Vers l'auto-stabilisation des systèmes à grande échelle

Sébastien Tixeuil

Rapport scientifique présenté en vue de l'obtention de l'Habilitation à Diriger des Recherches

Table des matières

Chapitre 1

Introduction

Historiquement, le développement des systèmes répartis a été principalement lié à plusieurs besoins, comme la communication entre entités géographiquement distantes, l'accélération des calculs suite à l'augmentation des ressources, la fiabilisation des systèmes due à la redondance des moyens de calcul. Plusieurs facteurs économiques ont également joué, comme le fait que plusieurs machines simples sont moins coûteuses qu'une seule machine complexe, à nombre d'utilisateurs égaux.

L'étude des systèmes répartis et des algorithmes spécifiques à ces systèmes, les algorithmes répartis, rend compte des spécificités de ces systèmes par rapport aux systèmes centralisés traditionnels : les informations sont locales (chaque élément du système ne possède qu'une fraction des informations, et doit en obtenir d'autres en communiquant avec d'autres éléments), et le temps est local (les éléments du système peuvent exécuter leurs instructions à des vitesses différentes, sans connaissance de celles des autres éléments). Ces deux facteurs induisent un non-déterminisme qui fait que deux exécutions successives du même système réparti seront probablement différentes. Le fait que certains éléments du système puissent tomber en panne accroît ce non-déterminisme et la difficulté que l'on a à prédire le comportement du système global.

Les algorithmes répartis posent les bases algorithmiques des protocoles qui sont utilisés dans les réseaux réels. Ils interviennent donc largement en amont de la phase d'implantation, et permettent de déterminer parmi plusieurs approches celles qui sont impossible à mettre en œuvre, celles qui sont coûteuses, ou celles qui sont potentiellement efficaces.

Lorsqu'on augmente le nombre de composants d'un système réparti, la possibilité qu'un ou plusieurs de ces composants tombe en panne augmente également. Lorsqu'on réduit le coût de fabrication des composants pour obtenir des économies d'échelle, on accroît également le taux de défauts potentiels. Enfin, lorsqu'on déploie les composants du système dans un environnement que l'on ne contrôle pas nécessairement, les risques de

pannes deviennent impossibles à négliger.

Dans les réseaux de capteurs, ces trois facteurs principaux sont combinés. Aussi, il devient certain que des pannes se produiront au cours de la vie du système. Il importe alors de maximiser la *vie utile* du système : c'est-à-dire le temps pendant lequel le système fournira des résultats utiles. En effet, les pannes peuvent avoir des répercussions importantes pour l'application qu'on exécute sur le système. Ces répercussions dépendent de la sévérité de la panne.

Dans ce mémoire, il est question d'algorithmique répartie dans les systèmes à grande échelle, c'est-à-dire des systèmes répartis pouvant comporter plusieurs dizaines (ou centaines) de milliers de machines élémentaires (ordinateurs, capteurs, *etc.*). Plus précisément, on se restreint ici à l'algorithmique répartie tolérante aux pannes. Dans le chapitre 2, on présente la taxonomie des pannes susceptibles d'apparaître et les approches classiques pour résoudre les problèmes qui y sont liés dans les systèmes répartis (sans l'aspect grande échelle). Le chapitre 3 montre que le passage à l'échelle est compromis par les hypothèses habituellement faites en algorithmique tolérante aux pannes. Par la suite, le chapitre 4 présente plusieurs pistes pour l'utilisation de techniques particulières de tolérance aux pannes (dérivées de l'auto-stabilisation) dans un contexte de grande échelle. Dans tout le reste du document, les références bibliographiques qui apparaissent en gras sont celles signées ou cosignées par l'auteur de ce mémoire.

Chapitre 2

Algorithmique répartie tolérante aux pannes

De manière classique, on représente généralement un système réparti par un graphe, dont les nœuds sont les machines (ou les capteurs) du système, et les liens représentent la capacité de communication de deux machines. Ainsi, deux machines sont reliées si elles sont capables de se communiquer des informations (par une connexion réseau par exemple). Dans certains cas, les liens du graphe sont orientés pour représenter le fait que la communication ne peut avoir lieu que dans un seul sens (par exemple une communication sans fils d'un satellite vers une antenne au sol). Dans la suite, on utilise indistinctement les termes de machine, capteur, nœud ou processus suivant le contexte.

2.1 Taxonomie des pannes dans les systèmes répartis

Un premier critère pour classifier les pannes dans les systèmes répartis est la localisation dans le temps. On distingue généralement trois types de pannes possibles :

1. *les pannes transitoires* : des pannes de nature arbitraire peuvent venir frapper le système, mais il existe un point de l'exécution à partir duquel ces pannes n'apparaissent plus ;
2. *les pannes définitives* : des pannes de nature arbitraire peuvent venir frapper le système, mais il existe un point de l'exécution à partir duquel ces pannes anéantissent pour toujours ceux qui en sont frappés ;
3. *les pannes intermittentes* : des pannes de nature arbitraire peuvent venir frapper le système, à tout moment de l'exécution.

Bien sûr, les pannes transitoires et les pannes définitives sont deux cas particuliers des pannes intermittentes. Cependant, dans un système ou les pannes intermittentes apparaissent rarement, un système qui tolère des

pannes transitoires peut être utile, car la vie utile du système peut rester suffisamment élevée.

Un deuxième critère est la nature des fautes. Un élément d'un système réparti peut être représenté par un automate dont les états représentent les valeurs possibles des variables de l'élément, et dont les transitions représentent le code exécuté par l'élément. On peut alors distinguer ces fautes suivant qu'elles surviennent sur l'état ou sur le code de l'élément :

les pannes d'état : le changement des variables d'un élément peut être dû à des perturbations dues à l'environnement (par exemple des ondes électromagnétiques), des attaques (par exemple un dépassement de tampon) ou simplement des défaillances du matériel utilisé. Il est par exemple possible que des variables prennent des valeurs qu'elles ne sont pas sensées prendre lors d'une exécution normale du système.

les pannes de code : le changement arbitraire du code d'un élément résulte la plupart du temps d'une attaque (par exemple le remplacement d'un élément par un adversaire malicieux), mais certains types moins graves peuvent correspondre à des bogues où à une difficulté à tenir la charge. On distingue donc plusieurs sous-catégories de pannes de code :

1. *les pannes crash* : à un point donné de l'exécution, un élément cesse définitivement son exécution et n'effectue plus aucune action ;
2. *les omissions* : à divers instants de l'exécution, un élément peut omettre de communiquer avec les autres éléments du système, soit en émission, soit en réception ;
3. *les duplications* : à divers instants de l'exécution, un élément peut effectuer une action plusieurs fois, quand bien même son code stipule que cette action doit être exécutée une fois ;
4. *les désequencements* : à divers instants de l'exécution, un élément peut effectuer des actions correctes, mais dans le désordre ;
5. *les pannes byzantines* : elles correspondent simplement à un type arbitraire de pannes, et sont donc les pannes les plus malicieuses possibles.

Les pannes crash sont incluses dans les omissions (un élément qui ne communique plus est perçu par le reste du système comme un élément qui a interrompu son exécution). Les omissions sont trivialement incluses dans les pannes byzantines. Les duplications et déséquencements sont également incluses dans les pannes byzantines, mais sont généralement considérées pour des comportements purement liés aux capacités de communication.

La figure 2.1 résume les relations d'inclusions pouvant être induites entre les différents types de pannes.

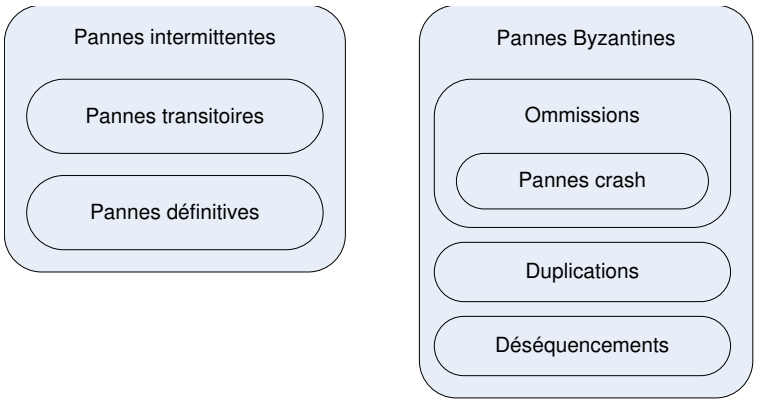

FIG. 2.1 – Taxonomie des pannes dans les systèmes répartis

2.2 Classes d'algorithmes tolérants aux pannes

Quand des pannes se produisent sur un ou plusieurs des éléments constitutifs d'un système réparti, il est essentiel de pouvoir les traiter. Si un système ne tolère aucune panne, la défaillance d'un seul de ses éléments peut compromettre la bonne exécution de tout le système : c'est le cas dans un système où une entité a un rôle central (comme le DNS). Pour préserver la vie utile du système, de nombreuses approches *ad hoc* ont été développées, en général spécifiques à un type de panne particulier et susceptible de se produire dans le système considéré. Cependant, ces solutions peuvent être classifiées suivant que l'effet des fautes est visible à un observateur (par exemple un utilisateur) du système ou pas. Une solution *masquante* cache à l'observateur l'occurrence des fautes (si celles-ci restent dans la limite tolérée par le système), alors qu'une solution *non masquante* ne présente pas cette propriété : l'effet des fautes est visible pendant un temps plus ou moins long, puis le système recommence à se comporter correctement.

Une approche masquante apparaît *a priori* préférable, puisqu'elle s'applique à un plus grand nombre d'applications (et en particulier les applications critiques de sécurité). En outre, elle correspond à l'idée intuitive que l'on se fait de la tolérance aux pannes : tant qu'une portion suffisament grande du système reste opérationnelle, on est en mesure d'obtenir le résultat escompté. Cependant, en général, une solution masquante est plus coûteuse (en ressources, en temps) qu'une solution non masquante, et ne peut tolérer des fautes que dans la mesure où celles-ci ont été prévues. En Informatique, pour des problèmes tels que le routage, où ne pas être capable d'acheminer des informations pendant quelques instants n'a pas de conséquences catastrophiques, une approche non-masquante est tout à fait indiquée (voir également section 3.2.1).

Deux classes principales d'algorithmes tolérants aux pannes peuvent être distinguées :

les algorithmes robustes : ils utilisent la redondance à plusieurs niveaux des informations, des communications, ou des nœuds du système, pour permettre des recoupements suffisant à s'assurer qu'exécuter la suite du code ne présente pas de danger. Ils font en général l'hypothèse qu'un nombre limité de fautes peut frapper le système, de manière à conserver au moins une majorité de éléments corrects (parfois plus si les fautes sont plus sévères). De tels algorithmes sont typiquement masquants.

les algorithmes auto-stabilisants : ils font l'hypothèse que les défaillances sont transitoires (c'est-à-dire limitées dans le temps), mais ne donnent pas de contraintes quant à l'étendue des fautes (qui peuvent concerner *tous* les éléments du système). Un algorithme est auto-stabilisant au sens de [22] s'il parvient en temps fini, à exhiber un comportement correct *indépendamment de l'état initial de ses éléments*, c'est-à-dire que les variables des éléments peuvent se trouver dans un état arbitraire (et impossible à atteindre par un cheminement normal de l'application). Les algorithmes auto-stabilisants sont typiquement non-masquants, car entre le moment ou les fautes cessent et le moment où le système est stabilisé sur un comportement correct, l'exécution peut s'avérer quelque peu erratique.

Les algorithmes robustes correspondent bien à la notion intuitive que l'on se fait de la tolérance aux pannes. Si un élément est susceptible de tomber en panne, alors on remplace chaque élément par trois éléments identiques, et à chaque fois qu'une action est entreprise, on fait effectuer l'action trois fois par chacun des éléments, et l'action effectivement entreprise est celle qui correspond à la majorité des trois actions individuelles. L'auto-stabilisation est *a priori* plus liée à la notion de convergence en mathématique ou en automatique, où on cherche à atteindre un point fixe indépendamment de la position initiale ; le point fixe correspondant ici à une exécution correcte. Le fait d'être capable de partir d'un état arbitraire peut sembler incongru (puisque les éléments sont *a priori* toujours démarrés dans un état bien connu), mais des travaux [71] ont montré que si un système réparti est sujet à des défaillances de nœuds de type arrêt et redémarrage (ce qui correspond à une panne franche suivie d'une réinitialisation), et que les communication peuvent ne pas être totalement fiables (des communications peuvent être perdues, dupliquées, déséquencées), alors un état arbitraire du système peut effectivement être atteint. Même si la probabilité d'une exécution menant à cet état arbitraire est négligeable dans des conditions normales, il n'est pas impossible qu'une attaque sur le système tente de reproduire une telle exécution. Dans tous les cas, et quelle que soit la nature de ce qui a provoqué l'atteinte de cet état arbitraire, un algorithme auto-sta-

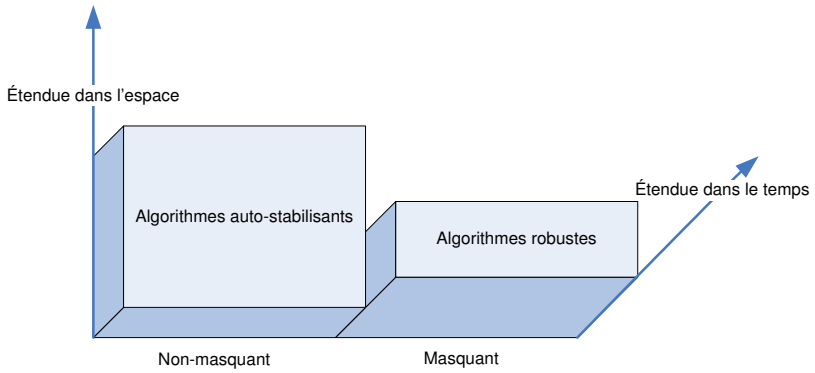

FIG. 2.2 – Classes d'algorithmes répartis tolérants aux pannes

bilisant est capable de fournir un comportement correct au bout d'un temps fini. D'ailleurs, les algorithmes répartis auto-stabilisants sont présents dans nombre de protocoles utilisés dans les réseaux d'ordinateurs **[53]**.

La figure 2.2 résume les capacités relatives des algorithmes robustes et des algorithmes auto-stabilisants. Il faut garder à l'esprit qu'aucune de ces classes ne peut, en utilisant les hypothèses de base, être développée davantage : un algorithme auto-stabilisant ne tolère pas des défaillances qui se produisent continuellement, et un algorithme robuste ne peut *a priori* pas tolérer des pannes très étendues. Par suite, aucune solution générale à des défaillances continuelles et étendues ne peut exister.

Chapitre 3

Les limites et problèmes dûs à la grande échelle

3.1 Hypothèses sur le système

Les algorithmes robustes font généralement des hypothèses qui perdent de leur pertinence dans les systèmes à grande échelle, parmi lesquelles :

communications complètes : dans de nombreux algorithmes robustes, un nœud est en mesure de parler à tout autre nœud, même en dépit de fautes des autres nœuds. Ceci revient, lorsqu'on modélise les capacités de communication par un graphe, à considérer que ce graphe est complet. Dans un réseau local où toutes les machines sont directement connectées, cette hypothèse est valide, mais dans un système comportant plusieurs dizaines (ou même centaines) de milliers de machines, elle devient dans le meilleur des cas inefficace (la latence augmente pour traverser le système), et sinon fantaisiste (la communication qui s'effectue *via* des nœuds défaillants ne peut plus avoir lieu).

Récemment, il a été proposé de résoudres des problèmes classiques (comme celui du consensus) avec des connaissances seulement partielles sur les participants. En particulier, la notion de *détecteur de participation* a été introduite dans [15] pour abstraire la notion de participant connu (par exemple car un message portant sa signature nous est parvenu). Par la suite, en considérant le graphe induit par les connaissances des participants, s'il existe un unique sous-ensemble des participants (se connaissant tous entre eux) qui peut finir par être connu de tous les participants, alors le problème du consensus peut être résolu sans connaissances complètes initiales.

communications globales : la plupart des solutions existantes nécessitent pour chaque phase (le nombre de phases total dépendant du nombre

de fautes que l'on souhaite pouvoir tolérer) un nombre quadratique de communications (en fonction de la taille du système), ce qui compromet son passage à l'échelle. En effet, lors d'une phase, un nœud envoie typiquement un message à chaque autre nœud du système.

communications synchrones : un résultat fondamental de la littérature des algorithmes répartis robustes [32] stipule que même en considérant un problème *a priori* simple (le *consensus*, où les nœuds du système doivent se mettre d'accord sur une valeur proposée par au moins l'un d'entre eux), et en considérant un modèle de fautes très simple (une seule faute peut survenir, et elle est de type crash), il est impossible de résoudre le problème dans un environnement *asynchrone* (où il n'existe pas de borne sur les vitesses relatives des nœuds du système). Ce résultat vient du fait que d'une part, la procédure de décision peut, dans certaines exécutions, dépendre de la décision communiquée par un seul nœud du système, et d'autre part que dans un système totalement asynchrone, il est impossible de distinguer un nœud en panne d'un nœud très lent. Or, un nœud victime d'une panne crash ne communiquera plus jamais, alors qu'un nœud très lent finira par envoyer son message. Si une décision est prise en pensant qu'il est fautif, alors s'il est très lent sa décision (prise avant l'envoi du message) peut être contraire à celle prise par les autres nœuds du système.

Ce résultat d'impossibilité a conduit les recherches vers des environnements synchrones ou partiellement synchrones (où des bornes, connues ou non des nœuds eux-mêmes, existent sur les vitesses relatives des nœuds du système). Une manière de formaliser les hypothèses sur le synchronisme dans lequel on se place est d'utiliser la notion de *détecteur de défaillances*. Un tel détecteur est un oracle distribué, interrogé par les nœuds du système pour obtenir des informations sur les nœuds fautifs. Plus le système est synchrone, et plus le détecteur est fiable, et plus il est facile de résoudre un problème en tolérant des pannes. Inversement, plus les hypothèses de synchronisme sont fortes, et plus il est difficile de les justifier lorsqu'on augmente le nombre de nœuds du système. Une classification des différents détecteurs de défaillances peut être trouvée dans [66, 33].

Si les systèmes réels sont dans la plupart des cas, au moins partiellement synchrones (ce qui revient à dire qu'il existe une borne, connue ou non, sur le rapport entre les vitesses relatives des nœuds du système). Par contre, les hypothèses de communications complètes et globales sont trop fortes pour être mises en pratique dans des systèmes à grande échelle.

Dans le cadre de l'auto-stabilitation, les hypothèses faites sur le systèmes ne comportent en général pas, comme pour les algorithmes robustes, de conditions sur la complétude ou la globalité des communications. De

nombreux algorithmes s'exécutent sur des systèmes dont les nœuds communiquent de manière locale uniquement. Par contre, plusieurs hypothèses peuvent être cruciales pour le bon fonctionnement de l'algorithme, et ont trait aux hypothèses faites sur l'ordonnancement du système :

atomicité des communications : la plupart des algorithmes auto-stabilisants dans la littérature utilisent des primitives de communication de haut niveau d'atomicité. On trouve dans la littérature au moins trois modèles historiques :

1. *le modèle à états (ou à mémoire partagée)* : en une étape atomique, un nœud peut lire l'état de chacun des nœuds voisins, et mettre à jour son propre état ;
2. *le modèle à registres partagés* : en une étape atomique, un nœud peut lire l'état de l'un des nœuds voisins, ou mettre à jour son propre état, mais pas les deux simultanément ;
3. *le modèle à passage de message* : il s'agit du modèle classique en algorithmique distribuée où en une étape atomique, un nœud envoie un message vers l'un des nœuds voisins, ou reçoit un message de l'un des nœuds voisins, mais pas les deux simultanément.

Avec l'étude récente de la propriété d'auto-stabilisation dans les réseaux de capteurs sans fils et ad hoc, plusieurs modèles de diffusion locale avec collisions potentielles sont apparus. Dans le modèle qui présente le plus haut degré d'atomicité [56], en une étape atomique un nœud peut lire son propre état et écrire partiellement l'état de chacun des nœuds voisins. Si deux nœuds écrivent simultanément l'état d'un voisin commun, une collision se produit et aucune des écritures n'a lieu. Un modèle plus réaliste **[45]** consiste à définir deux actions distinctes et atomiques pour la diffusion locale d'une part et la réception d'un message localement diffusé d'autre part.

Quand les communications sont bidirectionnelles, il est possible de simuler un modèle par un autre. Par exemple, [23] montre comment transformer le modèle à mémoire partagé en un modèle à registres partagés, puis comment transformer le modèle à registres partagés en un modèle à passage de messages. Dans les modèles spécifiques aux réseaux sans fils, [57] montre comment transformer le modèle à diffusion locale avec collision en un modèle à mémoire partagée ; de manière similaire, [43] montre que le modèle de **[45]** peut être transformé en un modèle à mémoire partagée. Le problème de ces transformations est double :

1. *la transformation consomme des ressources (temps, mémoire, énergie dans le cas des capteurs)* : ce qui pourrait être évité par une solution directe dans le modèle le plus proche du système considéré ;

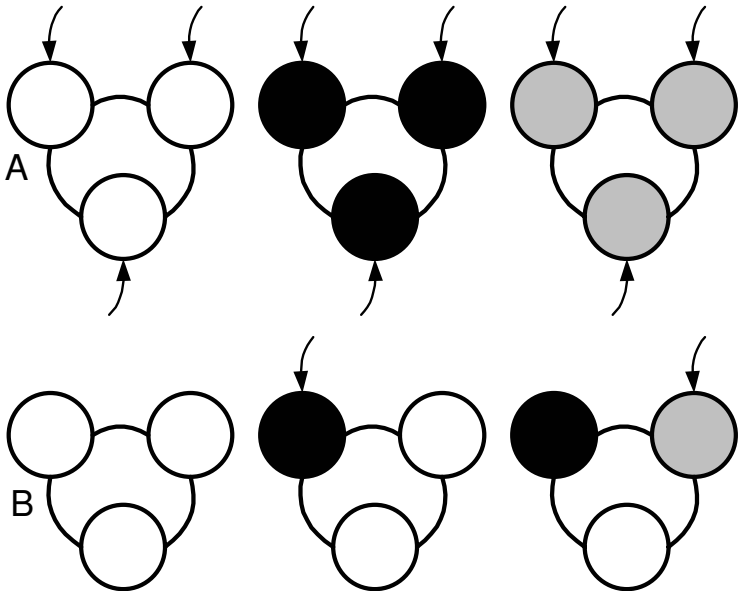

FIG. 3.1 – Coloration auto-stabilisante des nœuds

2. *la transformation n'est possible que dans les systèmes où les communications sont bidirectionnelles* : ceci est dû au fait que des acquittements doivent être envoyés régulièrement pour assurer que l'on simule correctement le modèle de plus haut niveau.

ordonnancement spatial : historiquement, les algorithmes auto-stabilisants faisaient l'hypothèse que deux nœuds voisins ne pouvaient exécuter leur code de manière simultanée. Ceci permet par exemple de briser des problèmes de symétrie dans certaines configurations. On distingue généralement trois possibilités principales pour l'ordonnancement, suivant les contraintes que l'on souhaite :

1. *l'ordonnancement central* : à un instant donné, un seul des nœuds du système peut exécuter son code ;
2. *l'ordonnancement global (ou synchrone)* : à un instant donné, tous les nœuds du système exécutent leur code ;
3. *l'ordonnancement distribué* : à un instant donné, un sous-ensemble arbitraire des nœuds du système exécute son code. Ce type d'ordonnancement spatial est le plus réaliste.

D'autres variantes sont également possibles (par exemple un ordonnancement localement central : à un instant donné, dans chaque voisinage, un seul des nœuds exécute son code) mais en pratique ils reviennent à l'un des trois modèles ci dessus (voir **[70]**). Plus le modèle d'ordonnancement spatial est contraint, il plus il est facile de résoudre des problèmes.

Par exemple, si on considère le problème du coloriage des nœuds d'un graphe (le but est d'attribuer à chaque nœud du graphe une couleur de telle sorte que deux nœuds voisins aient des couleurs distinctes), [6] montre qu'il est impossible de colorier un graphe arbitraire de manière répartie et déterministe (voir l'exécution A de la figure 3.1), principalement pour des raisons d'état initial symmétrique. Par contre, **[40]** montre que si l'ordonnancement spatial est localement central, alors une telle solution est possible (voir l'exécution B de la figure 3.1).

Certains algorithmes, qui font l'hypothèse de l'un de ces modèles, peuvent être exécutés dans un autre modèle, au prix comme précédemment d'un consommation accrue de ressources. Le modèle le plus général étant le modèle distribué, il peut être transformé en un modèle plus contraint par l'utilisation d'un algorithme d'exclusion mutuelle [22] (pour le modèle central) ou par un algorithme de synchronisation **[4]** (pour le modèle global).

ordonnancement temporel : Les premiers algorithmes auto-stabilisants présentés dans [22] sont indépendants de la notion de temps, c'est-à-dire qu'ils sont écrits dans un modèle purement asynchrone, où aucune hypothèse n'est faite sur les vitesses relatives des nœuds du système. Par la suite, des modèles d'ordonnancement plus contraints sont apparus, en particulier pour rendre compte des systèmes réels. On distingue généralement trois ordonnancements principaux :

1. *l'ordonnancement arbitraire* : aucune hypothèse n'est faite quant aux priorités d'exécution respectives des nœuds du système, sinon la simple progression (à chaque instant, au moins un nœud exécute des actions) ;
2. *l'ordonnancement équitable* : chaque nœud exécute des actions locales infiniment souvent ;
3. *l'ordonnancement borné* : entre l'exécution de deux actions d'un même nœud du système, chaque autre nœud exécute un nombre borné d'actions.

L'ordonnancement borné peut être contraint plus avant pour aboutir à un ordonnancement synchrone (ou global). Comme pour les variantes des modèles précédents, il existe des algorithmes pour transformer l'exécution d'un modèle vers un autre. Par exemple, les alternateurs [39] et **[52]** permettent de construire un modèle borné à partir d'un modèle équitable ou arbitraire. Par contre, du fait de son caractère non borné, le modèle équitable strict ne peut être construit à partir du modèle arbitraire.

3.2 Hypothèses sur les applications

Dans le cadre de l'auto-stabilisation, suivant le problème que l'on souhaite résoudre, le temps minimal nécessaire pour rejoindre une configuration correcte varie fortement. On considère généralement deux types principaux de problèmes :

les problèmes statiques : on souhaite effectuer une tâche qui consiste à calculer une fonction qui dépend du système dans lequel on l'évalue. Par exemple, il peut s'agir de colorier les nœuds d'un réseau de telle sorte que deux nœuds adjacents n'ont pas la même couleur. Un autre exemple est celui du calcul d'un arbre des plus courts chemins vers une destination : un nœud particulier, la destination, est distingué ; à chaque arête du graphe est associée un coût (qui peut représenter la latence, le coût financier, etc.) ; et chaque nœud du graphe doit obtenir en fin de compte le nom du voisin (son père dans l'arbre) qui lui permet d'arriver à la destination en minimisant le coût associé au chemin.

les problèmes dynamiques : on souhaite effectuer une tâche qui rend un service à d'autres algorithmes. Les protocoles de transformations de modèle comme ceux mentionnés dans [23] ou encore le problème du passage de jeton (décrit dans la section 3.2.2) entrent dans cette catégorie.

Du point de vue de l'aspect grande échelle, le point crucial réside dans la *localité* de la définition du problème. Par exemple, le problème de la coloration est un problème local : si chaque nœud est assuré localement que chacun de ses voisins a une couleur différente de la sienne, alors tous les nœuds du système sont également assurés de cette propriété. Par contre, le problème de trouver un arbre vers une destination (étudié dans la section 3.2.1) n'est pas un problème local : chaque nœud (sauf la destination) dispose simplement d'un pointeur vers un de ses voisins (son père dans l'arbre), mais n'a aucun moyen de savoir si la structure générale induite par les voisins ainsi choisis induit bien un arbre vers la destination (il peut très bien s'agir d'une racine isolée et d'un anneau orienté entre les autres nœuds). Pour les tâches dynamiques, le problème du passage de jeton entre les nœuds d'un réseau (détaillé dans la section 3.2.2) est également un problème global : si un nœud ne détecte pas de jeton dans son voisinage immédiat, il ne peut pas en conclure qu'il n'en existe pas dans le réseau ; s'il en possède un, il n'a aucun moyen de savoir qu'il n'en existe pas un autre.

3.2.1 Problèmes globaux statiques

Historiquement, la recherche de solutions auto-stabilisantes à des problèmes globaux statiques sur des graphes généraux a plutôt traité des ré-

seaux non-orientés où les communications sont bidirectionelles et s'effectuent au moyen de registres partagés (voir [23]). Ce modèle permet d'écrire les algorithmes et les preuves d'une manière élégante et concise. Pour implanter effectivement de tels protocoles dans les systèmes réels, où les processeurs communiquent par échange de messages, des transformateurs préservant la propriété des algorithmes originaux sont nécessaires. De tels transformateurs sont présentés dans[2, 23], et sont basés sur des variantes du protocole du bit alterné ou de la fenêtre glissante. Toutefois, l'utilisation de tels transformateurs implique que les récepteurs sont capables d'envoyer des acquittements de manière périodique, et les liens du réseau doivent donc être bidirectionnels. De plus, ces transformateurs font l'hypothèse que les processeurs possèdent des informations sur leur voisinage (*i.e.* ils connaissent l'identité de leurs voisins).

Par conséquent, dans les réseaux unidirectionnels, les transformateurs basés sur les acquittements ne peuvent pas être utilisés pour exécuter des algorithmes auto-stabilisants communicant par passage de messages. En effet, deux voisins pourraient n'être reliés que par un lien unidirectionnel. De surcroît, dans les réseaux unidirectionnels à passage de messages, il est généralement facile d'obtenir la liste de ses voisins entrants (en vérifiant qui a «récemment» envoyé un message), mais il est très difficile (voire impossible) de maintenir l'ensemble de ses voisins sortants (dans un réseau satellite ou un réseau de capteurs, un émetteur n'est en général pas capable de savoir qui écoute l'information qu'il communique).

La particularité des hypothèses système et le manque de transformateurs génériques a conduit à la conception d'algorithmes auto-stabilisants spécifiques pour les réseaux unidirectionnels [1, 16, 26] et **[21, 30, 31, 20]**. Les solutions [1, 16, 26] et **[21]** sont «classiques» au sens où une chouche d'auto-stabilisation est ajout à un algorithme non stabilisant bien connu pour assurer la stabilisation. Ceci induit typiquement un surcoût potentiel (en temps, en mémoire, en connaisances sur le réseau). A l'inverse, les approches proposées dans **[30, 31, 20]** sont basées sur des *conditions* portant sur les algorithmes locaux : soit la condition est satisfaite (et l'algorithme est auto-stabilisant), soit la condition n'est pas satisfaite (et l'algorithme ne stabilise pas). Par suite, aucun surcoût n'est induit par l'ajout de la propriété d'auto-stabilisation à l'algorithme (l'algorithme d'origine n'est pas modifié).

De nombreux problèmes globaux statiques se ramènent à la construction d'un arbre (ou d'une forêt), suivant une métrique particulière. On obtient suivant les cas un arbre en largeur, en profondeur, ou qui minimise (ou maximise) un ou plusieurs critères particuliers. L'algorithme réparti peut alors être réduit à l'exécution sur chaque nœud d'un opérateur spécifique au problème à traiter [29]. Quand cet opérateur vérifie certaines propriétés **[30, 31, 20]**, l'algorithme dérivé est auto-stabilisant. Cette manière de procéder permet ensuite de faciliter grandement la production de solu-

tion auto-stabilisantes : elle est générique (suivant l'opérateur, le graphe de communication, l'ordonnancement) et permet lors de l'écriture de la preuve de juste vérifier que les propriétés attendues sont bien satisfaites par l'opérateur. En outre, cette solution peut être utilisée dans des réseaux ou les communications ne sont pas fiables (pertes, duplications, déséquencements) et règle donc de manière uniforme certains problèmes caractéristiques des réseaux sans fils (voir également sections 4.3.1 et 5.2).

Les solutions de **[30, 31]** sont génériques mais s'exécutent dans un modèle à mémoire partagée unidirectionnel. Dans **[31]**, l'atomicité des communications est composée : en une étape atomique, un processeur est capable de lire le véritable état de tous ses voisins et de modifier son propre état. Dans **[30]**, l'atomicité est de type lecture-écriture : en une étape atomique, un processeur est capable de lire l'état de l'un de ses voisins, ou de mettre à jour son propre état, mais pas les deux. Aucune de ces approches ne peut être transformée par l'un des transformateurs précités pour s'exécuter dans des réseaux à communication par passage de messages. Les solutions [16, 26] et **[21]** sont spécifiques (un seul problème particulier est considéré, le routage dans [16], la communication de groupe dans [26]), le recensement dans **[21]**, mais s'exécutent dans des réseaux à passage de messages unidirectionnels. Alors que [16, 26] supposent des communication fiables, **[21]** tolère des pertes, duplications, et déséquencements de messages. La solution de [1] propose une solution générique dans le modèle à passage de messages, mais suppose que les communications sont fiables (avec des liens FIFO), que les liens ont des identifiants uniques, et que le réseau est fortement connexe, trois hypothèses que **[30, 31, 20]** ne font pas.

Dans **[20]** est proposé un algorithme générique qui peut être instancié pour résoudre des tâches statiques, et qui s'exécute dans des réseaux unidirectionnels où les processeurs communiquent par échange de messages. La solution de **[20]** est auto-stabilisante (elle retrouve en un temps fini un comportement correct depuis n'importe quel état initial). Elle tolère également la perte équitable des messages, la duplication finie, le déséquencement arbitraire à la fois dans la phase de stabilisation et dans la phase stabilisée. En outre, cette approche présente plusieurs aspects intéressants : le réseau n'a pas besoin d'être fortement connexe, les processeurs n'ont pas besoin de savoir si le réseau comprend des cycles, et aucune borne sur la taille, le diamètre ou le degré maximum du réseau n'a besoin d'être connue. Toutefois, si de telles informations sont connues, le temps de stabilisation peut s'en trouver fortement réduit, jusqu'au diamètre effectif du réseau.

La table 3.1 résume les différences principales entre les travaux cités en considérant les critères suivants : surcoût, modèle de communication, fiabilité des communications, et nature de l'algorithme. Notons que tous ces algorithmes ont un temps de stabilisation similaire, linéaire en fonction du diamètre du réseau, et sont donc optimaux concernant ce critère (voir

Référence	Surcoût	Communication	Fiabilité des communications	Algorithme
[1]	oui	passage de messages	fiable	generique
[16]	oui	passage de messages	fiable	spécifique
[30]	non	registres partagés	fiable	générique
[21]	oui	passage de messages	non-fiable	spécifique
[31]	non	mémoire partagée	fiable	générique
[26]	oui	passage de messages	fiable	spécifique
[20]	non	passage de messages	non-fiable	générique

TAB. 3.1 – Auto-stabilisation des problèmes statiques dans les réseaux unidirectionnels

section 3.2.3).

3.2.2 Problèmes globaux dynamiques

L'«étalon» des algorithmes auto-stabilisants pour des problèmes dynamiques (et le premier algorithme publié) est celui de l'exclusion mutuelle sur un anneau unidirectionnel. Le problème de l'exclusion mutuelle est un problème fondamental dans le domaine de l'informatique répartie [65, 5]. Considérons un système réparti de n processeurs. Tous les processeurs, de temps à autre, peuvent avoir à exécuter une section critique de leur code durant laquelle exactement un processeur est autorisé à utiliser une ressource partagée. Un système réparti qui résoud le problème de l'exclusion mutuelle doit garantir les deux propriétés suivantes : *(i) exclusion mutuelle* : exactement un processeur est autorisé à exécuter sa section critique à un instant donné ; *(ii) équité* : tout processeur doit être en mesure d'exécuter sa section critique infiniment souvent au cours de l'exécution. Une technique classique consiste à faire passer à chaque processeur un message spécial appelé *jeton*. La réception du jeton signifie que le processeur peut entrer en section critique. Ce jeton doit permettre le respect des contraintes de l'exclusion mutuelle : être présent en un unique exemplaire et passer infiniment souvent par chaque processeur.

Intuitivement, un protocole d'exclusion mutuelle auto-stabilisant par passage de jeton garantit que, même si on part d'un état où la spécification de l'exclusion mutuelle n'est pas respectée (zéro ou plusieurs jetons sont présents dans le système), alors en un nombre fini d'étapes, un seul jeton circule équitablement dans le réseau. En pratique, quand les nœuds communiquent par passage de messages, on se borne à prouver qu'à partir d'une configuration à plusieurs jetons, on aboutit en un temps fini à une configuration à jeton unique. En effet, [55] ont prouvé que dans le cas où on se trouve dans un système réparti où les processeurs communiquent par passage de messages, il est indispensable de disposer d'un mécanisme de temporisation (*timeout*) pour injecter des jetons spontanément : si un tel mécanisme n'est pas disponible, le système ne peut être auto-stabilisant

puisqu'il se retrouverait bloqué en démarrant d'une configuration initiale sans messages.

Un réseau est *uniforme* si tout processeur exécute le même code, et il est *anonyme* si les processeurs ne disposent par d'identificateurs pour exécuter des sections de code différentes. Bien sûr, si un protocole fonctionne dans un réseau uniforme et anonyme, alors il fonctionne *a fortiori* dans un réseau non uniforme ou non anonyme. Les protocoles uniformes et anonymes sont ceux qui offrent le plus de flexibilité lors d'un déploiment dans un réseau (et plus particulièrement dans un réseau de grande taille) en raison de l'absence de vérifications supplémentaires (identifiant uniques sur le réseau, code différent déployé de manière adéquate, etc.). Depuis les trois protocoles d'exclusion mutuelle auto-stabiliante par passage de jeton proposés dans l'article fondateur [22], de nombreux travaux ont traité de ce problème dans différents contextes et par exemple [42, 19, 10, 13, 67, 51] et **[18]** dans le cas des anneaux de processeurs unidirectionnels anonymes et uniformes.

Un protocole d'exclusion mutuelle par passage de jeton est *transparent* vis à vis de l'application qui utilise le protocole si il ne modifie pas le format des jetons qui sont échangés par l'application. Une telle propriété est souhaitable si par exemple le contenu du jeton est utilisé par l'application (c'est le cas dans un réseau de type *Token Ring* ou *FDDI*, où le jeton contient également les informations devant être transmises au destinataire). En effet, un protocole transparent est plus facile à implanter (il ne modifie pas le format des trames de l'application) et plus facile à intégrer à des réseaux hétérogènes (dont certaines parties utilisent un protocole de passage de jeton différent). En outre, la charge de la vérification de l'intégrité des messages peut être déléguée entièrement à l'application. Parmi les protocoles précités, seuls [42, 10] et le protocole synchrone de **[18]** sont transparents vis à vis de l'application qui utilise le protocole. Les protocoles présentés dans [13, 67, 51] et **[18]** utilisent soit plusieurs types de jetons (et donc ajoutent un champ *type* aux messages de l'application), soit des informations supplémentaires à chaque jeton circulant de manière à assurer la stabilisation.

Comme montré dans [14], il est impossible de résoudre le problème de l'exclusion mutuelle auto-stabilisante dans un anneau unidirectionnel anonyme et uniforme au moyen d'un protocole déterministe quand la taille n'est pas première. Aussi les solutions précédentes sont elles toutes probabilistes. Parmi celles-ci, [42, 10] ne proposent pas de calcul du temps de stabilisation, et les temps de stabilisation moyens attendus de [13, 51] et **[18]** sont de l'ordre de n^3, où n désigne la taille de l'anneau, et celui de [67] est de l'ordre de $n^2 \log(n)$. Dans **[28]**, il est prouvé que le temps de stabilisation de [42, 10] est de $\Theta(n^2)$ avec un ordonnancement synchrone.

Un autre critère d'évaluation est celui du *temps de service*, c'est à dire le temps, en phase stabilisée, entre deux passages du jeton sur un proces-

Protocole	Connaissance de n	Temps de stabilisation	Temps de service	Mémoire	Transparent
[13]	oui	$\Theta(n^3)$	$O(n^3)$	$O(\log(n))$	non
[67]	non	$\Theta(n^2 \log(n))$	$O(n^2 \log(n))$	$O(1)$	non
[51]	oui	$O(n^3)$	$O(n)$	$O(\log(n))$	non
[42, 10],**[28]**	non	$\Theta(n^2)$	$\Theta(n^2)$	0	oui
[18, 28]	non	$\Theta(n^2)$	$\Theta(n)$	$O(1)$	oui
[28]	oui	$\Theta(n)$	$\Theta(n)$	$O(\log(n))$	oui
[28]	oui	$\Theta(n)$	$\Theta(n)$	$O(1)$	oui

TAB. 3.2 – Algorithmes de passage de jeton auto-stabilisants

seur. Ce temps est important pour évaluer les performances du protocole quand il n'y a pas de défaillances et ainsi évaluer son surcoût par rapport à un protocole non stabilisant. Dans un système à n processeurs, le temps de service est en $\Omega(n)$, puisque si chaque processeur attend le minimum de temps, il attend autant que les autres. Le temps de service n'est pas calculé dans [42, 10, 13], et il est dans [67, 51] et **[18]** de l'ordre de n^3, $n^2 \log n$, et n, respectivement. Dans **[28]**, il est prouvé que le temps de service de [42, 10] est de $\Theta(n^2)$ avec un ordonnancement synchrone. Notons que **[18]** (respectivement **[41]**) a été le premier a proposer un algorithme d'exclusion mutuelle (respectivement de l-exclusion mutuelle) qui garantit un temps de service borné sous un ordonnancement arbitraire.

Dans **[28]** sont proposés plusieurs protocoles auto-stabilisants pour les réseaux synchrones anonymes et uniformes en anneau unidirectionel où les processeurs communiquent par échange de messages. Les deux premiers protocoles sont des transpositions dans un modèle à passage de messages des protocoles de [42, 10] d'une part, et de **[18]** d'autre part, qui utilisent dans leur version d'origine un modèle à mémoire partagée. Ensuite, **[28]** propose deux protocoles basés sur la notion de *ralentisseur*. Chaque processeur peut se proclamer ralentisseur et ralentir les jetons qu'il reçoît avec une certaine probabilité. Suivant la puissance du ralentisseur considéré (quantité de mémoire disponible), les résultats de complexités obtenus sont différents, mais chacun des protocoles a un temps de stabilisation moyen et un temps de service moyen en $O(n)$.

Les résultats enoncés précédemment sont reportés sur la table 3.2. Comme indiqué dans la section 3.2.3, le problème de l'exclusion mutuelle par passage de jeton dans le modèle proposé est en $\Omega(n)$. Le dernier algorithme présenté dans **[28]** est donc optimal pour tous les critères considérés (temps de stabilisation, temps de service, mémoire utilisée, et transparence). En outre, il peut être exprimé simplement :

1. chaque nœud possède un état qui peut prendre deux valeurs, *normal* et *ralentisseur* ;
2. un nœud normal qui reçoit un jeton retransmet le jeton a son succes-

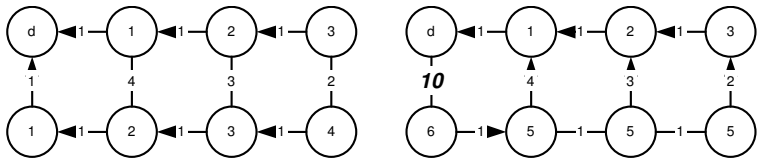

FIG. 3.2 – Construction d'un arbre des plus courts chemin vers d

seur dans l'anneau ;

3. un nœud ralentisseur qui reçoit un jeton garde le jeton ;
4. à chaque unité de temps (le système est supposé synchrone), chaque nœud normal devient ralentisseur avec probabilité $\frac{1}{n(n-1)}$, sinon (s'il est ralentisseur) il devient normal avec probabilité $\frac{1}{n}$.

Sur n étapes synchrones, il y a une probabilité constante d'obtenir un seul ralentisseur, et que ce ralentisseur reste le même pendant n étapes. Alors, tous les jetons se retrouvent sur le même nœud ralentisseur, et sont fusionnés au cours des n étapes. Par la suite, un jeton fait le tour de l'anneau en moyenne en $2n$ étapes (il reste en moyenne n étapes sur un unique ralentisseur, et une étape sur tous les autres nœuds).

3.2.3 Optimalité et passage à l'échelle

Les problèmes globaux présentent un problème de performance lorsqu'on s'intéresse au passage à l'échelle. Par exemple, la figure 3.2 montre deux configurations d'un algorithme auto-stabilisant de construction d'arbre des plus courts chemins vers une destination. Entre les deux configurations, seul le poids d'une arête (marquée en gras) a changé. Cependant, cette unique modification a induit le changement de parent de la moitié des nœuds du réseau.

De manière duale, la figure 3.3 montre une configuration initiale d'un algorithme auto-stabilisant d'exclusion mutuelle par passage de jeton dans un anneau anonyme (les identifiants ont été placés uniquement pour distinguer les nœuds dans le texte). Quand un processus possède le jeton (ce qui est déterminé par son voisinage uniquement), il apparaît en noir. Le but de l'algorithme est de garantir qu'au bout d'un temps fini, un unique jeton circule dans le réseau. Pour des raisons de symétrie du système, il n'est pas possible de concevoir dans ce contexte un algorithme qui supprimerait le jeton placé en $n/2 + 1$ mais pas celui placé en 1. Le seul moyen pour faire décroître le nombre de jeton consiste à les déplacer sur l'anneau de telle sorte qu'ils se rejoignent. Étant donné que la distance qui sépare les deux jetons initialement est de l'ordre de n – la taille du réseau –, et que le déplacement d'un jeton requiert l'action d'au moins un nœud, il faut au moins

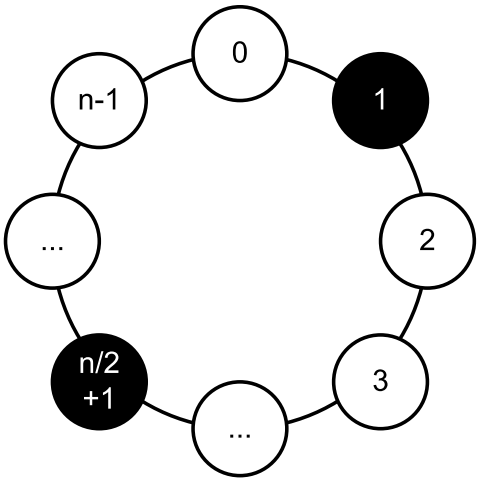

FIG. 3.3 – Exclusion mutuelle par passage de jeton auto-stabilisante

qu'un nombre de nœuds proportionnel à n agisse pour que les jetons se rejoignent.

Pour des réseaux de taille modeste (comprenant quelques dizaines ou quelques centaines de nœuds), il reste envisageable de tenter de résoudre des problèmes globaux. Pour des réseaux de grande taille (plusieurs dizaines ou centaines de milliers de nœuds), des algorithmes auto-stabilisants résolvant des problèmes globaux, même s'ils sont optimaux comme ceux développés dans les sections 3.2.1 et 3.2.2, ne peuvent plus être envisagés.

Chapitre 4

Solutions pour l'auto-stabilisation à grande échelle

L'auto-stabilisation, dans sa forme originelle, n'est pas adaptée aux systèmes à grande échelle. Toutefois, en restreignant certains aspects de la définition de base, il est possible de conserver des propriétés de tolérance aux pannes intéressantes pour les systèmes à grande échelle. Ces restrictions consistent à restreindre l'étendue ou la nature des fautes qui sont considérées pour permettre un retour rapide à la normale, ou encore le type de problèmes que l'on se propose de résoudre. Pour certaines approches, la propriété d'auto-stabilisation reste conservée, et des propriétés additionelles, utiles dans les systèmes à grande échelle, sont ajoutées.

4.1 Restreindre la nature des fautes

4.1.1 Détection et correction d'erreurs

Le moyen le plus simple pour ajouter la propriété d'auto-stabilisation à un système est d'utiliser un mécanisme de détection et de correction des fautes. Les approches historiques [55, 8] sont globales (au moins un nœud reçoit des informations de chacun des autres, ou envoie des informations à chacun des autres) et ne sont pas adaptées aux systèmes à grande échelle. Toutefois, plusieurs approches récentes sont envisageables pour de tels systèmes :

détecteurs et correcteurs localisés : Toutes les tâches ne sont pas équivalentes lorsqu'il s'agit de détecter qu'une corruption de mémoire a eu lieu. Par exemple, pour se rendre compte qu'un coloration des nœuds est incorrecte, il suffit de regarder les couleurs de ses voisins et de les

comparer avec la sienne. Chaque nœud détectant un conflit peut demander une action de correction. Par contre, pour s'assurer qu'une orientation du réseau est sans cycle, il faut potentiellement regarder à une distance proportionnelle à la taille du système (voir **[11]**).

Bien entendu, un algorithme particulier peut ajouter des variables supplémentaires pour permettre une détection plus rapide (par exemple la distance à la racine pour la construction d'un arbre, chaque nœud est alors en mesure de vérifier que son «père» dans l'arbre est bien à une distance inférieure à la sienne). C'est sur ce principe qu'est construit le *stabiliseur local* de [3]. En parallèle avec l'exécution normale du système, les nœuds surveillent l'état de leur voisinage, à une distance qui dépend du problème et de l'algorithme utilisé. Cette surveillance permet de détecter certaines corruptions de mémoire, et de déclencher une opération de correction adaptée. La phase de correction utilise la redondance des informations utilisées pour la détection : chaque nœud possède une copie de l'état de chacun de ses voisins à une certaine distance k. Cette redondance réserve la technique à des algorithmes pour lesquels cette distance est petite, en effet la mémoire et les échanges associés croissent de manière exponentielle en fonction de cette distance.

détecteurs et correcteurs probablement corrects : Cette approche consiste à considérer que des corruptions de mémoire réellement arbitraires sont hautement improbables. Des arguments probabilistes sont utilisés pour établir que en général, les corruptions de mémoire qui résultent des fautes survenues peuvent être détectées au moyen de techniques classiques en théorie de l'information, comme la redondance des données ou les codes de détection et de correction d'erreurs. En particulier, dans [44], on utilise des codes de détection d'erreur pour déterminer qu'une corruption de mémoire est survenue, avec grande probabilité. Si l'article considère uniquement le cas où une seule corruption survient (c'est-à-dire qu'un seul nœud du système est touché par cette corruption), il permet de retrouver un comportement correct en une seule étape de correction. Pour un système, même à grande échelle, où les corruptions de mémoire sont localisées dans chaque voisinage et ne sont pas malicieuses (c'est-à-dire qu'elles sont détectables par des techniques comme des codes de redondance cycliques), cette approche apparaît indiquée.

Une approche similaire a été suivie récemment dans [49]. Le principe de base est le suivant : à chaque variable de l'algorithme de départ, on associe k variables (3 dans l'article mentionné). Ensuite, à chaque accès à une variable particulière (c'est-à-dire en fait un ensemble de k variables représentant une variable logique), on utilise une fonction de parité qui permet de distinguer les corruptions visibles des ab-

sences de corruptions (ou des corruptions invisibles). Une corruption visible correspond à une détection par le code de parité. A chaque fois qu'une corruption visible est détectée, on utilise une fonction de majorité bit à bit pour rétablir la valeur de la variable avant que la corruption ne survienne. Cette approche a pour défaut d'augmenter la mémoire et les traitements sur un nœud particulier d'un facteur k. Par contre, la simplicité de sa mise en œuvre rend son implantation facile sur des nœuds peu puissants (comme des réseaux de capteurs).

4.1.2 Préservation de prédicats

Un système auto-stabilisant n'a pas besoin d'être initialisé. De plus, quand les paramètres ou l'environnement changent, il est en mesure de s'adapter au nouveau contexte sans qu'il y ait besoin d'écrire du code spécifique pour traiter les cas non prévus lors de la conception du système. Cette généralité dans l'approche de la tolérance aux pannes et de l'adaptativité est indubitablement un point fort de l'auto-stabilisation, mais dans un système à grande échelle, les aspects de dynamicité et de changement inopiné de l'environnement sont beaucoup plus susceptibles de se produire que des corruptions arbitraires de la mémoire des nœuds du système.

Plusieurs travaux récents dans le domaine de l'auto-stabilisation se concentrent sur des solutions plus robustes que les solutions simplement stabilisantes dans des contextes fortement dynamiques. A la base, ces algorithmes sont auto-stabilisants, et bénéficient donc des propriétés qui en résultent. De plus, ils préservent un prédicat local quand certains changements particuliers se produisent. D'une certaine manière, ils garantissent certaines propriétés quand des défaillances restreintes (mais potentiellement fréquentes) se produisent, et garantissent simplement l'auto-stabilisation quand des défaillances arbitraires interviennent. On peut dénombrer plusieurs approches complémentaires :

super-stabilisation : cette propriété (définie dans [25]) stipule qu'un algorithme super-stabilisant est auto-stabilisant d'une part, et préserve un prédicat (typiquement un prédicat de sûreté) quand des changements de topologie surviennent dans une configuration légitime. Ainsi, les changements de topologie sont restreints : si ces changements interviennent lors de la phase de stabilisation, le système peut ne jamais stabiliser. Par contre, s'ils interviennent seulement après qu'un état global correct a été atteint, le système reste stable. Cette propriété est strictement plus forte que la propriété d'auto-stabilisation : dans un système auto-stabilisant, les changements de topologie seraient assimilés à des défaillances transitoires (les nœuds n'ont pas en mémoire une vision correcte de leur voisinage), et aucune garantie de sûreté ne pourrait être donnée si à partir d'un état global correct, un tel changement de topologie survenait.

auto-stabilisation et communications non fiables : il s'agit ici des algorithmes à la fois auto-stabilisants et tolérant des pannes de liens (pertes, duplications, déséquencements, voir paragraphe 2.1). Si les pannes de liens sont transitoires (ou intermittentes mais se produisant rarement), l'auto-stabilisation «simple» permet de revenir à la normale. Si par contre ces pannes interviennent de manière intermittente mais régulière, le bon comportement du système n'est plus garanti. Remarquons tout d'abord que pour que le problème admette une solution, il est nécessaire que ces pannes de liens ne soient pas complètement arbitraires :

- *pertes* : si un canal peut perdre tous les messages qui transitent à travers lui, on ne peut résoudre aucun problème non trivial. On fait donc l'hypothèse que les pertes sont *équitables*, c'est-à-dire que si un nœud envoie une infinité de messages sur un lien adjacent, le lien délivre une infinité de messages au nœud situé à l'autre extrémité du lien. Bien sûr, le lien peut, ce-faisant, perdre une infinité de messages.
- *duplications* : si un canal peut dupliquer infiniment un message qui transite par ce canal, alors aucun problème non trivial ne peut être résolu de manière auto-stabilisante. En considérant que suite à une défaillance transitoire les liens de communications contiennent des messages erronés, ceux-ci peuvent être dupliqués et délivrés aux nœuds adjacents à l'infini, et compromettre l'intégrité du système indéfiniment. Aussi, on fait l'hypothèse qu'un même message ne peut être dupliqué qu'un nombre fini (mais potentiellement non borné) de fois.

Sous ces hypothèses (pertes équitables, duplication finie, déséquencement arbitraire), plusieurs solutions existent et restent auto-stabilisantes. En particulier, cela signifie que les défaillances des liens peuvent se produire pendant la phase de stabilisation, mais aussi pendant la phase stabilisée. Aussi, à partir d'une configuration légitime, les pertes, duplications et désequencements qui pourraient se produire n'ont pas d'impact sur la correction du système (elles sont donc masquées à l'utilisateur du système). Dans **[21]**, une solution au problème du recensement (trouver tous les nœuds d'un système et leurs positions respectives) présente les caractéristiques évoquées précédemment, et dans **[20]** une solution générique (c'est-à-dire paramétrable pour différentes métriques) est donnée au problème de la construction d'arborescences (voir également section 3.2.1).

Ces deux solutions présentent des caractéristiques communes pour la résistances aux différents types de défaillances des liens de communications. Pour gérer les pertes, chaque nœud réémet régulièrement son dernier message. Pour gérer les duplications, l'algorithme utilisé satisfait la propriété d'*idempotence* (c'est-à-dire que la réception succes-

sive d'un même message plusieurs fois ne change pas l'état du nœud qui le reçoit). Enfin, pour les déséquencements, l'algorithme traite chaque message de manière indépendante (ce qui conduit pour **[21]** à utiliser des messages de taille importante).

routage auto-stabilisant sans boucles : les prédicats que l'on souhaite préserver ici sont caractéristiques des protocoles de routage. L'intérêt principal de la construction des tables de routage par un algorithme réparti est de pouvoir effectuer leur mise à jour de manière dynamique, et en particulier alors que le réseau est en cours d'utilisation. Maintenant, quand une table de routage est modifiée localement sur un nœud, le chemin qu'un message particulier transitant par ce nœud est également modifié. Si on ne prend pas de précautions particulières, à un instant donné, des boucles logiques peuvent se produire quand on suit le cheminement des tables de routage en cours de mise à jour. Ces boucles logiques augmentent le nombre de sauts qu'un message particulier doit parcourir et, si ce message a une durée de vie limitée, peuvent conduire à la suppression du message. Le routage *sans boucles* permet de garantir que lors de la modification des tables de routages, à tout instant aucune boucle logique n'existe dans le système. Dans [17], une version auto-stabilisante d'un algorithme de routage sans boucles est présenté. Cependant, dans un environnement fortement dynamique, il est possible que même un algorithme de routage sans boucles se révèle insuffisant.

Sur la figure 4.1, une exécution possible d'un algorithme de routage sans boucles est représenté, avec un message qui doit être acheminé jusqu'à la destination d. Les changements des coûts sur les liens de communications occasionnent des mises à jour successives des tables de routage. A chaque instant, un nœud particulier utilise sa table de routage local, et à chaque instant, aucune boucle logique n'existe jusqu'à la destination. Cependant, la dynamicité du réseau fait qu'une boucle logique est construite au cours du temps, empêchant le message d'arriver à destination. Dans **[54]**, un algorithme auto-stabilisant sans boucles et *préservant les routes* est présenté. La propriété de préservation des routes signifie que si un arbre est initialement construit vers une destination, tout message émis vers cette destination arrive en temps fini. La technique générale pour aboutir à ce résultat se décompose en deux phases :

1. avant de changer de parent dans l'arbre qui le mène à la destination, un nœud s'assure auprès de tous ses descendants que ce changement n'occasionnera pas de boucle ;
2. les modifications de la table de routage ont une priorité inférieure à la transmission des messages, de telle sorte qu'un message se dirigeant vers la racine voit toujours décroître la distance

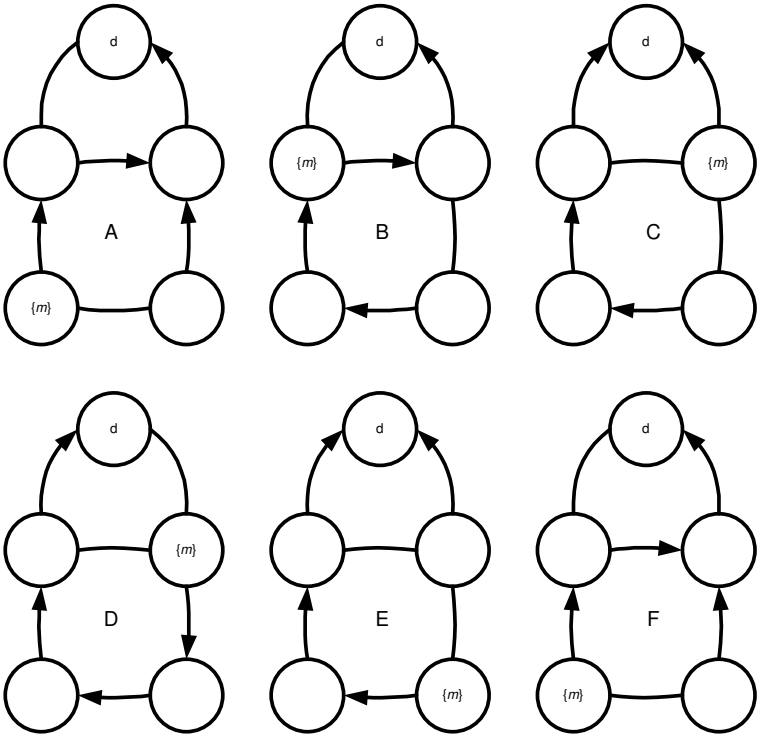

FIG. 4.1 – Routage sans boucles en environnement dynamique

(pour une métrique particulière) qui le sépare de la racine.

Ainsi, si des changements des poids des liens ont lieu après qu'un arbre a été construit, les messages arrivent toujours à destination. Si de plus il ne se produit pas de changement des poids des liens pendant une certaine période, alors le système converge vers un arbre des plus courts chemins (toujours suivant une métrique particulière) vers la destination.

4.2 Restreindre l'étendue géographique des fautes

Dans son acceptation classique, l'auto-stabilisation ne donne pas de contraintes sur le nombre de fautes qui peuvent frapper un système. Cette généralité peut ne pas être vérifiée dans un système à grande échelle, tel qu'un réseau de capteurs ou un réseau ad hoc de grande taille (plusieurs centaines de nœuds. En effet, du fait du grand nombre de nœuds présents, il est hautement probable que la grande majorité d'entre eux vont fonctionner correctement. Par contre, il est tout aussi probable que tout au long de l'exécution, plusieurs d'entre eux soient sujets à des pannes intermittentes.

En supposant que les fautes qui peuvent se produire ne concernent jamais qu'une petite partie du réseau, il est possible de concevoir des algorithmes qui convergent plus rapidement que des algorithmes auto-stabilisants classiques. Pour se donner un cadre formel, on considère que la distance à une configuration légitime est égale au nombre de nœuds dont il faut changer la mémoire pour atteindre une configuration légitime (de la même manière que pour une distance de Hamming). Bien sûr, il est possible que même si on est à distance k d'une configuration légitime, plus de k nœuds ont en fait vu leur mémoire corrompue. Du point de vue du retour à un état normal, on considère uniquement la configuration légitime la plus proche.

Les travaux qui cherchent à minimiser le temps de stabilisation dans un contexte où peu de fautes se produisent distinguent généralement deux degrés de stabilisation :

la stabilisation «visible» : ici, seules les variables de sortie de l'algorithme sont concernées. Les variables de sortie sont typiquement utilisées par l'utilisateur du système. Par exemple, si on considère un algorithme de construction d'arbre, seul le pointeur vers le nœud parent fait partie des variables de sortie.

la stabilisation «interne» : ici, toutes variables de l'algorithme sont concernées. Ce type de stabilisation correspond à la notion classique de l'auto-stabilisation.

Dans beaucoup de travaux, seule la stabilisation «visible» s'effectue rapidement (c'est-à-dire en temps relatif au nombre de fautes qui frappent

le système, plutôt qu'en un temps relatif à la taille du dit système), la stabilisation «interne» restant le plus souvent proportionnelle à la taille du réseau. Les algorithmes qui présentent cette contrainte ne sont donc pas capables de tolérer une fréquence de fautes plus élevée. Considérons en effet un algorithme dont le temps de stabilisation visible est une fonction de k (le nombre de fautes) et dont la stabilisation interne est une fonction de n (la taille du système). Maintenant, si une nouvelle faute survient alors que la stabilisation visible est effectuée mais pas la stabilisation interne, cela peut mener à un état global comprenant un nombre supérieur à k de défaillances, et aucune garantie ne peut plus être donnée sur le nouveau temps de stabilisation visible.

4.2.1 k-stabilisation

La k-stabilisation se définit comme l'auto-stabilisation, en restreignant les configurations de départ aux configurations qui sont à distance au plus k d'une configuration légitime. Du fait de l'environnement moins hostile, il est possible de résoudre des problèmes impossibles dans le cas de l'auto-stabilisation générale, et de proposer des temps de stabilisation visibles réduits, même pour des tâches globales comme celles décrites paragraphe 3.2.

Par exemple, il est notoire [50] qu'il est impossible de résoudre le problème du passage de jeton équitable de manière auto-stabilisante anonyme (les nœuds ne peuvent pas être distingués) et déterministe quand le graphe de communication est un anneau unidirectionnel (un nœud ne peut recevoir des informations que de son voisin de gauche, et ne peut envoyer des informations qu'à son voisin de droite). L'argument principal de la preuve d'impossibilité est le suivant : considérons une configuration où le système de n nœuds contient un unique jeton et est stabilisé. Sur la figure 4.2, le jeton est localisé sur le nœud dont l'état est $e2$. On construit alors un nouveau système de taille $2n$ et reproduisant les états des processeurs de manière symétrique (c'est-à-dire que le nœud i possède le même état local que le nœud $i+n$). Il existe deux jetons dans ce nouveau système, et si on exécute le code des nœuds en utilisant un ordonnancement synchrone, les deux jetons perdurent à jamais (le premier système de taille n laissant l'unique jeton perdurer à jamais).

Dans **[35]**, on considère qu'au plus $k < \frac{n}{c}$ fautes peuvent frapper le système (où c est une petite constante) ; dès lors, le problème du passage de jeton peut être résolu de manière déterministe et k-stabilisante. L'idée de base est d'ajouter une *vitesse* aux jetons. Cette vitesse est proportionnelle au nombre de nœuds correct précédant le jeton (ce nombre est calculé *via* une variable qui estime la distance au prochain jeton). Ensuite, un jeton dont les k prédécesseurs sont corrects aura une vitesse maximale, tandis qu'un jeton dont les k prédécesseurs ne sont pas tous corrects aura sa vitesse ralentie, et sera rattrapé par un jeton correct. Le temps de stabilisation visible de cet

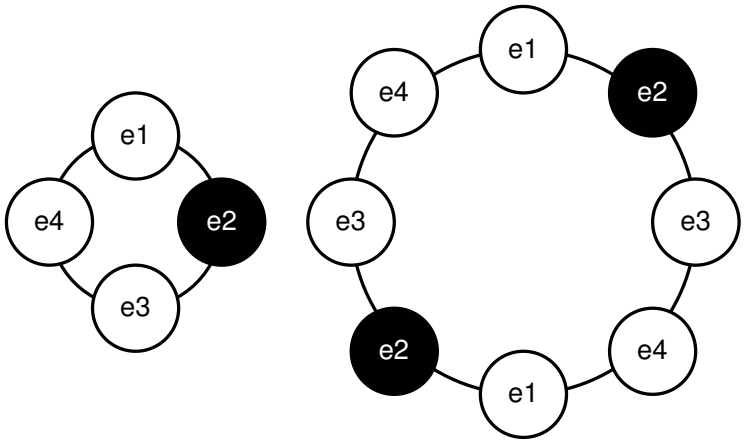

FIG. 4.2 – Impossibilité de l'exclusion mutuelle sur un anneau uniforme

algorithme est en $O(k)$.

4.2.2 Auto-stabilisation adaptative en temps

Pour certains problèmes, une corruption de mémoire peut entraîner une cascade de corrections dans tous le système [8]. Il serait pourtant naturel que lorsque moins de défaillances frappent le système, la stabilisation soit plus rapide. C'est ce principe que développe l'auto-stabilisation adaptative en temps (en anglais *time adaptive stabilization*, *scalable stabilization* ou encore *fault local stabilization*).

L'auto-stabilisation adaptative en temps a d'abord été étudiée pour les tâches statiques. Une tâche en particulier, le *bit persistent* fait l'objet de [59, 60] : il s'agit de tolérer la corruption d'un bit sur k nœuds, lorsque k est inconnu de chaque nœud. Ces deux approches sont basées sur de la collecte d'information auprès des autres nœuds pour effectuer ensuite un vote à la majorité pour établir la valeur véritable. Dans [60], le temps de stabilisation visible est de $O(k \log(n))$, pour $k \leq O(\frac{n}{\log(n)})$. Dans [59], si le nombre de fautes est inférieur à $\frac{n}{2}$, alors le temps de stabilisation visible est de $O(k)$. Ces deux algorithmes supposent un ordonnancement synchrone. Toujours pour les tâches statiques, [37] propose un algorithme qui transforme un premier algorithme A auto-stabilisant pour une tâche statique en un nouvel algorithme A' lui aussi auto-stabilisant, mais dont le temps de stabilisation visible est constant si k vaut 1. Son temps de stabilisation interne est de $O(T \times D)$, où T est le temps de stabilisation de A et ou D est le diamètre du réseau. L'algorithme de [37] s'exécute avec un ordonnancement asynchrone. Un autre procédé de transformation d'algorithme est celui présenté dans [38] : ce procédé ajoute des propriétés de stabilisation à un algorithme

non stabilisant pour un problème statique, dans le cas où le nombre de fautes est largement inférieur à la taille du réseau. Cependant, les résultats de complexité obtenus dépendent fortement de la répartition des fautes qui frappent le système : les meilleurs résultats sont obtenus quand les k fautes sont contigües (le temps de stabilisation est alors en $O(k^3)$), mais les performances décroissent (de manière exponentielle en k) quand les fautes sont localisées de manière arbitraire.

Le cas des tâches dynamiques est plus délicat à traiter dans le contexte de l'adaptativité en temps **[36]**. Considérons par exemple le cas de l'apparition d'une seule faute dans le problème du passage de jeton dans un réseau. Dans une exécution correcte (sans fautes), le jeton se propage dans le réseau, et du fait de la localité des échanges d'informations, la propagation du jeton ne peut se faire que de proche en proche (voir l'exécution A de la figure 4.3, où le nœud marqué J possède le jeton, et où les nœuds grisés sont les seuls capables d'agir). Or, quand une faute se produit à l'autre bout du réseau (voir l'exécution B de la figure 4.3, où le nœud marqué F est fautif), elle ne peut être corrigée que dans le voisinage (toujours en raison de la localité des échanges d'information). Même avec une hypothèse d'ordonnancement borné, il peut se faire que les actions corrigeant la faute n'interviennent qu'après un temps proportionnel à la taille du réseau (et non en fonction du nombre de fautes qui frappent le réseau). Ce cas de figure n'apparaît pas pour les tâches statiques : si une faute survient dans une configuration légitime, seul le voisinage est en mesure d'agir pour corriger. Ce résultat implique que pour obtenir des algorithmes pour des tâches dynamiques qui soient adaptatifs en temps, il est nécessaire de considérer que le système est soumis à un ordonnancement synchrone (ou supposer que dans un intervalle de temps fixé appelé *round*, tous les nœuds qui ont la capacité d'agir le font).

Plusieurs solutions au problème du passage de jeton dans un anneau ont été proposés dans ce contexte [12, 34] et **[35]**. Ils utilisent soit un ordonnancement synchrone ([34], **[35]**), soit mesurent le temps de stabilisation visible en rounds ([12]). L'algorithme de [12] stabilise en temps $O(k^2)$, celui de [34] en $O(k)$, et celui de **[35]** en $O(f)$, où f est le nombre effectif de fautes qui frappent le réseau (par opposition à k la borne sur le nombre maximum de fautes qui peuvent être tolérées).

4.2.3 Une classification

Il est possible de classer les algorithmes auto-stabilisants, k-stabilisants, et adaptatifs en temps suivant la difficulté des problèmes qu'il est possible de résoudre dans chaque cas. Par exemple, s'il est possible de résoudre un problème de manière auto-stabilisante, il est également possible de le résoudre de manière k-stabilisante (qui peut le plus peut le moins). Similairement, s'il est possible de résoudre un problème de manière adap-

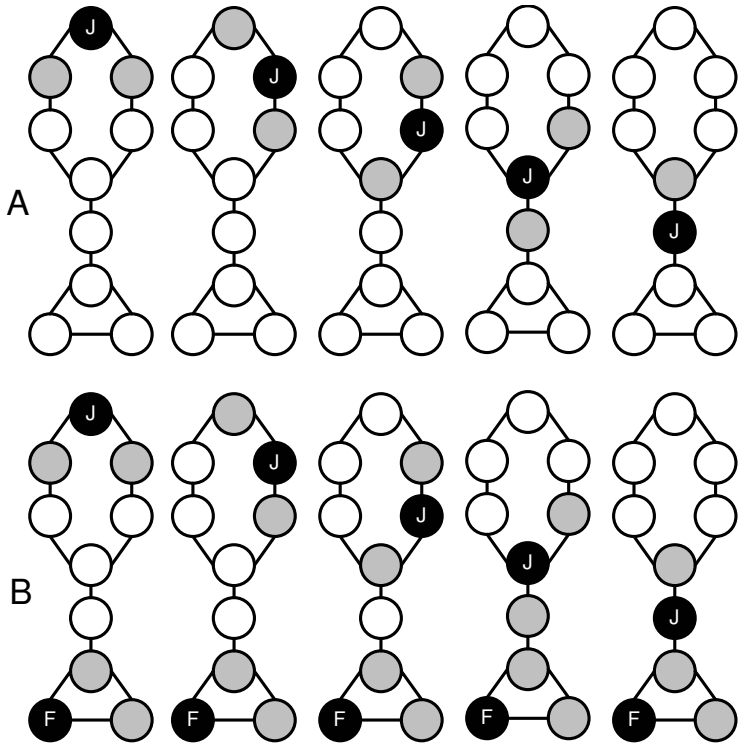

FIG. 4.3 – Adaptativité en temps pour les problèmes dynamiques

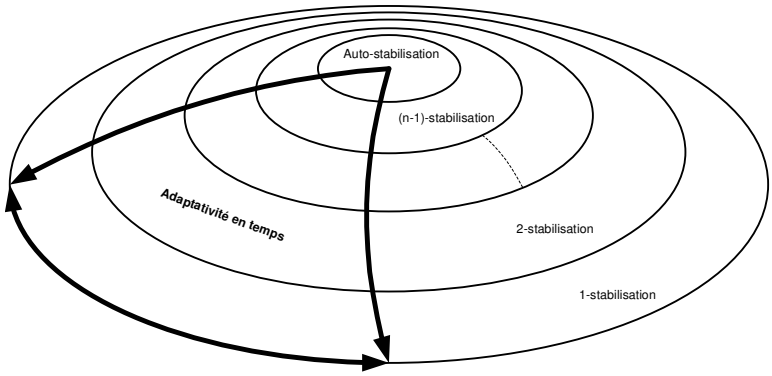

FIG. 4.4 – Classification des problèmes pour la k-stabilisation et l'adaptativité en temps

tative en temps, il est également possible de le résoudre sans essayer de contraindre le temps de stabilisation visible. Ainsi, la classe des problèmes solubles de manière adaptative en temps est un sous ensemble de la classes des problèmes solubles de manière auto-stabilisante, qui est elle-même un sous ensemble de la classes des problèmes admettant une solution k-stabilisante (voir figure 4.4). Ces inclusions sont strictes : des problèmes peuvent être résolus de manière k-stabilisante, mais pas auto-stabilisante (voir paragraphe 4.2.1), d'autres peuvent être résolus de manière auto-stabilisante, mais pas adaptative en temps (voir paragraphe 4.2.2).

4.3 Restreindre les classes de problèmes à résoudre

Dans le paragraphe 3.2, plusieurs problèmes spécifiques aux systèmes répartis sont globaux, c'est-à-dire qu'une modification sur un élément du système peut entraîner des répercussions dans tout le système. Dans cette section, il s'agit de parvenir à obtenir de bonnes performances (*i.e.* des performances qui ne dépendent pas de la taille du système) dans des systèmes à grande échelle. A l'inverse des sections 4.1 et 4.2, nous supposons que les corruptions de mémoire peuvent être parfaitement arbitraires, et que leur étendue est également arbitraire. Le principe est d'étudier des algorithmes dits *localisés*, c'est-à-dire que la correction dans une partie du système ne dépend pas de celle des autres parties du système.

4.3.1 Problèmes localisés

Les problèmes d'allocations de ressources (en général dérivés de problèmes de coloration de graphes) présentent généralement des contraintes

locales, ce qui fait qu'il sont le plus souvent localisables, c'est-à-dire solubles par des algorithmes localisés.

Allocation de créneaux TDMA

L'évitement et la gestion des collisions sont des aspects fondamentaux dans les protocoles pour les réseaux sans fils. Indirectement, un protocole de communication qui permet d'éviter les collisions permet d'économiser l'énergie, puisque le besoin de retransmettre un message s'en trouve réduit. L'accès au médium de communication par multiplexage temporel (TDMA, pour *Time Division Multiple Access*, où les utilisateurs émettent sur la même fréquence chacun à leur tour, les uns après les autres) est une technique raisonnable pour éviter les collisions.

Le problème algorithmique de l'allocation des créneaux de temps dans TDMA est lié au problème classique de l'allocation des fréquences dans FDMA (pour *Frequency Division Multiple Access*, où les utilisateurs utilisent des fréquences différentes pour communiquer). Pour FDMA, chaque couleur représente une fréquence, et pour éviter les collisions, on assure que tous les sommets à distance deux (ou moins) les uns des autres ont des couleurs différentes. Une contrainte supplémentaire est que les couleurs choisies par des sommets voisins sont suffisamment distantes pour éviter les interférences. Si l'ensemble des couleurs utilisées est l'intervalle des entiers $[0, \lambda]$, alors les couleurs (f_v, f_w) des sommets voisins (v, w) doivent satisfaire $|f_v - f_w| > 1$ pour éviter les interférences. La notation standard pour exprimer une telle contrainte est $L(\ell_1, \ell_2)$: pour toute paire de sommets à distance $i \in \{1, 2\}$, les couleurs diffèrent d'au moins ℓ_i. La coloration d'un graphe pour FDMA devrait donc satisfaire la contrainte $L(2, 1)$. De plus, une solution qui optimise le nombre de couleurs utilisées est préférable, puisqu'elle réduit le nombre de fréquences nécessaires. Le problème de la coloration dans TDMA est légèrement différent. Soit $L'(\ell_1, \ell_2)$ la contrainte que pour toute paire de sommets à distance $i \in \{1, 2\}$, les couleurs diffèrent de $\ell_i \bmod (\lambda + 1)$. Cette contrainte exprime les problèmes aux bordures des créneaux de temps. La contrainte de coloration usuelle pour TDMA est $L'(1, 1)$. Si les créneaux sont imprécis (par exemple parce que la synchronisation dans le temps n'est pas parfaite), il est possible de demander une séparation plus stricte des couleurs, comme $L'(2, 2)$. Minimiser le nombre de couleurs pour TDMA est souhaitable, car si une période de temps correspondant à la séquence de couleurs $0..\lambda$ est ramenée à l'intervalle unitaire $[0, 1]$, chaque couleur représente une fraction $1/(\lambda+1)$ de la bande passante. Donc plus λ est petit et mieux la bande passante est utilisée.

Le premier algorithme auto-stabilisant de type TDMA pour les réseaux de capteurs est présenté dans [56]. Ils partent d'une topologie en grille (extensible à toute topologie par plongement dans la grille) et supposent que chaque sommet connaît sa position dans la grille (cette position est utilisée

pour calculer l'allocation des créneaux). Utiliser leur approche dans des graphes généraux requiert que le plongement dans la grille soit le même pour tous les sommets et connu avant la mise en œuvre de l'algorithme. Par conséquent, cet algorithme ne peut être utilisé dans des réseaux évoluant dynamiquement. Dans **[45]**, un algorithme d'allocation de créneaux de temps est proposé, et gère les évolutions dynamiques du réseau, les défaillances transitoires, et le passage à l'échelle. L'approche pour gérer à la fois la dynamicité du réseau et les défaillances transitoires est celle de l'auto-stabilisation, qui assure que le système converge vers une allocation TDMA valide après une défaillance transitoire ou un changement de topologie. Le cas du passage à l'échelle est traité par le fait que l'algorithme probabiliste d'allocation de créneaux de temps présente un temps moyen de stabilisation en $O(1)$. La base de cet algorithme consiste en une technique de coloration rapide probabiliste, qui pourrait être exploitée afin de résoudre d'autres problèmes dans les réseaux de capteurs, ou dans certains réseaux ad hoc. Cette technique consiste à colorier rapidement le graphe pour effectuer un nommage unique de voisinage, et est détaillée ci-après.

Nommage unique de voisinage

Un algorithme qui effectue un nommage unique de voisinage donne à chaque nœud un nom distinct de celui de ses N^k voisins, où k est une constante donnée et où N^k désigne le voisinage à distance k. Ceci peut sembler bizarre étant donné que généralement, on suppose que les nœuds disposent déjà d'un identifiant unique (par exemple, l'adresse MAC de leur dispositif réseau sans fils), mais si on essaie d'utiliser ces identifiants pour de la coloration, l'ensemble potentiellement grand des identifiants peut poser problème lors du passage à l'échelle. Il est donc intéressant d'affecter des noms, tirés d'un espace de taille constante, en s'assurant qu'ils sont localement uniques. Ce problème peut être vu comme une coloration de N^k. L'idée de base de l'algorithme de coloration est le suivant : soit $\gamma = \lceil \Delta^t \rceil$ pour un $t > k$[1]. Si un nœud ne possède pas une couleur unique (choisie entre 0 et γ) dans son cache de N_p^k (supposé réémis régulièrement par chaque nœud à tout son voisinage en utilisant des techniques de type CSMA/CA – *emph Carrier Sense Multiple Access / Collision Avoidance*), il choisit une nouvelle couleur aléatoirement parmi les couleurs disponibles. Une propriété-clé de cet est la suivante : la propriété d'unicité de la couleur d'un nœud p est stable. De manière similaire, la pro-

[1]Il y a un compromis à trouver pour le choix de t dans $\gamma = \Delta^t$. D'abord, t devrait être suffisamment grand pour que le choix d'un nouvel identifiant soit unique avec grande probabilité. En général, de grandes valeurs de t diminuent le temps de convergence moyen de l'algorithme de nommage unique de voisinage, et de petites valeurs de t réduisent la constante d, qui à son tour réduit le temps de convergence moyen des algorithmes qui utilisent ce nommage unique.

priété d'unicité de tout sous-ensemble des nœuds est également stable. En d'autres termes, une fois qu'un nœud est considéré comme unique pour tous les voisinages auxquels il appartient, il est stable. Il est alors possible de raisonner à partir d'un modèle markovien sur les exécutions et montrer que la probabilité d'une séquence d'actions menant d'un ensemble stable donné à un ensemble stable plus grand est positive. En outre, le nommage unique de voisinage de **[45]** possède une propriété que les identifiants globaux du système n'ont pas : comme les identifiants sont de taille constante, la plus longue chaîne d'identifiants croissants dans le graphe est également de taille constante. Cette taille constante permet de construire d'autres algorithmes auto-stabilisants à partir de cette brique de base (par exemple, **[62]** utilise cette technique de nommage unique de voisinage à distance 2 pour construire une hiérarchisation du réseau, et **[45]** l'utilise à distance 3 pour une allocation de créneaux TDMA), tout en conservant un temps de stabilisation constant, et donc indépendant de la taille du réseau.

4.3.2 Tolérer les entités malicieuses

Comme indiqué dans le chapitre 2.1, le modèle de défaillance byzantin est le plus fort : un nœud du système peut tout simplement exhiber un comportement arbitraire. Bien sûr, pour causer des dommages au système, il est nécessaire que ce comportement arbitraire passe inaperçu auprès des nœuds corrects, c'est-à-dire que les valeurs communiquées et échangées doivent rester dans les intervalles de valeurs que les autres nœuds s'apprêtent à trouver. La plupart des solutions classiques utilisent une ou plusieurs hypothèses qui ne sont pas réalistes dans les systèmes à grande échelle comme ceux des réseaux de capteurs ou les réseaux ad hoc de grande taille :

1. ils supposent une connectivité totale (voir paragraphe 3.1) ;
2. ils supposent qu'une large majorité des nœuds est correcte (en général égale à plus des deux tiers des nœuds) ;
3. ils supposent que les nœuds ont accès à des primitives cryptographiques sûres (ce qu'un capteur à la capacité de traitement limitée n'est pas en mesure de fournir).

Puisque certains problèmes sont *a priori* localisables, est intéressant de s'intéresser à leur capacité à tolérer des fautes plus importantes que des simples corruptions de mémoires, comme les fautes byzantines. Plus précisément, on cherche à concevoir des algorithmes qui :

1. sont auto-stabilisants ;
2. peuvent s'exécuter sur des topologies (dynamiques) quelconques ;
3. n'utilisent pas de primitives cryptographiques ;

4. ne font pas d'hypothèses sur le nombre de nœuds byzantins ;
5. tolèrent les nœuds byzantins au sens ou ceux-ci n'ont que peu d'influence sur les nœuds corrects.

Une première approche pour obtenir des algorithmes pour obtenir de telles propriétés est présentée dans [63]. Le rayon de contamination byzantin est défini comme la distance jusqu'à laquelle le comportement des nœuds byzantins se fait sentir. On cherche évidemment à obtenir un rayon de contamination le moins grand possible. Un problème est r-restrictif si sa spécification interdit des combinaisons d'états dans une configuration sur des nœuds à distance au plus r. Par exemple, le problème de la coloration des nœuds d'un réseau est 1 restrictif, puisque deux nœuds voisins ne doivent pas avoir la même couleur. Par contre, le problème de la construction d'un arbre est r-restrictif (pour tout r compris entre 1 et $n-1$) car la correction implique que tous les parents choisis doivent former un arbre. Le théorème principal de [63] stipule que si un problème est r-restrictif, le meilleur rayon de contamination que l'on peut obtenir est r. Il est facile de voir que l'algorithme de nommage unique de voisinage mentionné section 4.3.1 est 1-restrictif pour un voisinage à distance 1, et admet d'ailleurs un rayon de contamination de 1 : si un nœud byzantin agit, il ne peut prendre – pour avoir un effet – que la même couleur que l'un de ses voisins ; ce voisin agit, mais s'il est correct, il prend une couleur qui n'est prise par aucun de ses voisins, ce qui fait que la réaction au comportement byzantin s'arrête dès ce nœud.

Dans [68], les auteurs considèrent le problème du coloriage des liens dans des réseaux dont la topologie est un arbre, de manière auto-stabilisante et tolérant les byzantins. Le coloriage des liens consiste à affecter des couleurs à chaque lien de telle sorte que deux liens adjacents au même nœud n'ont pas la même couleur. Cette coloration présente également des applications dans le domaine de l'allocation de fréquences dans les réseaux sans fils. Le fait que le réseau soit un arbre (orienté) permet de simplifier le problème, car le réseau n'est pas symétrique, et la décision de coloration peut être prise par un seul des deux nœuds adjacents (le nœud père dans [68]). Malgré le modèle simplifié, les auteurs montrent que :

1. il est nécessaire d'utiliser au moins $d+1$ couleurs, où d est le degré maximum du graphe de communication pour permettre un rayon de contamination constant (alors que d couleurs seraient suffisants sinon) ;
2. il est nécessaire d'avoir un ordonnancement spatial (voir section 3.1) centralisé si on souhaite tolérer à la fois des corruptions de mémoire et des byzantins ;
3. il existe un algorithme pour les arbres orientés qui utilise $d+1$ couleurs et possède un rayon de contamination de 2.

Quand le réseau est uniforme (tous les nœuds exécutent le même code) et anonyme (les nœuds n'ont pas de moyen de se distinguer l'un de l'autre), un algorithme auto-stabilisant de coloriage des liens ne peut plus faire l'hypothèse que la couleur d'un lien est déterminée par un unique nœud. En effet, puisque les nœuds sont uniformes, il ne peut y avoir priorité de l'un sur l'autre, et la coloration de l'arête qui les joint doit résulter d'un accord local entre eux. Dans **[61]**, un algorithme de coloration des arêtes auto-stabilisant et tolérant les byzantins est présenté. A la différence de [68], l'algorithme de **[61]** considère des réseaux uniformes et arbitraires (et non des arbres orientés), et utilise $2d-1$ couleurs (au lieu de $d+1$). En ce qui concerne le rayon de contamination byzantin, le protocole de **[61]** est optimal, puisque l'influence d'un nœud byzantin est limitée à eux-mêmes ; c'est-à-dire que le sous-système composé uniquement des processus correct est toujours correct.

Le principe de l'algorithme de **[61]** est le suivant : chaque nœud maintient une liste de couleurs affectées à ses liens incidents, et échange périodiquement cette liste avec ses voisins. A partir de la liste reçue de son voisin v, un nœud u peut proposer une couleur pour le lien (u,v). Cette couleur proposée ne doit pas apparaître dans l'ensemble des couleurs incidents de u ni de v. L'ordonnancement spatial étant central (du fait de la nécessité montrée dans [68]), il est impossible que deux voisins proposent une couleur en même temps. Puisque l'ensemble des couleurs disponibles est $2d-1$, u est toujours en mesure de proposer une couleur qui n'est déjà utilisée ni par u, ni par v. Si à la fois u et v sont corrects, la couleur c du lien (u,v) n'est plus jamais changée. En cas de nœud byzantin, il peut cependant arriver qu'un tel byzantin propose sans arrêt des couleurs en conflit avec celles des autres voisins. Si cette couleur est en conflit avec une couleur sur laquelle u et v se sont déjà mis d'accord, cette proposition est ignorée. Le cas restant survient quand u a deux voisins v et w (où u et v sont corrects et w byzantin) et u ne s'est encore mis d'accord ni avec v, ni avec w. Le nœud byzantin w pourrait continuellement proposer des couleurs en conflit avec v, et u pourrait toujours choisir la couleur proposée par w. Pour assurer que ce comportement ne peut pas se produire infiniment souvent, **[61]** utilise une liste des priorités de telle sorte que les voisins de u obtiennent alternativement la priorité dans la proposition de la couleur du lien. Ensuite, une fois que u et v se sont mis d'accord sur la couleur du lien (u,v), cette couleur ne peut plus être modifiée par w, car ses propositions sont systématiquement rejetées.

Chapitre 5

Conclusion et perspectives

Les techniques classiques d'algorithmique répartie tolérante aux pannes sont pour la plupart inadaptées au passage à l'échelle. Les utiliser conduirait à des mécanismes qui soit coûtent trop de ressources (mémoire, temps de calcul), soit sont disproportionnés par rapport au problème à résoudre.

Pour contourner les résultats d'impossibilité dans le cadre de l'auto-stabilisation, plusieurs pistes ont été suivies : restreindre les hypothèses sur les fautes susceptibles de se produire (que ce soit leur nature, ou leur localisation géographique), ou restreindre le type d'applications que l'on se propose de résoudre. Pour le cas particulier des réseaux sans fils, plusieurs problèmes d'allocation de ressources (fréquences, créneaux de temps) peuvent être résolus de manière fortement tolérante aux pannes : corruption arbitraire des mémoires, comportement malicieux étendus.

La frontière entre les problèmes impossibles à résoudre car trop coûteux et ceux que l'on sait traiter avec des contraintes raisonnables reste malgré tout bien floue. Plusieurs résultats récents montrent qu'il existe probablement un compromis entre les ressources utilisées et la capacité à tolérer des défaillances, mais un gros travail supplémentaire est encore nécessaire pour obtenir une vision précise de ce compromis.

5.1 Perspectives théoriques

Depuis sa définition en 1974 par Dijkstra [22], l'auto-stabilisation dispose de fondements mathématiques solides. Il n'est d'ailleurs pas surprenant que divers formalismes issus des Mathématiques, de l'Automatique, ou de l'Informatique aient été tour à tour utilisés pour prouver l'auto-stabilisation : les fonctions de transfert [69], les systèmes d'itérations [7], l'algèbre max-plus **[31]**, la réécriture de mots [9], la logique temporelle [58] ou d'ordre supérieur [64], etc. Les travaux mentionnés dans les chapitres précedents ouvrent la possibilité de développer d'autres aspects théoriques liès à la notion d'auto-stabilisation.

5.1.1 Auto-stabilisation en compétition

Dans les approches «purement» auto-stabilisantes, les différents nœuds collaborent pour accomplir une tâche commune (satisfaire la spécification du problème), en dépit d'un environnement qui est vu comme un adversaire tentant de faire échouer la stabilisation. Dans l'auto-stabilisation en présense de nœuds byzantins (mentionnée section 4.3.2), on distingue deux sous-ensembles de nœuds, ceux qui exécutent correctement l'algorithme (les nœuds *corrects*), et ceux qui tentent de faire échouer la stabilisation (les nœuds *byzantins*). Là encore, tous les nœuds corrects collaborent à la même tâche, et tous les nœuds byzantins ont des ressources illimitées pour mener leurs actions.

Il est probable que le modèle de l'auto-stabilisation en présence de byzantins est trop extrême pour correspondre à la réalité. Par exemple, dans le routage interdomaine dans Internet, les nœuds ont un objectif commun (permettre l'acheminement des communication à travers Internet) mais également un objectif local (par exemple, maximiser son profit personnel). On peut alors raffiner le modèle d'auto-stabilisation en présence de byzantins en un modèle non plus binaire (corrects contre byzantins), mais unifié dans l'objectif global et en compétition dans l'objectif local. Un exemple de tel probleme est le suivant : on considère un réseau ou les nœuds sont partitionnés en deux groupes, les *pro-débit* (qui cherchent à maximiner le débit) et les *pro-latence* (qui cherchent à minimiser la latence) ; ensuite on souhaite construire dans un réseau où chaque lien a une débit et une latence donnée un arbre couvrant. Si chaque nœud exécute un algorithme auto-stabilisant comme ceux décrits section 3.2.1, on peut facilement construire des exemples où un arbre couvrant ne sera jamais construit. Par suite, trouver un algorithme auto-stabilisant qui permette de résoudre un objectif global puis de maximiser un objectif local en tenant compte du fait que cette maximisation peut mettre en péril l'objectif global est un problème ouvert.

5.1.2 Complexité et auto-stabilisation

Il existe une grande quantité de résultats d'impossibilité ou de bornes inférieures en algorithmique distribuée. Pour les résultats de bornes inférieures, proportionnellement, peu de résultats concernent spécifiquement l'auto-stabilisation. L'explication est double :

1. *borne inférieure en mémoire* : si un algorithme distribué «classique» (*i.e.* non-stabilisant) a necéssairement besoin d'une certaine quantité de mémoire, alors un algorithme auto-stabilisant également (il doit fonctionner correctement à partir de la configuration initiale de l'algorithme classique) ;
2. *borne inférieure en temps* : si un algorithme distribué a nécessairement besoin d'un certain temps avant de résoudre un problème, alors un

algorithme auto-stabilisant aussi (le temps de stabilisation est égal au *maximum*, sur toutes les éxécutions possibles du système, dont celles qui partent d'un état bien connu).

En d'autres termes, les résultats de bornes inférieures de l'algorithmique classique se transposent directement vers l'auto-stabilisation, mais l'inverse n'est pas necéssairement vrai. En particulier, de nombreux problèmes impossible à résoudre de manière auto-stabilisante (notamment pour des raisons de symétrie dans la configuration initiale du système) peuvent l'être facilement par un algorithme initialisé (en particulier en restreignant les configurations initiales de manière à ce qu'une symétrie ne puisse apparaître).

Récemment, plusieurs algorithmes distribués d'approximation de problèmes NP-complets ont été développés. Des versions auto-stabilisantes de certains d'entre eux commencent tout juste à apparaître. Il existe en général un compromis entre la *localité* de l'algorithme (la quantité d'information qu'il doit connaitre de son voisinage) et son *efficacité* (la qualité de l'approximation proposée). Dans un contexte d'auto-stabilisation, la question de savoir si ce compromis existe reste ouverte.

5.1.3 Auto-stabilisation systématique

Au cours des chapitres précédents (et en particulier la section 3.2.1), nous avons montré qu'une condition sur le code exécuté par un algorithme réparti pouvait impliquer l'auto-stabilisation du système tout entier, sous des hypothèses systèmes très diverses. En particulier, suivant le modèle considéré (atomicité faible ou forte), des résultats différents sont obtenus (ordre partiel dans le cas de l'atomicité forte, ordre total dans la cas de l'atomicite forte). Or, un ordre partiel permet de résoudre le problème de la liste ordonnée des ancêtres, ce qui dans un graphe fortement connexe (qui correspond à la plupart des cas pratiques) permet en utilisant la technique de [25] de résoudre tout problème statique. Le fait que le même opérateur (utilisant l'ordre partiel) ne fonctionne pas correctement dans un modèle à atomicité faible n'implique pourtant pas qu'il n'existe aucun opérateur permettant de résoudre le problème. Un algorithme *ad hoc* a même été proposé dans **[21]** pour résoudre le problème de la liste ordonnée des identifiants dans un modèle à atomicité faible. La question ouverte est celle de l'existence d'un opérateur universel (pour les tâches statiques) dans un modèle à atomicité faible.

Par la suite, même si un tel opérateur existe et montre l'universalité de l'approche, il n'est pas nécessairement le plus adapté pour résoudre un problème particulier. En effet, l'espace mémoire (en $O(n \log_2(n))$) et la quantité d'informations transférée sur le réseau reste importante par rapport à d'autres opérateurs spécialisés. Actuellement la technique qui consiste à

trouver un opérateur adéquat pour résoudre un problème, même si elle permet de simplifier considérablement la preuve d'auto-stabilisation, reste ad hoc. La question de savoir s'il est possible, systématiquement, à partir d'un spécification d'un problème statique donné, de concevoir l'opérateur satisfaisant aux propriétés enoncées, semble mériter qu'on s'y intéresse.

5.2 Perspectives pratiques

Lorsque qu'une théorie est mature, les applications arrivent sans tarder. Pour l'auto-stabilisation, de nombreux protocoles actuellement utilisés dans le routage sur Internet utilisent des principes d'auto-stabilisation, à des degrés divers. Par exemple, le protocole d'échange d'état des liens dans OSPF (*Open Shortest Path First*, un protocole de routage intra-domaine dans Internet) a été prouvé auto-stabilisant par Nancy Lynch. Hors du domaine du routage cependant, les applications restent pour le moment limitées. Cette limitation peut s'expliquer par différents facteurs, dont les deux principaux sont les suivants :

1. *les hypothèses de l'auto-stabilisation ne s'appliquent pas à tous les systèmes réels* : par exemple, l'auto-stabilisation suppose que les processeurs ne cessent jamais leur exécution, or il est bien connu qu'une suite de trois instructions mal écrites (et pouvant résulter d'une corruption de mémoire) suffit à bloquer définitivement le processeur Pentium d'Intel ;
2. *on ne dispose pas pour les logiciels courants d'un support d'application auto-stabilisant* : les systèmes d'exploitation (bureautique et réseaux) sur lesquels s'appuient les logiciels actuellement développés n'ont pas été conçus ni prouvés auto-stabilisants, et construire des briques logicielles auto-stabilisants au dessus de telles fondations peut sembler artificiel.

Systèmes auto-stabilisants

Pour rendre possible de développement véritable de systèmes répartis auto-stabilisants, deux approches complémentaires sont possibles :

la conception ascendante : on part des fondements du systèmes (matériel, système d'exploitation), et on aboutit à des programmes qui se basent sur des fondations elles-mêmes auto-stabilisantes. Par exemple, le travail de Shlomi Dolev et de son équipe s'inscrit spécifiquement dans ce cadre. Dans [24], ils proposent des mécanismes pour rendre auto-stabilisant un processeur, c'est à dire pour garantir qu'au bout d'un temps fini, le processeur exécute indéfiniment les instructions élémentaires *fetch*, *decode*, *execute*. Plusieurs approches sont décrites : il

est possible de concevoir à partir de rien un nouveau processeur, ou bien d'ajouter un mécanisme matériel externe (appelé *watchdog*) pour vérifier que le processeur ne se trouve pas dans un état incorrect. Par la suite, dans [27], ils ont posé les bases d'un système d'exploitation minimal mais auto-stabilisant, et ont considéré des services supplémentaires (gestion de la mémoire dans, compilation de code dans). Si les fondations de ce travail sont cohérentes et permettent de se rendre compte de l'avancement général du projet, il est également clair que la mise à disposition d'un système complet et utilisable pour des applications évoluées prendra encore plusieurs années.

la conception descendante : on part des applications que l'on souhaite développer et qui correspondent à des besoins actuels et clairement exprimés, et on montre que, sous l'hypothèses que les couches inférieures sont auto-stabilisantes, ces nouveaux services sont eux aussi auto-stabilisants. La vérification de la propriété d'auto-stabilisation dans ce contexte pose de nombreux problèmes pratiques, car l'auto-stabilisation est compromise par des exécutions particulières du systèmes, dont la probabilité d'occurrence est infinitésimale. De plus, reproduire une exécution particulière qui a mis en évidence un problème de l'implantation d'un algorithme auto-stabilisant sur un système réel (potentiellement composé d'un grand nombre de machines réelles) diminue encore cette probabilité.

Nous avons élaboré une infrastructure, FAIL-FCI **[46]**, qui devrait permettre à terme la conception descendante de systèmes auto-stabilisants. Cette infrastructure est actuellement développée dans le cadre de divers projets (le projet GrideXplorer de l'ACI «Masse de Données», et le projet FRAGILE de l'ACI «Sécurité et Informatique»), et prend la forme de deux composants principaux :

1. *un langage de spécification de scénarios de fautes* (FAIL, pour *FAult Injection Language*) : ce langage permet de spécifier, en utilisant un formalisme proche des automates synchronisés, des scénarios destinés à des mesures quantitatives (toutes les x secondes, une proportion y des composants du système a une probabilité z de subir une panne) ou qualitatives (une fois que le processus p_1 de ma machine m_1 a exécuté la ligne de code x, alors le processus p_2 de la machine m_2 doit subir une panne avant d'exécuter la ligne de code y). Ceci permet en particulier de spécifier des scénarios de fautes évolués, comme des fautes en cascade (ou épidémiques) où il existe un rapport de causalité entre la première faute et les suivantes.
2. *un intergiciel d'injection de pannes distribué* (FCI, pour *FAIL Cluster Implementation*) : ce logiciel s'exécute entre le système d'exploitation et l'application sous test. Un point fort de cette approche est qu'elle est transparente pour le concepteur et le programmeur de l'application,

car le code source de l'application n'est pas modifié et l'application n'a pas à être recompilée.

En l'état actuel, FAIL-FCI permet d'élaborer des scénarios de fautes (ou d'attaques en considérant des fautes malicieuses) élaborés, et le prototype développé permet d'injecter des fautes dans deux types d'applications réparties :

1. les applications natives (c'est à dire élaborées à partir de code compilé en programme exécutable, ce qui est généralement le cas en C ou en FORTRAN) ;
2. certaines applications fonctionnant sous machine virtuelle (pour l'instant, seul le code de la machine virtuelle Java est supporté **[48]**).

Les fautes qu'il est possible d'injecter *via* notre outil sont limitées dans sa version actuelle : les pannes crash (suivies d'un redémarrage éventuel à partir de l'état initial) et les suspensions (généralement suivies d'une reprise) qui servent à simuler l'asynchronisme du système.

FAIL-FCI a déjà permi de révéler des anomalies de fonctionnement dans plusieurs applications (comme une application de calcul global dans **[47]**), et a été remarqué dans le cadre du réseau d'excellence européen CoreGRID : pour le deuxième programme joint d'activité (démarré en septembre 2005), une tâche spécifique «Injection de fautes et stress d'applications» a été introduite. Nous prévoyons de poursuivre le développement de notre outil. En particulier, nous prévoyons la possibilité d'introduire de nouveaux types de fautes (corruption de mémoire, préemption de ressources), de tester de nouveaux types d'applications (par exemple les applications Pair-à-Pair, bien connues pour leur capacité à passer à l'échelle, n'ont été que faiblement étudiées concernant les aspects de tolérance aux pannes), et de permettre l'injection de pannes pour d'autres modèles d'exécution (par exemple les applications basées sur MPI). Un autre aspect utile serait de développer la possibilité de rejouer des exécution réelles (modélisées par des traces d'utilisateurs) pour juger de la performance d'une application dans des conditions réelles.

Le but à terme de FAIL-FCI serait de disposer d'un outil pour effectuer des bancs d'essais standardisés pour la tolérances aux fautes dans les systèmes répartis.

Réseaux de capteurs sans fils

Les réseaux de capteurs sans fils représentent l'une des perspectives les plus évidentes pour l'application effective de l'auto-stabilisation dans les systèmes réels. Les raisons de l'engouement constaté de la communauté sont multiples, mais les deux principales sont les suivantes :

la nature des problèmes à résoudre : de nombreux problèmes actuellement étudiés dans le domaine des réseaux de capteurs peuvent être

modélisés par des graphes, pour lesquels de nombreuses solutions réparties, voire auto-stabilisantes, sont connues. En outre, l'aspect distribué de la solution est ici essentiel car au vu de la taille des réseaux de capteurs actuellement prévu dans les prochaines années (plusieurs dizaines de milliers), il est impensable d'imaginer initialiser un tel réseau composant par composant suite aux résultats d'un algorithme séquentiel.

Dans le modèle réseau en couches communément accepté pour les réseaux sans fils, l'algorithmique répartie intervient dans les quatre couches les plus hautes : liaison de données, réseau, transport et application. Dans les solutions proposées jusqu'à présent toutefois, l'essentiel se situe dans les couches liaison de données et réseau.

Pour la couche liaison de données, et plus particulièrement la sous-couche d'accès au médium de communication (*MAC*), plusieurs types de protocoles peuvent être utilisés, les plus répandus dans le cadre des réseaux sans fils étant *CSMA*, *TDMA* **[45]** ou *FDMA*. Dans tous les cas, l'objectif principal est de permettre l'accès au médium en dépit de problèmes qui compromettent la performance du réseau (latence, débit) ou l'énergie utilisée pour communiquer (cruciale dans les réseaux de capteurs). Le problème principal est celui des collisions, qui survient lorsque des nœuds voisins utilisent le médium radio de manière concomitante pour émettre des informations ; les nœuds récepteurs peuvent alors recevoir un signal brouillé ou inutilisable. Dans les réseaux de capteurs, d'autres problèmes viennent se greffer, comme celui qui fait que recevoir un signal est presque aussi coûteux (en terme d'énergie consommée) que rester en attente de réception d'un signal ; cette limitation pratique induit sur le plan algorithmique des techniques qui proposent un compromis entre la latence et la consommation électrique pour communiquer dans le réseau.

De par leur nature, les techniques lies au TDMA et au FDMA sont liées au coloriage des nœuds ou des liens d'un graphe. Comme les réseaux sans fils que nous considérons doivent être auto-organisés, ce coloriage ne peut être prédéfini avant de déployer le système, et doit résulter d'un algorithme exécuté par le système lui-même : un algorithme distribué. Les solutions distribuées actuelles aux problèmes de coloriage de graphes montrent que des bornes théoriques existent quant à la qualité du coloriage effectué suivant la localité de l'algorithme distribué (celle-ci étant directement liée à l'énergie consommée). En outre, TDMA requiert que les horloges des nœuds du système soient synchronisées, ce qui peut nécessiter l'utilisation d'algorithmes répartis de synchronisation d'horloge. D'autres algorithmes distribués basées sur des solutions à des problèmes de graphes peu-

vent être utiles pour la couche réseau : par exemple, il est possible de construire une infrastructure efficace du point de vue de l'énergie en auto-organisant le réseau de manière hiérarchique ou en déterminant un sous réseau présentant des propriété particulières. Les problèmes de graphe considérés sont alors le plus souvent liés à la notion d'ensemble dominant (ensemble de nœuds capable de communiquer avec tous les autres nœuds du graphe), le but consistant à rendre cet ensemble aussi petit et/ou efficace que possible.

les spécificités techniques des réseaux de capteurs : les réseaux de capteurs sans fils sont des machines basées sur des composants simples et peu coûteux. Ces machines supportent peu de périphériques, peu de services système, et leur système d'exploitation reste de très petite taille : par exemple, TinyOS, le système d'exploitation utilisé dans la grande majorité des plates-formes actuellement déployées dans le milieu académique, utilise 3450 octets pour son code et 226 octets pour ses données. Cette taille réduite rend possible l'étude de l'auto-stabilisation à l'échelle du système d'exploitation tout entier. En outre, le fait que ces réseaux de capteurs vont être fabriqués et déployés à très grande échelle induit nécessairement que la tolérance aux pannes doit être considérée dès le départ comme un composant essentiel. Pour la plupart des applications considérées (collecte de données sur une longue période), des solutions non-masquantes, comme l'auto-stabilisation, sont probablement préférable, du fait de l'utilisation moindre de ressources par rapport aux approches masquantes à base de réplication et de consensus.

L'essort actuel de la recherche dans le domaine des résaux de capteurs laisse supposer qu'à moyen terme, ces réseaux seront effectivement déployés à grand échelles (plusieurs dizaines de milliers de nœuds). Dans ce cadre, il n'est plus question d'administrer de manière individuelle chaque composant du réseau, et de gérer les pannes au moyen d'un intervention humaine. Les techniques pour l'auto-stabilisation à grande échelle, développées pour l'instant de manière théorique, gagneraient à être déployées de manière massive sur de vrais périphériques. En effet, elles permettraient de gérer de manière unifiée et simple l'auto-organisation rendue nécessaire par le passage à l'échelle, et la tolérance aux pannes qui se produiront inévitablement et constament. De plus, l'adaptation de la plate-forme d'injection de fautes FAIL-FCI aux réseaux de capteurs est à l'étude et serait, à notre connaissance, unique.

Bibliographie

[1] Yehuda Afek and Anat Bremler-Barr. Self-stabilizing unidirectional network algorithms by power supply. *Chicago J. Theor. Comput. Sci.*, 1998, 1998.

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

[3] Yehuda Afek and Shlomi Dolev. Local stabilizer. *J. Parallel Distrib. Comput.*, 62(5) :745–765, 2002.

[4] Luc Onana Alima, Joffroy Beauquier, Ajoy Kumar Datta, and Sébastien Tixeuil. Self-stabilization with global rooted synchronizers. In *Proceedings of the 18th International Conference on Distributed Computing Systems, 26 - 29 May, Amsterdam, The Netherlands*, pages 102–109. IEEE Press, 1998.

[5] James H. Anderson, Yong-Jik Kim, and Ted Herman. Shared-memory mutual exclusion : major research trends since 1986. *Distributed Computing*, 16(2-3) :75–110, 2003.

[6] Dana Angluin. Local and global properties in networks of processors (extended abstract). In *STOC '80 : Proceedings of the twelfth annual ACM symposium on Theory of computing*, pages 82–93, New York, NY, USA, 1980. ACM Press.

[7] Anish Arora, Paul C. Attie, Michael Evangelist, and Mohamed G. Gouda. Convergence of iteration systems. *Distributed Computing*, 7(1) :43–53, 1993.

[8] Baruch Awerbuch, Boaz Patt-Shamir, George Varghese, and Shlomi Dolev. Self-stabilization by local checking and global reset (extended abstract). In Gerard Tel and Paul M. B. Vitányi, editors, *Distributed Algorithms, 8th International Workshop, WDAG '94*, volume 857 of *Lecture Notes in Computer Science*, pages 326–339. Springer, 1994.

[9] Joffroy Beauquier, Béatrice Bérard, Laurent Fribourg, and Frédéric Magniette. Proving convergence of self-stabilizing systems using first-order rewriting and regular languages. *Distributed Computing*, 14(2) :83–95, 2001.

[10] Joffroy Beauquier, Stéphane Cordier, and Sylvie Delaët. Optimum probabilistic self-stabilization on uniform rings. In *Proceedings on the Workshop on Self-stabilizing Systems*, pages 15.1–15.15, 1995.

[11] Joffroy Beauquier, Sylvie Delaët, Shlomi Dolev, and Sébastien Tixeuil. Transient fault detectors. In Shay Kutten, editor, *Distributed Computing, 12th International Symposium, DISC '98, Andros, Greece, September 24-26, 1998, Proceedings*, pages 62–74. Springer, 1998.

[12] Joffroy Beauquier, Christophe Genolini, and Shay Kutten. Optimal reactive -stabilization : The case of mutual exclusion. In *Proceedings of the Eighteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 209–218, 1999.

[13] Joffroy Beauquier, Maria Gradinariu, and Colette Johnen. Memory space requirements for self-stabilizing leader election protocols. In *Proceedings of the ACM Conference on Principles of Distributed Computing (PODC 99)*, pages 199–208, 1999.

[14] James E. Burns and Jan K. Pachl. Uniform self-stabilizing rings. *ACM Trans. Program. Lang. Syst.*, 11(2) :330–344, 1989.

[15] David Cavin, Yoav Sasson, and André Schiper. Consensus with unknown participants or fundamental self-organization. In Ioanis Nikolaidis, Michel Barbeau, and Evangelos Kranakis, editors, *Ad-Hoc, Mobile, and Wireless Networks : Third International Conference, ADHOC-NOW 2004, Vancouver, Canada, July 22-24, 2004. Proceedings*, volume 3158 of *Lecture Notes in Computer Science*, pages 135–148. Springer, 2004.

[16] Jorge Arturo Cobb and Mohamed G. Gouda. Stabilization of routing in directed networks. In Ajoy Kumar Datta and Ted Herman, editors, *Self-Stabilizing Systems, 5th International Workshop, WSS 2001, Lisbon, Portugal, October 1-2, 2001, Proceedings*, volume 2194 of *Lecture Notes in Computer Science*, pages 51–66. Springer, 2001.

[17] Jorge Arturo Cobb and Mohamed G. Gouda. Stabilization of general loop-free routing. *J. Parallel Distrib. Comput.*, 62(5) :922–944, 2002.

[18] Ajoy K Datta, Maria Gradinariu, and Sébastien Tixeuil. Self-stabilizing mutual exclusion using unfair distributed scheduler. In *IEEE International Parallel and Distributed Processing SYmposium (IPDPS'2000)*, pages 465–470, Cancun, Mexico, May 2000. IEEE Press.

[19] Sylvie Delaët. *Auto-stabilisation : Modèle et Applications à l'Exclusion Mutuelle*. PhD thesis, Université Paris Sud, December 1995.

[20] Sylvie Delaët, Bertrand Ducourthial, and Sébastien Tixeuil. Self-stabilization with r-operators revisited. In *Proceedings of the Seventh Symposium on Self-stabilizing Systems (SSS'05)*, volume 3764 of *Lecture Notes in Computer Science*, pages 68–80, Barcelona, Spain, October 2005. Springer Verlag.

[21] Sylvie Delaët and Sébastien Tixeuil. Tolerating transient and intermittent failures. *Journal of Parallel and Distributed Computing*, 62(5) :961–981, May 2002.

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

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

[24] Shlomi Dolev and Yinnon A. Haviv. Self-stabilizing microprocessor - analyzing and overcoming soft-errors (extended abstract). In Christian Müller-Schloer, Theo Ungerer, and Bernhard Bauer, editors, *Organic and Pervasive Computing - ARCS 2004, International Conference on Architecture of Computing Systems, Augsburg, Germany, March 23-26, 2004, Proceedings*, volume 2981 of *Lecture Notes in Computer Science*, pages 31–46. Springer, 2004.

[25] Shlomi Dolev and Ted Herman. Superstabilizing protocols for dynamic distributed systems. *Chicago J. Theor. Comput. Sci.*, 1997, 1997.

[26] Shlomi Dolev and Elad Schiller. Self-stabilizing group communication in directed networks. *Acta Inf.*, 40(9) :609–636, 2004.

[27] Shlomi Dolev and Reuven Yagel. Toward self-stabilizing operating systems. In *15th International Workshop on Database and Expert Systems Applications (DEXA 2004), with CD-ROM, 30 August - 3 September 2004, Zaragoza, Spain*, pages 684–688. IEEE Computer Society, 2004.

[28] Philippe Duchon, Nicolas Hanusse, and Sébastien Tixeuil. Optimal randomized self-stabilizing mutual exclusion in synchronous rings. In *Proceedings of the 18th Symposium on Distributed Computing (DISC 2004)*, number 3274 in Lecture Notes in Computer Science, pages 216–229, Amsterdam, The Nederlands, October 2004. Springer Verlag.

[29] Bertrand Ducourthial. New operators for computing with associative nets. In Luisa Gargano and David Peleg, editors, *SIROCCO'98, 5th International Colloquium on Structural Information & Communication Complexity*, pages 51–65. Carleton Scientific, 1998.

[30] Bertrand Ducourthial and Sébastien Tixeuil. Self-stabilization with r-operators. *Distributed Computing*, 14(3) :147–162, July 2001.

[31] Bertrand Ducourthial and Sébastien Tixeuil. Self-stabilization with path algebra. *Theoretical Computer Science*, 293(1) :219–236, 2003. Extended abstract in Sirrocco 2000.

[32] Michael J. Fischer, Nancy A. Lynch, and Mike Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2) :374–382, 1985.

[33] Felix Freiling, Rachid Guerraoui, and Petr Kouznetsov. The failure detector abstraction. Technical Report TR-2006-003, 2006.

[34] Christophe Genolini. Optimal k-stabilization : the case of synchronous mutual exclusion. In *Proceedings of Parallel and Distributed Computing Systems (PDCS'2000)*, pages 371–376, November 2000.

[35] Christophe Genolini and Sébastien Tixeuil. Reactive k-stabilization and time adaptivity : possibility and impossibility results. Technical Report 1276, Laboratoire de Recherche en Informatique, University of Paris Sud XI, 2001.

[36] Christophe Genolini and Sébastien Tixeuil. A lower bound on k-stabilization in asynchronous systems. In *Proceedings of IEEE 21st Symposium on Reliable Distributed Systems (SRDS'2002)*, Osaka, Japan, October 2002.

[37] Sukumar Ghosh, Arobinda Gupta, Ted Herman, and Sriram V. Pemmaraju. Fault-containing self-stabilizing algorithms. In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 45–54, 1996.

[38] Sukumar Ghosh and Xin He. Scalable self-stabilization. *J. Parallel Distrib. Comput.*, 62(5) :945–960, 2002.

[39] Mohamed G. Gouda and F. Furman Haddix. The linear alternator. In Sukumar Ghosh and Ted Herman, editors, *3rd Workshop on Self-stabilizing Systems*, pages 31–47. Carleton University Press, 1997.

[40] Maria Gradinariu and Sébastien Tixeuil. Self-stabilizing vertex coloring of arbitrary graphs. In *International Conference on Principles of Distributed Systems (OPODIS'2000)*, pages 55–70, Paris, France, December 2000.

[41] Maria Gradinariu and Sébastien Tixeuil. Tight space uniform self-stabilizing l-mutual exclusion. In *IEEE International Conference on Distributed Computing Systems (ICDCS'01)*, pages 83–90, Phoenix, Arizona, May 2001. IEEE Press.

[42] Ted Herman. Probabilistic self-stabilization. *Information Processing Letters*, 35(2) :63–67, 1990.

[43] Ted Herman. Models of self-stabilization and sensor networks. In Samir R. Das and Sajal K. Das, editors, *Distributed Computing - IWDC 2003, 5th International Workshop*, volume 2918 of *Lecture Notes in Computer Science*, pages 205–214. Springer, 2003.

[44] Ted Herman and Sriram V. Pemmaraju. Error-detecting codes and fault-containing self-stabilization. *Inf. Process. Lett.*, 73(1-2) :41–46, 2000.

[45] Ted Herman and Sébastien Tixeuil. A distributed tdma slot assignment algorithm for wireless sensor networks. In *Proceedings of the First Workshop on Algorithmic Aspects of Wireless Sensor Networks (AlgoSensors'2004)*, number 3121 in Lecture Notes in Computer Science, pages 45–58, Turku, Finland, July 2004. Springer-Verlag.

[46] William Hoarau and Sébastien Tixeuil. A language-driven tool for fault injection in distributed applications. In *Proceedings of the IEEE/ACM Workshop GRID 2005*, page to appear, Seattle, USA, November 2005.

[47] William Hoarau, Sébastien Tixeuil, and Fabien Vauchelles. Fault injection in distributed java applications. Technical Report 1420, Laboratoire de Recherche en Informatique, Université Paris Sud, October 2005.

[48] William Hoarau, Sébastien Tixeuil, and Fabien Vauchelles. Fault injection in distributed java applications. In *International Workshop on Java for Parallel and Distributed Computing (joint with IPDPS 2006)*, page to appear, Greece, April 2006. IEEE.

[49] Chin-Tser Huang and Mohamed G. Gouda. State checksum and its role in system stabilization. In *25th International Conference on Distributed Computing Systems Workshops (ICDCS 2005 Workshops)*, pages 29–34. IEEE Computer Society, 2005.

[50] Amos Israeli and Marc Jalfon. Token management schemes and random walks yield self-stabilizing mutual exclusion. In *Proceedings of the Ninth Annual ACM Symposium on Principles of Distributed Computing*, pages 119–131, 1990.

[51] Colette Johnen. Service time optimal self-stabilizing token circulation protocol on anonymous unidirectional rings. In *Proceedings of the IEEE Symposium on Reliable Distributed Systems*, pages 80–89. IEEE Press, 2002.

[52] Colette Johnen, Luc Onana Alima, Ajoy Kumar Datta, and Sébastien Tixeuil. Self-stabilizing neighborhood synchronizer in tree networks. In *Proceedings of the 19th International Conference on Distributed Computing Systems*, pages 487–494, Austin, TX, USA, May-June 1999. IEEE Computer Society.

[53] Colette Johnen, Franck Petit, and Sébastien Tixeuil. Auto-stabilisation et protocoles réseaux. *Technique et Science Informatiques*, 23(8) :1027–1056, 2004.

[54] Colette Johnen and Sébastien Tixeuil. Route preserving stabilization. In *Proceedings of the Sixth Symposium on Self-stabilizing Systems (SSS'03)*, Lecture Notes in Computer Science, San Francisco, USA, june 2003. Springer Verlag. Also in the Proceedings of DSN'03 as a one page abstract.

[55] Shmuel Katz and Kenneth J. Perry. Self-stabilizing extensions for message-passing systems. *Distributed Computing*, 7(1) :17–26, 1993.

[56] Sandeep S. Kulkarni and Umamaheswaran Arumugam. Collision-free communication in sensor networks. In Shing-Tsaan Huang and Ted

Herman, editors, *Self-Stabilizing Systems, 6th International Symposium, SSS 2003*, volume 2704 of *Lecture Notes in Computer Science*, pages 17–31. Springer, 2003.

[57] Sandeep S. Kulkarni and Umamaheswaran Arumugam. Transformations for write-all-with-collision model. In Marina Papatriantafilou and Philippe Hunel, editors, *Principles of Distributed Systems, 7th International Conference, OPODIS 2003*, volume 3144 of *Lecture Notes in Computer Science*, pages 184–197. Springer, 2004.

[58] Sandeep S. Kulkarni, John M. Rushby, and Natarajan Shankar. A case-study in component-based mechanical verification of fault-tolerant programs. In Anish Arora, editor, *1999 ICDCS Workshop on Self-stabilizing Systems, Austin, Texas, June 5, 1999, Proceedings*, pages 33–40. IEEE Computer Society, 1999.

[59] Shay Kutten and Boaz Patt-Shamir. Stabilizing time-adaptive protocols. *Theor. Comput. Sci.*, 220(1) :93–111, 1999.

[60] Shay Kutten and David Peleg. Fault-local distributed mending. *J. Algorithms*, 30(1) :144–165, 1999.

[61] Toshimitsu Masuzawa and Sébastien Tixeuil. A self-stabilizing link coloring algorithm resilient to unbounded byzantine faults in arbitrary networks. In *Proceedings of OPODIS 2005*, Lecture Notes in Computer Science, page to appear, Pisa, Italy, December 2005. Springer-Verlag.

[62] Nathalie Mitton, Eric Fleury, Isabelle Guérin-Lassous, and Sébastien Tixeuil. Self-stabilization in self-organized wireless multihop networks. In *Proceedings of the 25th IEEE International Conference on Distributed Computing Systems Workshops (WWAN'05)*, pages 909–915, Columbus, Ohio, USA, June 2005. IEEE Press.

[63] Mikhail Nesterenko and Anish Arora. Tolerance to unbounded byzantine faults. In *21st Symposium on Reliable Distributed Systems (SRDS 2002)*, pages 22–. IEEE Computer Society, 2002.

[64] I. S. W. B. Prasetya. Mechanically verified self-stabilizing hierarchical algorithms. In Ed Brinksma, editor, *Tools and Algorithms for Construction and Analysis of Systems, Third International Workshop, TACAS '97, Enschede, The Netherlands, April 2-4, 1997, Proceedings*, volume 1217 of *Lecture Notes in Computer Science*, pages 399–415. Springer, 1997.

[65] Michel Raynal. A simple taxonomy for distributed mutual exclusion algorithms. *Operating Systems Review*, 25(2) :47–50, 1991.

[66] Michel Raynal. A short introduction to failure detectors for asynchronous distributed systems. *SIGACT News*, 36(1), March 2005.

[67] Laurent Rosaz. Self-stabilizing token circulation on asynchronous uniform unidirectional rings. In *Proceedings of the ACM Conference on Principles of Distributed Computing (PODC 2000)*, pages 249–258, 2000.

[68] Yusuke Sakurai, Fukuhito Ooshita, and Toshimitsu Masuzawa. A self-stabilizing link-coloring protocol resilient to byzantine faults in tree networks. In *Principles of Distributed Systems, 8th International Conference, OPODIS 2004*, volume 3544 of *Lecture Notes in Computer Science*, pages 283–298. Springer, 2005.

[69] Oliver E. Theel and Felix C. Gärtner. An exercise in proving convergence through transfer functions. In Anish Arora, editor, *1999 ICDCS Workshop on Self-stabilizing Systems, Austin, Texas, June 5, 1999, Proceedings*, pages 41–47. IEEE Computer Society, 1999.

[70] Sébastien Tixeuil. *Auto-stabilisation Efficace*. PhD thesis, University of Paris Sud XI, January 2000.

[71] George Varghese and Mahesh Jayaram. The fault span of crash failures. *J. ACM*, 47(2) :244–293, 2000.