + 1$ rondes après la correction du bit de sortie. Ainsi, en $O(D)$ rondes, les valeurs des bits d'entrée sont correctes : le système est stabilisé.

5. Autostabilisation robuste

5.1. *Définition*

Traditionnellement, les méthodes permettant de réaliser des protocoles tolérants aux pannes sont classés en deux catégories : les approches *masquantes* et *non masquantes*.

L'approche masquante consiste à concevoir des protocoles où les pannes n'ont pas d'effet sur la sûreté des services fournis par les protocoles. Pour ce faire, les processeurs suspectent toutes les informations qu'ils reçoivent et chaque étape de calcul est garantie par un nombre suffisant de contrôles préalables. Cette approche est généralement utilisée dans des réseaux où les pannes sont *définitives*, *e.g.*, arrêt définitif de processeurs (*crash*), perte de lien de communications, perte de messages... Dans ce type de système, les processeurs non défaillants sont supposés avoir des données et un comportement correct. Ainsi, cette approche devient inapplicable lorsque tout ou partie des composants (processeur ou lien de communications) du système peuvent subir des corruptions.

L'autostabilisation est une approche non masquante qui autorise tous les processeurs à avoir un comportement incorrect mais garantit le retour du système à un comportement normal en un temps fini après que les pannes ont cessé. L'autostabilisation est donc adaptée aux systèmes pouvant subir des pannes transitoires comme par exemple un nombre fini de corruptions mémoires. Cette approche a été longtemps utilisée uniquement dans le contexte des fautes transitoires. De ce fait, de nombreux algorithmes autostabilisants deviennent inopérants en cas de pannes franches comme par exemple l'arrêt définitif d'un seul processeur.

(Gopal et Perry, 1993) proposent une approche transversale : l'*autostabilisation robuste*[4]. Elle consiste simplement à concevoir des algorithmes autostabilisants pour des systèmes pouvant subir plusieurs autres types de pannes, comme par exemple, les arrêts définitifs de processeurs, les fautes intermittentes, ou encore les fautes byzantines (voire (Varghese et Jayaram, 2000) pour une taximonie détaillée des types de fautes).

5.2. *État de l'art*

La plupart des articles sur l'autostabilisation robuste traitent de l'autostabilisation dans des systèmes où les processeurs peuvent tomber définitivement en panne, *e.g.*, (Gopal et Perry, 1993; Anagnostou et Hadzilacos, 1993; Beauquier et Kekkonen-Moneta, 1997b; Beauquier et Kekkonen-Moneta, 1997a; Hutle et Widder, 2005b; Hutle et Widder, 2005a; Delporte-Gallet *et al.*, 2010). (Gopal et Perry, 1993) ont proposé un algorithme autostabilisant robuste de synchronisation de phases fonctionnant dans un réseau complet synchrone où au plus k processeurs peuvent s'arrêter définitivement (pour simplifier, on utilisera l'anglicisme *crash*). En utilisant cet algorithme, ils ont aussi proposé une méthode pour transformer des algorithmes tolérants aux pannes *crash* en algorithmes autostabilisants robustes. (Anagnostou et Hadzilacos, 1993) démontrent qu'il n'existe pas d'algorithme autostabilisant robuste permettant de calculer la taille d'un réseau en anneau si le réseau est asynchrone et peut subir au plus un crash. (Beauquier et Kekkonen-Moneta, 1997b) étendent ce résultat d'impossibilité à des réseaux de topologie plus complexe mais démontrent aussi que dans un réseau quasi synchrone, il devient possible de résoudre ce problème avec des détecteurs de pannes. Enfin, (Delporte-Gallet *et al.*, 2010) considèrent le problème de l'élection de leader autostabilisant dans un réseau partiellement synchrone complet et sujet à un nombre fini mais quelconque de pannes crash. Leur résultat 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. Ils proposent alors la pseudostabilisation robuste (*fault-tolerant pseudostabilisation*) comme une alternative crédible. En effet, ils montrent qu'il est possible d'écrire des algorithmes d'élection pseudostabilisants fonctionnant en dépit de pannes définitives dynamiques (c'est-à-dire non statique) 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. En d'autres termes, le pouvoir d'ex-

4. Notons qu'en anglais, le terme choisit par les auteurs est *Fault-Tolerant Self-Stabilisation* (FTSS). Or, la stabilisation étant en elle-même tolérante à certaines fautes, il nous semble que le vocable le plus approprié devrait plutôt être « Robust Self-Stabilization », que nous traduisons en français par « stabilisation robuste ».

pression de la pseudostabilisation robuste est quasi identique à celui de l'algorithmique robuste.

Il faut aussi noter que quelques articles proposent des algorithmes autostabilisants pour des systèmes sujets à des fautes intermittentes comme la perte fréquente de messages (Delaët et Tixeuil, 2002; Delaët *et al.*, 2005). Par exemple, (Delaët *et al.*, 2005) proposent une méthode générique permettant de résoudre des tâches silencieuses dans un système à passage de messages sujet aux pertes fréquentes de messages. Enfin, il existe aussi plusieurs articles traitant de l'autostabilisation dans des systèmes sujets aux pannes byzantines[5], *e.g.*, (Nesterenko et Arora, 2002; Masuzawa et Tixeuil, 2006). (Nesterenko et Arora, 2002) proposent des algorithmes autostabilisants où l'influence des processeurs byzantins est contenue dans une région de diamètre constant. Les auteurs s'intéressent à des protocoles locaux comme le dîner des philosophes et le coloriage de nœuds. (Masuzawa et Tixeuil, 2006) montrent que deux processeurs byzantins suffisent à rendre impossible le calcul autostabilisant d'une orientation consistente dans un arbre. En revanche, ils proposent un algorithme qui stabilise en présence d'au plus un processeur byzantin. Plus précisement, les états légitimes correspondent aux états où l'orientation est consistente entre tous les processeurs non byzantins, c'est-à-dire, le rayon de contention du processeur byzantin est zéro.

5.3. *Exemple d'algorithme*

Nous présentons maintenant un algorithme autostabilisant robuste d'élection de leader proposé dans (Delporte-Gallet *et al.*, 2010). Cet algorithme fonctionne dans un réseau complet : pour chaque paire de processeurs distincts (p,q), il existe un canal de communication unidirectionel de p vers q et un autre de q vers p. Les processeurs communiquent entre eux par envoi et réception de messages. Il n'y a aucune hypothèse sur l'ordre dans lequel les messages sont délivrés. Le système est complètement synchrone, *i.e.*, les processeurs et les canaux de communication sont synchrones. Le fait que les processeurs soient *synchrones* signifie que le temps d'exécution de chacun de leurs pas de calcul est borné[6]. L'algorithme consiste en une boucle infinie contenant un nombre fini d'actions. Le temps d'exécution d'un tour de boucle est donc lui aussi borné. Pour simplifier, chaque processeur exécute un tour de boucle en une unité de temps. Le fait que les canaux de communication soient *synchrones* signifie que chaque message en transit dans un canal est délivré en un temps borné. Pour simplifier, la borne sur le temps de livraison des messages est supposée être une constante entière non nulle notée δ. Enfin, un nombre quelconque de processeurs peut tomber en panne définitivement, ces pannes étant supposées statiques.

5. Un processeur est dit byzantin lorsqu'il a un comportement quelconque qui ne suit pas le code de son protocole.
6. Il faut noter que ce modèle est plus fin que le modèle à états synchrone, en particulier, du fait de l'absence d'ordre dans la livraison des messages.

L'algorithme présenté a aussi la particularité d'être *efficace en communication* (Larrea *et al.*, 2000), c'est-à-dire qu'une fois stabilisé, un nombre minimal de liens de communication est utilisé ($n-1$ canaux unidirectionnels où n est le nombre de processeurs). Dans cet algorithme, chaque processeur maintient une variable `Elu` où est calculée l'identité d'un processeur leader. L'algorithme garantit alors qu'à partir d'une configuration initiale quelconque, le système atteint en un temps fini une configuration γ où :

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

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

L'algorithme élit le processeur vivant ayant la plus petite identité en utilisant trois principes : le premier principe assure l'efficacité en communication, les deux autres la stabilisation du système.

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

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

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

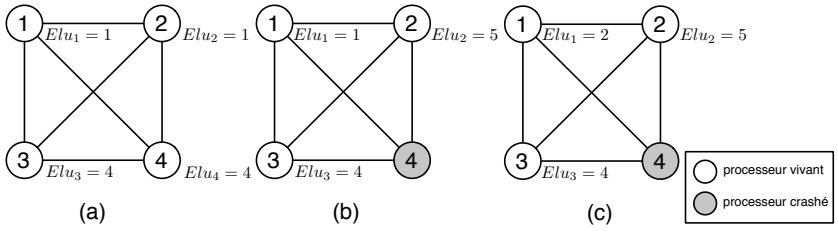


Figure 4. *Configurations initiales possibles*

L'algorithme utilise ensuite deux autres principes pour stabiliser à partir d'une configuration initiale quelconque. La figure 4 présente trois configurations initiales possibles qui illustrent les problèmes à résoudre pour obtenir une élection de leader autostabilisante. L'état des variables `Elu` étant quelconque dans la configuration ini-

tiale, plusieurs processeurs – comme les processeurs 1 et 4 dans la configuration (a) – peuvent penser être le leader. Ensuite, certaines variables `Elu` peuvent contenir des identités de processeurs qui n'existent pas ou plus dans le réseau. Par exemple, dans la configuration (b), les variables $\texttt{Elu}_2$ et $\texttt{Elu}_3$ contiennent les valeurs 5 et 4, or il n'y a pas de processeur 5 dans le réseau et le processeur 4 est crashé. Enfin, dans une configuration initiale quelconque, il est possible qu'aucun des processeurs vivants ne se considère comme un leader (cf. configuration (c)).

Pour assurer que le processeur vivant avec le plus petit identifiant se désigne un jour comme leader, nous adoptons la règle suivante :

(2) Si un processeur p ne reçoit pas de message `VIVANT` d'un processeur q tel que $q < p$ et $q \leq \texttt{Elu}_p$ pendant une longue période, alors p suspecte son leader (*i.e.*, $\texttt{Elu}_p$) et devient un leader potentiel en affectant $\texttt{Elu}_p$ à p. Cette période d'attente doit être suffisamment longue pour que p soit sûr qu'il n'y a aucun leader dans le réseau. Dans (Delporte-Gallet *et al.*, 2010), l'algorithme est prouvé en utilisant une période d'au moins $5k\delta$ unités de temps.

Grâce à la règle (2), le plus petit processeur vivant ℓ finit par vérifier $\texttt{Elu}_\ell = \ell$ pour toujours même si $\texttt{Elu}_\ell$ était initialement affecté à une identité plus petite n'existant pas dans le réseau ou appartenant à un processeur qui est tombé en panne durant l'exécution. A partir de ce moment là, ℓ envoie régulièrement des messages `VIVANT` aux autres processeurs par la règle (1). Ainsi, la règle (2) assure aussi que tous les autres processeurs vivants finissent par satisfaire $\texttt{Elu} \geq \ell$ pour toujours. Il reste alors à assurer qu'ils adoptent ℓ comme leader. Pour cela, nous appliquons la règle suivante :

(3) Lorsqu'un processeur p reçoit un message (`VIVANT`,q), p affecte $\texttt{Elu}_p$ à q si $q < p$ et $q \leq \texttt{Elu}_p$.

Les principes (1) et (2) nécessitent l'utilisation de *temporisateurs*. Deux compteurs locaux sont utilisés pour mettre en œuvre ces deux *temporisateurs*, ils sont incrémentés (modulo la période plus un) à chaque tour de boucle. Chaque tour de boucle s'effectuant en 1 unité de temps, un processeur peut déduire facilement le temps écoulé à partir de la valeur de ses compteurs. Bien sûr, dans la configuration initiale, les compteurs peuvent avoir des valeurs quelconques et, par conséquent, le premier déclenchement du temporisateur peut arriver trop tôt. Cependant, chaque compteur étant réinitialisé à 0 après chaque déclenchement, les déclenchements suivants se feront au moment prévu.

6. Convergence sûre

6.1. *Définition*

La propriété de *convergence sûre* permet d'améliorer la convergence des algorithmes autostabilisants. Bien qu'implicitement utilisée dans plusieurs articles précédents (cf. état de l'art), la notion de *convergence sûre* (*safe convergence*) apparaît pour la première fois dans (Kakugawa et Masuzawa, 2006). L'idée est la suivante. Dans un

système très instable, par exemple un réseau de capteurs où la dynamicité perturbe régulièrement la configuration du système, on souhaite converger le plus rapidement possible vers une configuration où un minimum de sûreté est garanti. Ainsi, dans un algorithme autostabilisant à convergence sûre A, deux spécifications $\mathcal{S}_1$, $\mathcal{S}_2$ sont considérées, $\mathcal{S}_2$ étant une spécialisation de $\mathcal{S}_1$ (*i.e.*, lorsque $\mathcal{S}_2$ est vérifiée, $\mathcal{S}_1$ l'est aussi, l'inverse n'étant pas vrai). L'algorithme A est alors autostabilisant à la fois pour $\mathcal{S}_1$ et $\mathcal{S}_2$ mais converge plus vite vers $\mathcal{S}_1$ que vers $\mathcal{S}_2$. Ainsi, A atteint très vite une configuration légitime de $\mathcal{S}_1$ puis tout en restant dans une configuration légitime de $\mathcal{S}_1$, il converge vers une configuration légitime de $\mathcal{S}_2$. Deux mesures de complexité en temps sont à considérer dans le cadre de l'autostabilisation à convergence sûre : le *premier* temps de convergence vers $\mathcal{S}_1$ et le *second* temps de convergence vers $\mathcal{S}_2$.

6.2. *État de l'art*

Plusieurs articles ont implicitement utilisés la notion de convergence sûre avant qu'elle ne soit formellement définie par Kakugawa et Masuzawa. (Cobb et Gouda, 2002) proposent un algorithme de routage autostabilisant qui converge d'abord vers une configuration à partir de laquelle les routes calculées sont toujours sans circuit, puis l'algorithme converge vers une configuration où les routes calculées maximisent la métrique voulue. Un autre algorithme de routage autostabilisant est proposé dans (Johnen et Tixeuil, 2003). Cet algorithme converge vers une configuration à partir de laquelle il garantit la livraison de tous les messages, même en cas de changement topologique. Tout en gardant cette propriété de sûreté, l'algorithme converge ensuite vers une configuration où les routes sont minimales en termes de nombre de sauts.

Jusqu'à maintenant, les articles faisant explicitement référence à la convergence sûre (Kakugawa et Masuzawa, 2006; Kamei et Kakugawa, 2007; Kamei et Kakugawa, 2008) proposent des algorithmes d'optimisation autostabilisants. Ce type de problème se prête naturellement à la notion de convergence sûre comme le montre la figure 5 : le système converge tout d'abord vers une configuration réalisable puis tout en restant dans une configuration réalisable, le système converge vers une configuration optimale. Par exemple, l'algorithme proposé dans (Kakugawa et Masuzawa, 2006) converge en une ronde dans une configuration réalisable où un ensemble dominant de processeurs est défini puis converge en $O(D)$ rondes (où D est le diamètre du réseau) vers une configuration optimale où l'ensemble dominant calculé est minimal (c'est-à-dire aucun sous-ensemble de cet ensemble n'est un ensemble dominant). Cet algorithme – ainsi que ceux proposés dans (Kamei et Kakugawa, 2007; Kamei et Kakugawa, 2008) – est écrit dans le modèle à états et fonctionne dans un réseau identifié synchrone de topologie quelconque. Dans (Kamei et Kakugawa, 2007; Kamei et Kakugawa, 2008), les auteurs considèrent le problème de l'ensemble dominant faiblement connecté minimal. Les deux algorithmes proposés convergent vers une configuration réalisable en une ronde. Cependant, le premier converge vers une configuration optimale en $O(n^2)$ rondes alors que le deuxième converge vers une configuration optimale en $O(n)$ rondes (où n est le nombre de processeurs du réseau).

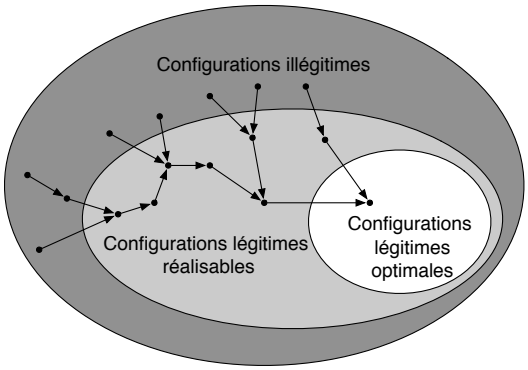


Figure 5. *Convergence sûre d'un algorithme d'optimisation*

6.3. *Exemple d'algorithme*

Nous allons maintenant illustrer la notion de convergence sûre avec l'algorithme de calcul d'un ensemble dominant minimal proposé dans (Kakugawa et Masuzawa, 2006). Cet algorithme est écrit dans le modèle à états et suppose un démon synchrone (à chaque étape[7] l'ensemble des processeurs activables exécute une action).

Un ensemble de processeurs est dit *dominant* si tout processeur du réseau qui n'est pas dans cet ensemble est voisin d'au moins un nœud qui est dans cet ensemble. Un ensemble dominant est dit *minimal* si aucun sous-ensemble propre de cet ensemble est un ensemble dominant.

L'algorithme converge en une ronde vers une configuration légitime réalisable où un ensemble dominant du réseau est distingué. Puis, l'algorithme converge en $O(D)$ rondes vers une configuration légitime optimale où un ensemble dominant minimal de processeurs est distingué. Notons que cet ensemble dominant minimal est aussi un ensemble indépendant maximal, un cas particulier d'ensemble dominant minimal où aucun élément de l'ensemble dominant n'est voisin d'un autre élément de cet ensemble.

Dans cet algorithme, chaque processeur p maintient deux variables :

– *une variable booléenne* d_p. Cette variable détermine si p se considère comme membre de l'ensemble dominant ou pas. Lorsque $d_p = 1$, p est *dominant* c'est-à-dire qu'il se considère membre de l'ensemble dominant. Dans le cas contraire, p est dit *dominé* ;

– *une variable* m_p. Cette variable contient soit l'identité du processeur de p (noté simplement p) soit l'identité d'un voisin de p. Cette variable est utilisée pour construire un ensemble à la fois indépendant et dominant (un tel ensemble est alors un ensemble

7. Notons que dans un modèle synchrone, une étape de calcul correspond à une ronde.

dominant minimal). Un processeur p se considère membre de l'ensemble dominant indépendant en construction si et seulement si $d_p = 1$ et $m_p = p$. Dans ce cas, p est dit *dominant indépendant*.

Dans une configuration légitime réalisable chaque processeur p vérifie la condition $Safe(p)$: si $d_p = 0$ alors m_p contient l'identité d'un voisin q tel que $d_q = 1$. De plus, dans une configuration légitime optimale, chaque processeur p sera soit un dominant indépendant ($d_p = 1 \wedge m_p = p$) soit un dominé ($d_p = 0$) tel que ($m_p = q$) où q est voisin dominant indépendant.

Les variables sont modifiées à l'aide des 5 règles mutuellement exclusives dont les priorités sont décroissantes :

– *Règle 1.* Lorsque l'identité d'un processeur p est un maximum local, il devient un dominant indépendant s'il ne l'est pas déjà (*i.e.*, $d_p \leftarrow 1$ et $m_p \leftarrow p$).

– *Règle 2.* Lorsqu'un processeur n'a pas de voisin qui est un dominant indépendant, il devient un dominant indépendant s'il ne l'est pas déjà.

– *Règle 3.* Lorsqu'un processeur a une identité plus grande que celle de ses voisins dominant indépendant, il devient un dominant indépendant s'il ne l'est pas déjà.

– *Règle 4.* Lorsqu'un processeur p a au moins un voisin dominant indépendant mais que sa variable m_p n'est pas affectée à l'identité la plus grande parmi celles de ses voisins dominants indépendants, p devient un dominant et affecte m_p à l'identité la plus grande parmi celle de ses voisins dominants indépendants.

– *Règle 5.* Un processeur p peut passer du statut de dominant à celui de dominé si, (1) il a au moins un voisin dominant indépendant, (2) sa variable m_p est correctement affectée vis-à-vis de la règle 4 et (3), aucun voisin de p ne se considère dominé par p (aucun voisin q ne vérifie $m_q = p$).

Les règles assurent la convergence en une ronde vers une configuration réalisable. En effet, grâce aux quatres premières règles, si un processeur p ne vérifie pas la condition $Safe(p)$ alors il exécute l'une des quatres premières règles et devient un dominant en une ronde. Les règles 4 et 5 assurent que si un processeur p perd son statut de dominant durant la première ronde, il vérifie tout de même $Safe(p)$ après la première ronde. En effet, pour perdre son statut de dominant, un processeur doit au moins avoir un voisin dominant indépendant et celui-ci ne peut perdre son statut de dominant qu'après au moins deux rondes (d'abord il passe de dominant indépendant à dominant puis il passe de dominant à dominé).

Après la première ronde, si un processeur dominant p a un voisin dominant indépendant q tel que $q > p$, il doit devenir dominé non indépendant. Pour cela, il affecte d'abord sa variable m_p à l'identité la plus grande parmi celle de ses voisins dominants indépendants, si ce n'est pas déjà le cas (règle 4). Ensuite, tous les voisins qui le considéraient comme dominant deviennent dominants et choisissent un autre dominant que p (eux-mêmes ou l'un de leur voisin dominant indépendant) lors de la ronde suivante (règles 2 ou 4). p peut alors perdre son statut de dominant dans la ronde suivante (règle 5) sans que le système ne quitte une configuration réalisable. Ainsi, la

convergence sûre vers une configuration optimale est réalisée de proche en proche et le système converge de manière « sûre » en $O(D)$ rondes, où D est le diamètre du réseau.

7. Stabilisation instantanée

7.1. *Définition*

La stabilisation instantanée a été définie pour la première fois dans (Bui *et al.*, 1999). Un algorithme instantanément stabilisant est un algorithme qui, quelle que soit la configuration initiale, vérifie toujours la spécification pour laquelle il est écrit. Ainsi un algorithme instantanément stabilisant est un algorithme autostabilisant dont le temps de stabilisation est nul.

La façon dont ont été étudiés les problèmes dans le cadre de l'autostabilisation et ses formes dérivées mérite de s'y attarder quelque peu. Nous verrons en effet qu'elle est essentielle pour comprendre en quoi la stabilisation instantanée constitue une véritable rupture dans l'approche des problèmes à résoudre et pourquoi en particuliers la puissance d'expression de la stabilisation instantanée est quasiment équivalente à celle de l'autostabilisation malgré une définition manifestement plus radicale.

Contrairement aux approches des sections précédentes, il ne s'agit pas d'obtenir des propriétés supplémentaires à la stabilisation par la conception astucieuse de nouveaux algorithmes performants, mais de constater les inconvénients de la définition d'un phénomène dynamique (les exécutions d'un algorithme) par des critères statiques (les configurations) et d'en tirer les conséquences.

Le point faible de l'approche classique de l'autostabilisation est la définition des spécifications du problème à résoudre en termes de configurations exclues. Dès lors que cet ensemble de configurations est non vide, un algorithme autostabilisant ne peut être instantanément stabilisant puisqu'il suffit que la configuration initiale fasse partie de cet ensemble pour que l'algorithme ne puisse vérifier la spécification au début de l'exécution. Il s'ensuit que, non seulement cette approche est peu adaptée à la description d'un phénomène dynamique, mais en plus, elle interdit d'imaginer ce qui est inimaginable par cette approche : un temps de stabilisation nul. En effet, comment espérer qu'un grand nombre de problèmes puisse être défini avec un ensemble de configurations exclues vide ?[8]

Or, non seulement une approche basée sur le côté dynamique du problème (actions) plutôt que par le côté statique (configurations) permet de s'affranchir complètement de cette notion de configurations exclues, mais elle permet aussi d'entrevoir des

8. Notons cependant qu'il existe quelques solutions pour des problèmes et des topologies particulières qui sont définis par un ensemble de configurations exclues vide (Johnen *et al.*, 2002), y compris dans des travaux où il n'est fait mention que d'autostabilisation, mais où les algorithmes présentés sont instantanéments stabilisants (Herman et Ghosh, 1995; Gouda et Haddix, 1997; Boulinier *et al.*, 2006).

algorithmes pour lesquels l'enchaînement des actions obtenu, quelle que soit la configuration initiale, correspond à ce que l'utilisateur en attend (Bui *et al.*, 2007). Tout le problème revient alors à expliquer formellement ce phénomène.

7.2. *Un exemple démonstratif : l'exclusion mutuelle*

Le partage de ressources en accès exclusif est un problème classique en gestion de systèmes et aussi en algorithmique distribuée. La spécification généralement utilisée en autostabilisation ressemble à ceci :

Vivacité. Tout processeur demandeur de la SC (Section Critique) peut entrer en SC en un temps fini.

Sûreté. Jamais plus d'un processeur en SC simultanément. [S1]

Dans la discussion qui suit, seule la sûreté est intéressante comme génératrice de problèmes dans l'obtention d'une solution instantanément stabilisante. Il est clair qu'ainsi formulée, cette spécification interdit à tout algorithme d'exclusion mutuelle d'être instantanément stabilisant : toute configuration initiale comportant au moins deux processeurs en SC fait que le début de l'exécution ne vérifie pas [S1].

Le premier travail à effectuer lors de la recherche d'une solution instantanément stabilisante est donc de trouver une spécification dynamique. Dans le cas de l'exclusion mutuelle, le comportement d'un processeur suit une règle immuable quel que soit l'algorithme considéré, cette règle est la suivante : le processeur qui doit utiliser la SC fait une demande d'accès, exécute ensuite la SC quand il y est autorisé puis libère cette section, la rendant ainsi accessible à un autre demandeur (éventuellement lui-même). Le comportement peut donc être schématisé en trois phases essentielles : Demande, Entrée en SC et Libération, qui s'exécutent de façon séquentielle. Nous pouvons alors formuler une autre spécification de la sûreté du problème de l'exclusion mutuelle :

Sûreté. Tout processeur demandeur qui exécute la SC, l'exécute seul. [S2]

Ou encore :

Sûreté. Si plusieurs processeurs exécutent la SC simultanément, alors aucun d'entre eux n'est demandeur. [S3]

La formulation [S2] semble plus proche de [S1] que [S3]. Cependant il est facile de constater que [S2] et [S3] sont équivalentes car [S3] est simplement la contraposée de [S2]. L'intérêt de [S3] est que transparaît dans son énoncé la réponse à la question : quel est l'apport de cette spécification ? Si plusieurs processeurs sont en SC dans la configuration initiale alors aucun de ces processeurs n'a pu exécuter la phase de demande et cela ne contredit donc aucune des spécifications [S2] et [S3].

Est-il pour autant possible d'écrire un algorithme instantanément stabilisant pour ces spécifications ? La réponse est oui (Cournier *et al.*, 2009). Cet algorithme utilise une circulation de jeton unique instantanément stabilisante basée sur un reset général reposant sur un algorithme de PIR (Propagation d'Information avec Retour) qui est bien sûr instantanément stabilisant et possède la propriété fondamentale suivante : tout processeur peut être abusé au plus par deux messages non initialisés par le processeur initiateur du PIR. Tout processeur désirant entrer en SC, fait donc sa demande en initialisant un compteur à 0. Il laisse passer sans les utiliser les deux premiers jetons qu'il reçoit (il incrémente juste son compteur) et au troisième jeton reçu, il est assuré (d'après la propriété du PIR évoquée ci-dessus) que le jeton est bien unique dans le réseau et qu'il peut donc l'utiliser comme autorisation d'entrée en SC.

Dès lors se pose la question essentielle : les spécifications [S2] et [S3] définissent-elles le même problème que [S1] et si oui en quel sens ?

7.3. *Equivalence de spécifications*

Il est clair que [S1] et [S2] ne sont pas les mêmes spécifications. En effet, il a été montré précedemment que [S2] et [S3] étaient équivalentes, mais que [S1] n'était pas équivalente à [S2] en ce sens qu'il est possible d'écrire des solutions instantanément stabilisantes pour [S2], mais pas pour [S1]. Faut-il en conclure qu'elles ne définissent pas le même problème ?

Lorsque le problème de l'exclusion mutuelle est énoncé informellement, il l'est *a priori* dans un contexte libre de toute perturbation (c'est-à-dire en l'absence de toute défaillance). Et les solutions, dans ce contexte, sont des algorithmes pour lesquels les configurations initiales sont parfaitement définies (sous-ensemble strictement inclus dans l'ensemble de toutes les configurations). Il est donc normal que la définition formelle (spécification) obtenue tienne compte explicitement ou implicitement des conditions idéales de l'environnement considéré à savoir un système dans lequel il n'y a aucune faute. Appelons un tel système un « système sauf ».

Cela étant posé, est-il raisonnable lorsque le système n'est plus sauf (appelons-le « système perturbé ») de ne pas tenir compte de la (ou des) perturbation(s) possible(s) subie(s) par le système ? La réponse dictée par la stabilisation instantanée est bien évidemment non, car la réflexion économisée sur la définition du problème dans un contexte différent risque d'avoir d'énormes conséquences sur les solutions proposées. C'est ce qui permet notamment de faire la différence entre des solutions instantanément stabilisantes et des solutions autostabilisantes n'étant pas instantanément stabilisantes.

Il reste néanmoins à montrer que, si différence il y a entre [S1] et [S2], cette différence n'est pas significative dans le cadre d'un système sauf. Plus précisément il faut montrer que [S1] est équivalente à [S2] suivant la définition suivante.

Définition 1 *Soient deux spécifications S et S'. Nous disons que $S \Rightarrow S'$, si toute solution pour S dans un système sauf est une solution pour S' dans un système sauf. S et S' sont dites équivalentes ($S \Leftrightarrow S'$) si et seulement si $S \Rightarrow S'$ et $S' \Rightarrow S$.*

Dans un système sauf, tout processeur entrant en SC est forcément un processeur qui a effectué sa phase de demande, il y a donc équivalence entre la notion de processeur en SC et celle de processeur demandeur en SC. Donc toute solution vérifiant la sûreté de [S1] vérifie celle de [S2] et réciproquement. La spécification [S2] pour laquelle il existe au moins une solution instantanément stabilisante définit donc le même problème que [S1]. Elle tient simplement compte du type de perturbation de l'environnement du système, permettant ainsi de résoudre le problème posé d'une manière plus efficace.

Il faut cependant noter qu'à aucun moment il n'a été démontré que [S1] ou [S2] définissait parfaitement le problème de l'exclusion mutuelle. L'énoncé initial de ce problème étant informel (comme la plupart des énoncés de problème) il est malheureusement impossible de démontrer formellement que [S1] est une spécification exacte de ce problème. Nous pouvons simplement constater qu'elle est communément admise comme définissant formellement ce problème. Et d'après ce qui vient d'être démontré, il est tout aussi clair qu'il n'y pas de spécification unique pour un problème donné.

Nous pouvons alors conclure de manière formelle que si [S1] est une spécification acceptable pour le problème de l'exclusion mutuelle alors [S2] l'est aussi et la solution de (Cournier *et al.*, 2009) est donc bien une solution instantanément stabilisante pour le problème de l'exclusion mutuelle.

7.4. *La stabilisation instantanée : un travail en amont et en aval de la spécification*

Nous avons vu au cours de la discussion précédente qu'une véritable rupture existe entre l'approche classique de l'autostabilisation et de ses formes dérivées et celle de la stabilisation instantanée. Il est remarquable à cet égard de constater que les algorithmes instantanément stabilisants qui ont été écrits jusqu'à aujourd'hui auraient très bien pu être trouvés comme solutions autostabilisantes à des spécifications à base de configurations exclues mais que la forme de raisonnement sous-jacente ne permettait pas de l'envisager consciemment. En effet, la concentration de la réflexion étant toute dirigée vers un seul but : atteindre l'ensemble des configurations non exclues, toutes les autres propriétés intéressantes sont restées inaccessibles parce qu'invisibles jusqu'à la définition de la stabilisation instantanée. C'est aussi pourquoi, par exemple, les algorithmes instantanément stabilisants de PIR ou de circulation de jeton sont des solutions meilleures ou aussi bonnes sur tous les critères (espace et temps) que les solutions autostabilisantes connues jusqu'alors dans le modèle à états, que ce soit dans l'arbre (Bui *et al.*, 2007; Petit et Villain, 2007) où les bornes exactes sont atteintes ou dans un graphe quelconque (Cournier *et al.*, 2006; Cournier *et al.*, 2009) où des solutions asymptotiquement optimales en nombre de rondes existent.

La stabilisation instantanée a souvent été regardée de façon circonspecte jusqu'à maintenant car elle semblait remettre en cause des résultats classiques sur la notion de borne minimale sur le temps de stabilisation concernant certains problèmes. En effet, le résultat de (Cournier *et al.*, 2003) signifie que dans le modèle à états, tous les problèmes sur systèmes non anonymes admettant une solution autostabilisante admettent une solution instantanément stabilisante. En d'autres termes, la stabilisation instantanée est aussi puissante que l'autostabilisation dans le cadre du modèle à états pour les systèmes non anonymes. La conséquence fondamentale de ce résultat est que tous les résultats proclamant une borne minimale sur le temps de stabilisation de l'ordre de $\Omega(D)$, où D est le diamètre du réseau pour certains problèmes sont à relire en tenant compte du fait qu'il ne s'agit pas d'une borne liée au problème comme il l'est souvent annoncé (puisque l'existence de solutions instantanément stabilisantes montre que l'optimal est en $\Theta(1)$) mais seulement d'une borne due à la spécification choisie pour décrire le problème.

L'étape finale de la recherche concernant la stabilisation instantanée est la démonstration de sa faisabilité dans le modèle à messages. A l'heure actuelle, il est connu (Delaët *et al.*, 2009) que la plupart des problèmes n'ont pas de solutions instantanément stabilisantes dans un modèle à passage de messages ayant des canaux de communication à capacité non bornée. En revanche, l'approche consistant à supposer une borne sur la capacité des canaux semble prometteuse, plusieurs solutions pour des problèmes et des topologies spécifiques ayant déjà été proposées (Dolev et Tzachar, 2009; Delaët *et al.*, 2009). Cependant un résultat général est toujours attendu : la stabilisation instantanée est-elle aussi puissante (du point de vue des problèmes) que l'autostabilisation dans les systèmes à passage de messages ayant des canaux de communication à capacité bornée ?

Enfin, nous conclons sur ce sujet en rappelant l'importance de la méthodologie à la base de la stabilisation instantanée : « (re)définir le problème de manière formelle en intégrant la spécificité de l'environnement perturbé et dans un deuxième temps, chercher des solutions efficaces ».

8. Conclusion

Dans cet article, nous avons présenté des *spécialisations* de l'autostabilisation[9]. Parmi celles proposées dans la littérature, nous avons présenté la *superstabilisation*, le *confinement de fautes*, l'*autostabilisation adaptable en temps*, l'*autostabilisation robuste*, la *convergence sûre* et la *stabilisation instantanée*. Notre inventaire n'est pas exhaustif, d'autres approches plus fortes ayant été proposées, comme la « stabilisation sûre » (Ghosh et Bejan, 2003) et la « stabilisation k-forte » (Beauquier *et al.*, 2006). La première permet d'écrire des algorithmes autostabilisants silencieux ayant la propriété supplémentaire suivante : à partir d'une configuration vérifiant un prédicat de sûreté particulier (plus faible que la spécification de l'algorithme), ce prédicat est ou reste

9. Des *généralisations* sont traitées dans (Devismes *et al.*, 2011).

satisfait quand le nombre de fautes est inférieur ou égal à une valeur k donnée. La seconde traite des algorithmes permettant d'isoler fortement les fautes, c'est-à-dire que le comportement des processeurs non défaillants n'est jamais perturbé, qu'il y ait ou non des défaillances dans le système, y compris dans leur voisinage. Ils doivent en outre assurer qu'après au plus k fautes, le temps de recouvrement est linéaire en fonction de k.

Ces différentes techniques ont parfois été introduites pour s'adapter à un contexte particulier, par exemple la superstabilisation pour les systèmes dynamiques, la stabilisation adaptable en temps et la stabilisation sûre aux algorithmes silencieux, etc. Il est fort à parier que la liste n'est pas prête d'être exhaustive car tous les domaines où ce type d'approche pourrait aider à établir des résultats intéressants n'ont pas été explorés, par exemple les réseaux à large échelle, les systèmes pair-à-pair, ou encore les systèmes de robots mobiles autonomes.

9. Bibliographie

Anagnostou E., Hadzilacos V., « Tolerating Transient and Permanent Failures (Extended Abstract) », *WDAG*, p. 174-188, 1993.

Beauquier J., Delaët S., Haddad S., « A 1-Strong Self-stabilizing Transformer », *8th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS 2006)*, p. 95-109, 2006.

Beauquier J., Kekkonen-Moneta S., « Fault-tolerance and self-stabilization : impossibility results and solutions using self-stabilizing failure detectors », *Int. J. Systems Science*, vol. 28, n° 11, p. 1177-1187, 1997a.

Beauquier J., Kekkonen-Moneta S., « On FTSS-solvable distributed problems », *WSS*, p. 64-79, 1997b.

Boulinier C., Petit F., Villain V., « Toward a Time-Optimal Odd Phase Clock Unison in Trees », *SSS*, p. 137-151, 2006.

Bui A., Datta A. K., Petit F., Villain V., « State-optimal snap-stabilizing PIF in tree networks », *WSS*, p. 78-85, 1999.

Bui A., Datta A. K., Petit F., Villain V., « Snap-stabilization and PIF in tree networks », *Distributed Computing*, vol. 20, n° 1, p. 3-19, 2007.

Burman J., Herman T., Kutten S., Patt-Shamir B., « Asynchronous and Fully Self-stabilizing Time-Adaptive Majority Consensus », *OPODIS*, p. 146-160, 2005.

Chen N., Yu H., Huang S., « A self-stabilizing algorithm for constructing spanning trees », *Information Processing Letters*, vol. 39, p. 147-151, 1991.

Cobb J. A., Gouda M. G., « Stabilization of General Loop-Free Routing », *J. Parallel Distrib. Comput.*, vol. 62, n° 5, p. 922-944, 2002.

Cournier A., Datta A. K., Petit F., Villain V., « Enabling Snap-Stabilization », *ICDCS*, p. 12-19, 2003.

Cournier A., Devismes S., Villain V., « Snap-Stabilizing PIF and Useless Computations », *ICPADS (1)*, p. 39-48, 2006.

Cournier A., Devismes S., Villain V., « Light enabling snap-stabilization of fundamental protocols », *ACM Trans. Auton. Adapt. Syst.*, vol. 4, p. 6 :1-6 :27, February, 2009.

Delaët S., Devismes S., Nesterenko M., Tixeuil S., « Snap-Stabilization in Message-Passing Systems », *International Conference on Distributed Systems and Networks (ICDCN 2009)*, n° 5404 in *LNCS*, p. 281-286, January, 2009.

Delaët S., Ducourthial B., Tixeuil S., « Self-stabilization with r-Operators Revisited », *Self-Stabilizing Systems*, p. 68-80, 2005.

Delaët S., Tixeuil S., « Tolerating Transient and Intermittent Failures », *J. Parallel Distrib. Comput.*, vol. 62, n° 5, p. 961-981, 2002.

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'auto-stabilisation : Techniques généralisant l'approche », *TSI*, 2011. Au sommaire de ce même numéro.

Dolev S., Gouda M. G., Schneider M., « Memory Requirements for Silent Stabilization », *Acta Inf.*, vol. 36, n° 6, p. 447-462, 1999.

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

Dolev S., Herman T., « Parallel composition for time-to-fault adaptive stabilization », *Distributed Computing*, vol. 20, n° 1, p. 29-38, 2007.

Dolev S., Tzachar N., « Empire of colonies : Self-stabilizing and self-organizing distributed algorithm », *Theor. Comput. Sci.*, vol. 410, n° 6-7, p. 514-532, 2009.

Ghosh S., Bejan A., « A Framework of Safe Stabilization », *6th International Symposium on Self-Stabilizing Systems (SSS 2003)*, p. 129-140, 2003.

Ghosh S., Gupta A., « An Exercise in Fault-Containment : Self-Stabilizing Leader Election », *Inf. Process. Lett.*, vol. 59, n° 5, p. 281-288, 1996.

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.

Ghosh S., He X., « Fault-containing self-stabilization using priority scheduling », *Inf. Process. Lett.*, vol. 73, n° 3-4, p. 145-151, 2000.

Gopal A. S., Perry K. J., « Unifying Self-Stabilization and Fault-Tolerance (Preliminary Version) », *PODC*, p. 195-206, 1993.

Gouda M. G., Haddix F. F., « The linear alternator », *WSS*, p. 31-47, 1997.

Herman T., Observations on Time-Adaptive Self-Stabilization, Technical report, University of Iowa Iowa City, 1997.

Herman T., « Superstabilizing Mutual Exclusion », *Distributed Computing*, vol. 13, n° 1, p. 1-17, 2000.

Herman T., Ghosh S., « Stabilizing Phase-Clocks », *Inf. Process. Lett.*, vol. 54, n° 5, p. 259-265, 1995.

Herman T., Pemmaraju S. V., « Error-detecting codes and fault-containing self-stabilization », *Inf. Process. Lett.*, vol. 73, n° 1-2, p. 41-46, 2000.

Huang S.-T., Chen N.-S., « A Self-Stabilizing Algorithm for Constructing Breadth-First Trees », *Inf. Process. Lett.*, vol. 41, n° 2, p. 109-117, 1992.

Hutle M., Widder J., « On the Possibility and the Impossibility of Message-Driven Self-stabilizing Failure Detection », *Self-Stabilizing Systems*, p. 153-170, 2005a.

Hutle M., Widder J., « Self-Stabilizing Failure Detector Algorithms », *Parallel and Distributed Computing and Networks*, p. 485-490, 2005b.

Johnen C., Alima L., Datta A. K., Tixeuil S., « Optimal Snap-stabilizing Neighborhood Synchronizer in Tree Networks », *Parallel Processing Letters*, vol. 12, n° 3-4, p. 327-340, 2002.

Johnen C., Tixeuil S., « Route Preserving Stabilization », *Self-Stabilizing Systems*, p. 184-198, 2003.

Kakugawa H., Masuzawa T., « A self-stabilizing minimal dominating set algorithm with safe convergence », *Proceedings of the 20th international conference on Parallel and distributed processing*, IPDPS'06, IEEE Computer Society, Washington, DC, USA, p. 263-263, 2006.

Kamei S., Kakugawa H., « A Self-stabilizing Approximation Algorithm for the Minimum Weakly Connected Dominating Set with Safe Convergence », *Proceedings of the First International Workshop on Reliability, Availability, and Security (WRAS)*, Paris, France, p. 57-67, September, 2007.

Kamei S., Kakugawa H., « A Self-stabilizing Approximation for the Minimum Connected Dominating Set with Safe Convergence », *OPODIS*, p. 496-511, 2008.

Katayama Y., Ueda E., Fujiwara H., Masuzawa T., « A Latency Optimal Superstabilizing Mutual Exclusion Protocol in Unidirectional Rings », *J. Parallel Distrib. Comput.*, vol. 62, n° 5, p. 865-884, 2002.

Kutten S., Patt-Shamir B., « Time-Adaptive Self Stabilization », *PODC*, p. 149-158, 1997.

Kutten S., Patt-Shamir B., « Asynchronous Time-Adaptive Self Stabilization », *PODC*, p. 319, 1998.

Kutten S., Patt-Shamir B., « Stabilizing Time-Adaptive Protocols », *Theor. Comput. Sci.*, vol. 220, n° 1, p. 93-111, 1999.

Larrea M., Fernández A., Arévalo S., « Optimal Implementation of the Weakest Failure Detector for Solving Consensus », *SRDS*, p. 52-59, 2000.

Masuzawa T., Tixeuil S., « Bounding the Impact of Unbounded Attacks in Stabilization », *SSS*, p. 440-453, 2006.

Nesterenko M., Arora A., « Tolerance to Unbounded Byzantine Faults », *SRDS*, p. 22-31, 2002.

Petit F., Villain V., « Optimal snap-stabilizing depth-first token circulation in tree networks », *J. Parallel Distrib. Comput.*, vol. 67, n° 1, p. 1-12, 2007.

Varghese G., Jayaram M., « The Fault Span of Crash Failures », *Journal of the ACM*, vol. 47, n° 2, p. 244-293, March, 2000.

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 :

Spécialisation 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