Auto-stabilisation : Solution Elégante pour Lutter contre Les Fautes

Sylvie Delaët

Auto-stabilisation :
Solution Elégante pour Lutter contre Les Fautes

Sylvie Delaët

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

Soutenu le 8 novembre 2013 devant le jury composé de :

Alain Denise, Professeur, Université Paris Sud, France

Shlomi Dolev, Professeur, Ben Gurion University of the Negev, Israël

Pierre Fraigniaud, Directeur de recherche CNRS, France

Ted Herman, Professeur, University of Iowa, Etats Unis d'Amérique

Nicolas Thiéry, Professeur, l'Université Paris Sud, France

Franck Petit, Professeur, Université Paris Pierre et Marie Curie, France

Après avis des rapporteurs :

Ted Herman, Professeur, University of Iowa, Etats Unis d'Amérique

Rachid Guerraoui, Professeur, Ecole Polytechnique Fédérale de Lausanne, Suisse

Franck Petit, Université Paris Pierre et Marie Curie, France

Remerciements

Chaque début d'écriture est un retour à la case départ. Et la case départ, c'est un endroit où l'on se sent très seul. Un endroit où aucun de vos accomplissements passés ne compte.

Quentin Tarantino

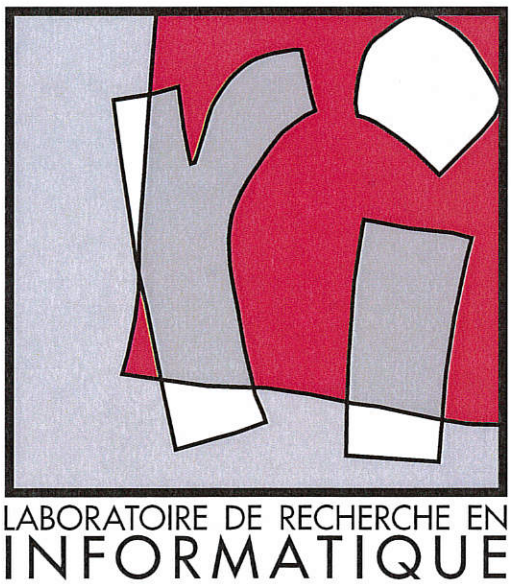


FIGURE 1 – Case Départ.

La tradition veut qu'on remercie les gens qui ont contribué à l'aboutissement de ce rapport scientifique, même si on souhaite encore intéragir avec eux.

Merci à Ted Herman, Rachid Guerraoui et Franck Petit pour avoir accepté de prendre le temps d'évaluer ce manuscrit.

Merci aux membres du jury : Alain Denise, Shlomi Dolev, Pierre Fraigniaud, Ted Herman, Nicolas Thiéry et Franck Petit d'avoir accepté d'être présents à la soutenance physiquement ou via le réseau.

Merci à l'ensemble de mes co-auteurs (dans l'ordre choisi par DBLP) : Sébastien Tixeuil, Joffroy Beauquier, Shlomi Dolev, Mikhail Nesterenko, Péva Blanchard, Stéphane Devismes, Partha Sarathi Mandal, Mariusz Rokicki, Bertrand Ducourthial, Olivier Pérès, Janna Burman, Adnan Vora, Sammy Haddad, Stéphane Cordier,

Duy Le Nguyen et Johanne Cohen pour leur collaboration productive.

Ma reconnaissance va plus largement à l'ensemble de mes collègues scientifiques, administratifs et techniques, de mes étudiants, de mes enseignants, de mes amis, de ma famille.

Ma personnalité veut que j'ajoute ici un bestiaire plus intime pour souligner le bonheur d'avoir achevé ce document et les soutiens dont j'ai bénéficié. On remarquera que ce bestiaire est bien plus féminin que la partie qui le précède (ce qui rétablit la parité violée par la tradition):

. Merci à Bianca et Quicky à qui je pense tous les jours puisqu'elles me prêtent leur mot de passe.

. Merci à Franck d'avoir apprécié ce poisson et d'avoir plongé dans ce bocal où requins, anguilles et dauphins décident de la vie des autres.

. Merci à tous ceux qui ont fait leur travail normalement ou un peu plus pour que ce petit monde universitaire ressemble plus à l'humanité qu'à la jungle:

- Laurence qui a su libérer ma plume et ma parole;
- Emmanuelle qui a soigner certains maux pour que j'écrive ces mots;
- Marianne et Dominique qui montrent que l'amitié dans le travail existe;
- Nicole qui prouve qu'avec la retraite les secrets s'estompent;
- Géraldine et Myriam qui m'offrent depuis 18 ans leur sourire en me vendant un repas quotidien;
- L'équipe Baradiski qui prépare les pistes, en ferme certaines mais sait tendre les bonnes perches au bon moment;
- Véronique qui m'a permis d'être suffisamment en confiance pour que je vole le temps de rédiger ainsi que toutes celles et ceux qui me remplacent auprès de mes enfants depuis leur naissance pour que je fasse "avancer la science".

Enfin, et pour être tout à fait exhaustive, il me reste à remercier cette personne à qui je dois tant et qui se reconnaîtra.

Merci Papa, Merci Maman

Merci HAL vous qui dirigez ma vie...

Je dédie cette réussite aux Max Delaët <3

Sylvie

Sommaire

La consigne n'a pas changé, dit l'allumeur. C'est bien là le drame ! La planète d'année en année a tourné de plus en plus vite, et la consigne n'a pas changé !

Antoine de Saint-Exupéry

Chapitre 1

Ma vision de ce document

Ce que l'homme appelle vérité, c'est toujours sa vérité, c'est-à-dire l'aspect sous lequel les choses lui apparaissent.

Protagoras

Mon premier contact avec la recherche fut la vaste question posée à l'occasion de mon stage de DEA : comment faire acquérir automatiquement une propriété à un algorithme réparti ? En effet, du fait de sa répartition sur des sites distants, du fait des problèmes liés aux modes de communication, l'algorithme réparti est vraiment différent de l'algorithme centralisé. Le seul fait d'initialiser un algorithme réparti ou encore de savoir dans quelle configuration il se trouve est un problème en soi. Jeune scientifique, j'ai abordé ce vaste et intéressant problème de manière cartésienne en énumérant des propriétés possibles des algorithmes répartis trouvées dans la littérature puis en étudiant comment elles pouvaient être acquises (ou pas) automatiquement. Ce travail méthodique de classification m'a menée à rencontrer au cours de ma thèse un concept intéressant : **l'auto-stabilisation** (self-stabilization). Un algorithme réparti est auto-stabilisant si, en dépit de mauvaises initialisations, malgré toutes les pannes transitoires (possibles et aussi étendues que possibles), il profite des moments sans perturbation pour recommencer à se comporter correctement. Ce qui apparaissait à l'époque comme une simple propriété initialement introduite par Dijsktra en 1974 [Dij74] est en fait, de mon point de vue, un concept primordial de l'algorithmique répartie. Je défendais en 1995 la première thèse française sur le sujet [Del95] en présentant à la fois un modèle pour l'auto-stabilisation et une application à l'exclusion mutuelle [1]. Depuis, de nombreux travaux ont suivi et la propriété d'auto-stabilisation est aujourd'hui devenue un domaine de recherche à part entière dont le symposium annuel se réunit pour la quatorzième fois cette année et qui figure dans l'appel à communications de nombreuses et prestigieuses conférences.

1.1 Organisation du document

Je vais consacrer les pages de ce document à présenter un sous-ensemble des travaux de recherche que j'ai effectué au cours des vingt dernières années et que j'estime aujourd'hui pertinent et cohérent.

L'Université Paris-Sud, qui il y a deux ans s'est dotée d'une image et d'un slogan, m'a autorisée à présenter mon mémoiret d'≪ Habilitation à Diriger les Recherches ≫ à trois rapporteurs de son choix pour, peut-être, avoir l'honneur d'y encadrer officiellement de ses doctorants, voire être, peut-être, amenée un

1. 1974+21=1995, 1995+18=2013. En 1974, l'âge de la majorité était de 21 ans. En 1995, l'âge de la majorité était de 18 ans. En 2013, il faut toujours avoir 18 ans pour se croire adulte.

FIGURE 1.1 – Université Paris-Sud en image.

jour à la quitter. En son hommage, je vais organiser ce manuscrit autour de son slogan : « comprendre le monde, construire l'avenir ».

Dans la partie « *Comprendre le monde* » (chapitre 3 page 15), je présenterai, en trois parties, et à la lueur de quelques-uns de mes travaux, le domaine des recherches que j'ai la chance d'effectuer en son nom.

Dans la partie « *Construire l'avenir* » (chapitre 4 page 43) je présenterai, en quatre sections, toujours éclairées de quelques-uns de mes travaux, les différentes voies qui intéressent aujourd'hui des chercheurs dont je suis ou intéresseront demain des chercheurs, des ingénieurs, des techniciens, des êtres humains dont je serai sûrement.

En référence au travail de l'allumeur de réverbère(s) de Saint-Exupéry [dSE99], ce document est organisé selon un mouvement argumentatif allant crescendo, du plus intuitif au plus technique, commençant par ce chapitre introductif nommé « *Ma vision de ce document* », numéroté 1 et ne comportant qu'une section jusqu'à la liste de mes publications, classées par type selon six sections, qui forme l'annexe B sous le titre « *Biographie de l'auteur* » .

J'invite également le lecteur, selon son érudition, à s'attarder sur les deux sections de vulgarisation du chapitre 2 page ci-contre intitulé « *Un peu de cuisine(s)* » ou sur les cinq articles agrafés dans la cinquième partie, l'annexe A page 51. En effet, si ce manuscrit n'était réservé qu'aux rapporteurs spécialistes de l'algorithmique répartie l'annexe A comme Agrafage suffirait mais j'ai la prétention que tout lecteur en tire quelque chose : un sourire, un savoir voire la saveur du savoir...

Bonne lecture.

Chapitre 2

Un peu de cuisine(s)

Il faut avoir déjà beaucoup appris de choses pour savoir demander ce qu'on ne sait pas.

Jean-Jacques Rousseau

C'est la fête alors on va cuisiner. Mais quelle cuisine ?

- La cuisine de Maman seule devant son fourneau qui s'organise pour offrir un repas à sa famille ;
- La cuisine de Jacques Borel où des commis travaillent à la chaîne pour fournir les uns après les autres des milliers de repas chauds au bord de l'autoroute ;
- La cuisine du grand restaurant de Bernard Loiseau où, des pâtissiers aux sauciers, tous s'activent sous les ordres de l'unique Chef afin de servir de l'art culinaire à moins de convives que de personnel ;
- La cuisine de l'auberge espagnole où chacun fabrique sa spécialité à son rythme pour que, au fil des échanges, des repas soient répartis entre les randonneurs affamés.

Et que se passe-t-il si des défaillances surviennent ?

- Si Maman a de la fièvre, au mieux le repas sera mauvais, au pire il sera inexistant mais cela n'affectera que les cinq membres de sa famille [1].
- Si un des commis de Jacques Borel a des absences, c'est toute la chaîne qui en pâtit et les milliers de repas manquent de qualité, de quantité voire ne sont même pas consommables (ce qui peut occasionner des conséquences dramatiques sur l'autoroute avec des conducteurs malades ou sous-alimentés qui provoquent des accidents).
- Si le rôtisseur du grand restaurant se fait « porter pâle », seules les œuvres sans rôtis pourront être abouties, les autres tomberont à l'eau. La qualité de service s'en ressentira gravement mais comme l'art est partout, le repas, s'il est servi, restera une œuvre culinaire.
- Dans l'auberge espagnole, il est normal de ne pas être présent ou efficace à tout moment, mais si tout le monde fait de son mieux, tous les randonneurs (philosophes [Dij68, CM84, Lyn80] ou non) mangeront à leur faim.

Selon la Taxinomie de Flynn [Fly72], il en va de l'organisation de nos ordinateurs comme de celle de nos cuisines :

- La cuisine de maman, c'est l'ordinateur monoprocesseur : une seule instruction est traitée à la fois sur une seule donnée (système SISD). Si une défaillance survient, elle affecte les compétences de l'unique processeur. La portée de l'erreur est restreinte. Constater la défaillance et remettre en route

1. Il est bien connu qu'une maman a trois enfants et qu'elle conçoit toujours et exclusivement des repas pour elle, ses enfants et leur père.

le système peut être relativement simple. Les algorithmes exécutés peuvent supporter une initialisation. Le comportement du système, en l'absence de recours volontaire à une unité aléatoire, est entièrement déterministe. Les données entrées dans l'ordinateur déterminent entièrement les sorties qu'on en obtient. C'est le modèle de la machine de Turing qui reste la base de tout calcul de complexité dans le monde séquentiel.
- La chaîne de production de Jacques Borel représente les ordinateurs vectoriels (SIMD). Dans ces systèmes les données « avancent » en subissant l'instruction unique des traitements de plusieurs unités successives. On retrouve ce type d'architecture dans la littérature sous le nom de pipeline ou de GPU par exemple et la défaillance entraîne un défaut de qualité de service sur toute la chaîne de production.
- Le fonctionnement du grand restaurant est plutôt utilisé dans les systèmes embarqués où de toutes petites unités spécialisées traitent des données (MISD).
- Les architectures MIMD des super-calculateurs actuels, des grands réseaux de communication, des réseaux de capteurs ont en commun avec l'auberge espagnole le concept de répartition, c'est-à-dire qu'un ensemble distinct de données est traité simultanément par un ensemble distinct d'instructions. La collaboration entre ses unités se fait par des communications elles-mêmes plus ou moins fiables. Par ailleurs l'ordonnancement des actions élémentaires subit un contrôle centralisé ou lui aussi réparti.

Maintenant que nous avons vu le parallèle entre le monde des cuisines et les différents modèles du parallélisme, nous allons nous concentr sur la seule auberge espagnole et son organisation c'est-à-dire que nous nous intéresser exclusivement à l'algorithmique répartie.

2.1 L'auberge espagnole de l'algorithmique répartie

Après une description de l'auberge délimitée par des paragraphes sans numérotation, les sous-sections numérotées reprennent point par point les concepts soulevés au sein de l'auberge espagnole.

Dans notre auberge, les convives sont aussi les concepteurs des repas. Dans la suite du document, nous les appellerons des cuisiniers ou des randonneurs selon que nous nous intéresserons à la cuisine qu'ils produisent ou à la manière dont ils s'organisent à l'auberge et nous continuerons de les appeler des convives tant que le propos restera général.

Ce qu'il y a à faire. Pour ce qui est de l'organisation, elle dépend de la tâche à accomplir par les randonneurs qui la prennent en charge. Et tout est à faire, on peut citer l'exemple de la corvée de poubelle qui tourne entre les randonneurs, chacun devant la faire à son tour, ou du calcul du nombre de randonneurs dans le but de payer la taxe de séjour qui couvre entre autres les frais de gestion des ordures par la communauté urbaine.

Offre alimentaire. Il existe plusieurs manières de nourrir un convive affamé. Soit on laisse à sa disposition en permanence un buffet où il se sert à volonté, soit on lui fabrique un plat sur commande. Dans un cas comme dans l'autre les randonneurs doivent s'organiser en cuisine pour échanger les ingrédients, les idées, les demandes (ou commandes), les recettes. Les cuisiniers doivent exécuter des recettes pour fournir les plats mis à disposition des convives ou répondant à leur demande.

Possibilité de communiquer. Les possibilités de communication entre convives sont importantes. Dans l'auberge, il est peut-être possible que tous les convives transmettent des informations ou des ingrédients et en reçoivent de tous les autres. Ce n'est plus le cas si à cause de problèmes de langue ou d'organisation spatiale de l'auberge certains convives peuvent n'obtenir des informations que de certains autres et réciproquement ne transmettre des informations qu'à certains autres (et pas nécessairement les mêmes).

Mode de communication. La manière dont nos convives communiquent est également très importante.

- Est-ce par la parole ? Auquel cas toute communication est instantanée mais quelques perturbations peuvent survenir (par exemple si un bruit de casserole qui tombe couvre une demande de fleur de sel qu'on veut ajouter en fin de cuisson d'une viande saignante, la viande restera sans sel).
- Est-ce par le dépôt de messages sur des tableaux noirs situés à des points stratégiques de l'auberge ? Auquel cas certaines demandes peuvent être perdues car effacées avant d'avoir été consultées.
- Est-ce par des SMS ? Auquel cas tous les problèmes de compatibilité entre les opérateurs téléphoniques possibles peuvent amener les messages à être perdus, à arriver en double, ou à arriver dans un ordre différent de celui dans lequel ils ont été envoyés.

Ordonnancement des actions. La qualité du repas peut aussi dépendre de l'ordonnancement des interventions des convives : ainsi, si tous travaillent de concert, on ne pourra pas cuisiner les mêmes plats que si certains cuisinent vite tandis que d'autres traînent. Le soufflé au fromage [2] par exemple nécessite d'être servi immédiatement à sa sortie du four. Avec les mêmes ingrédients le quatre-quarts fromager [3] peut tolérer que les convives arrivent à table à des moments différents.

Genre des fautes. Il est possible que certains convives arrivent à l'improviste. Il est possible que certains convives meurent ou quittent définitivement l'auberge sans prévenir (tout le monde se souvient de l'histoire de Bianca disparue alors même qu'elle s'était engagée à faire une sauce béchamel pour agrémenter les coulemelles ramassées par ses amis...). Il est possible que les lendemains de fêtes trop arrosées, certains, voire tous les convives, soient dans des états perturbés ou adoptent des comportements passagers inadaptés. Il est possible que certains convives soient des mythomanes (le dernier cuisinier qui a dit avoir pêché un saumon de 30 kg a laissé espérer aux quatre-vingt-dix neuf autres cuisiniers [4] un merveilleux repas de saumon mais l'affabulateur a été démasqué [5], tout le monde a mangé du thon blanc en boîte, car le thon blanc c'est excellent...). Tous ces états ou comportements déplorables peuvent affecter la production de repas ou désorganiser le service.

2.1.1 Acteurs

Dans notre parallèle avec l'algorithmique répartie nous assimilerons les convives à des **unités de calcul autonomes** qui communiquent les unes avec les autres

2. Recette disponible en page 104.
3. Recette disponible en page 103.
4. On compte en moyenne une portion de 300 grammes de saumon par personne et par repas.
5. La plus grande variété de saumon pèse 18kg.

dans le but de résoudre des problèmes. Les données entrantes du système réparti sont alors les ingrédients, les données sortantes, les plats.

2.1.2 Spécification

Le problème à résoudre est représenté par une **spécification**, plus ou moins formelle, sur le comportement attendu du système. Les problèmes étudiés sont de deux types : assurer que des plats sont bien conçus en fonction des ingrédients fournis en entrée (c'est-à-dire effectuer un calcul, en fonction d'un flux de données entrant) ou assurer qu'un comportement est respecté au sein de l'auberge (c'est-à-dire assurer un service). Les spécifications sont qualifiées de **silencieuses** (on trouve aussi **statiques** dans la littérature) ou de **dynamiques** selon qu'elles portent sur un calcul à effectuer (par exemple, calculer le nombre de randonneurs) ou sur un service à fournir (par exemple, s'assurer que la corvée de vidage des poubelles tourne équitablement entre les randonneurs). Dans le premier cas, c'est le résultat obtenu à la fin du calcul qui importe, dans le deuxième cas, c'est la manière d'y parvenir qui est la plus importante. La spécification silencieuse implique un résultat ultimement figé (même si celui-ci est en permanence recalculé dans le cadre auto-stabilisant pour assurer qu'il soit valide après une panne transitoire, voir Section 3.3.3 page 36). Par contre, une spécification dynamique induit un comportement et le système doit donc passer par plusieurs configurations (c'est-à-dire par plusieurs états globaux) jusqu'à ce que le service soit satisfait (peut-être indéfiniment d'ailleurs). Les noms de statique et de dynamique font référence à cet aspect figé [6] ou mouvant [7] des spécifications [Lyn96].

2.1.3 Expression des spécifications

Que la spécification d'un problème soit silencieuse ou dynamique, il est possible de l'exprimer de plusieurs manières.

Le buffet laissé à la disposition des convives en permanence où ils se servent à volonté dont on parle au paragraphe « *Offre alimentaire* » page 4 représente les spécifications que je nommerai **passives** dans la suite de ce document. Les spcifications passive sont celles où on fournit en permanence un service, ou le résultat d'un calcul, et l'utilisateur en profite à sa convenance. Le plat fabriqué

6. Exemple de résultat d'une spécification statique recalculé en permanence : il y a **42** randonneurs dans l'auberge.

7. Exemple de résultat d'une configuration dynamique, le résultat change en permanence : c'est **Anish** qui doit sortir les poubelles, c'est **Andre** qui doit sortir les poubelles, c'est **Alex** qui doit sortir les poubelles, c'est **Ajoy** qui doit sortir les poubelles, c'est **Baruch** qui doit sortir les poubelles, c'est **Christian** qui doit sortir les poubelles, c'est **David** qui doit sortir les poubelles, c'est **Edsger** qui doit sortir les poubelles, c'est **Franck** qui doit sortir les poubelles, c'est **Guevara** qui doit sortir les poubelles, c'est **Hagit** qui doit sortir les poubelles, c'est **Isabelle** qui doit sortir les poubelles, c'est **Jennifer** qui doit sortir les poubelles, c'est **Koichi** qui doit sortir les poubelles, c'est **Leslie** qui doit sortir les poubelles, c'est **Maria** qui doit sortir les poubelles, c'est **Mikhail** qui doit sortir les poubelles, c'est **Nancy** qui doit sortir les poubelles, c'est **Olivier** qui doit sortir les poubelles, c'est **Pierre** qui doit sortir les poubelles, c'est **Quentin** qui doit sortir les poubelles, c'est **Rachid** qui doit sortir les poubelles, c'est **Sandeep** qui doit sortir les poubelles, c'est **Sébastien** qui doit sortir les poubelles, c'est **Seth** qui doit sortir les poubelles, c'est **Shlomi** qui doit sortir les poubelles, c'est **Shinn-Tsaan** qui doit sortir les poubelles, c'est **Sylvie** qui doit sortir les poubelles, c'est **Shay** qui doit sortir les poubelles, c'est **Shmuel** qui doit sortir les poubelles, c'est **Stéphane** qui doit sortir les poubelles, c'est **Sayaka** qui doit sortir les poubelles, c'est **Sukumar** qui doit sortir les poubelles, c'est **Ted** qui doit sortir les poubelles, c'est **Toshimizu** qui doit sortir les poubelles, c'est **Taisuke** qui doit sortir les poubelles, c'est **Ulrich** qui doit sortir les poubelles, c'est **Vincent** qui doit sortir les poubelles, c'est **Wei** qui doit sortir les poubelles, c'est **Xavier** qui doit sortir les poubelles, c'est **Yukiko** qui doit sortir les poubelles, c'est **Zohir** qui doit sortir les poubelles, c'est **Anish** qui doit sortir les poubelles...

sur demande d'un utilisateur correspond au cas où le service n'est fourni que sur requête. J'appellerai ce type de spécification **active**.

La distinction entre ces deux manières d'exprimer une spécification est importante. D'une part, elle impacte les performances de l'algorithme qui résoud la spécification [8]. D'autre part, en présence de défaillances, une spécification active est trivialement satisfaite tant que le service n'est pas utilisé, alors que l'invariant maintenu pour une spécification passive peut être compromis.

2.1.4 Graphe de communication

En algorithmique répartie, le modèle de communication est le plus souvent représenté par un graphe dont les sommets sont des unités de calcul autonomes (qu'elles soient convives, cuisiniers ou randonneurs) et les arêtes sont des possibilités de communiquer directement. Si tous les convives peuvent parfois transmettre des informations et recevoir des informations de tous les autres sans intermédiaire, le graphe est **complet**. Si toutes les communications sont réciproques, le **graphe de communication** admet simplement des arêtes, mais si certaines communications point à point entre deux convives ne se font que dans un sens alors le graphe est constitué d'arcs orientés. La modélisation par un **graphe orienté** est donc évidemment plus générale que celle selon un **graphe non orienté**. Un critère de complexité spécifique aux systèmes répartis, celui de la *localité* [Pel00] : « à quelle distance faut-il communiquer pour résoudre un problème ? », utilise le graphe des communications.

2.1.5 Modèle de communication

Une fois connu le graphe de communication, on s'intéresse à la manière dont les communications ont lieu. Ce passage fait écho au paragraphe « *Mode de communication* » (page 5). La communication par la parole est une communication synchrone : celui qui parle et celui(ceux) qui écoute(nt) sont synchronisés et partagent la même information. Le dépôt de messages sur des tableaux noirs est la communication asynchrone par **registre partagé** [Ray13b] entre au moins deux participants. Le mode de communication le plus général est celui représenté par les SMS c'est-à-dire une communication par **passage de messages** [Ray13a] asynchrone.

2.1.6 Démon

En algorithmique répartie et plus particulièrement dans le domaine de l'auto-stabilisation, les problèmes d'ordonnancement (voir paragraphe « *Ordonnancement des actions* », page 5) sont modélisés à l'aide du **démon**. Le démon est une sorte de grand ordonnanceur imaginaire qui aurait le pouvoir d'autoriser chaque convive à travailler ou de lui interdire de le faire. Ce démon est vu, pour les preuves d'algorithmes, comme un ennemi qui veut empêcher l'algorithme de fonctionner correctement. La puissance du démon est limitée par les propriétés des machines ou par le caractère déterministe des exécutions qu'on veut considérer. Quelques démons usuels dans le cadre de l'auto-stabilisation sont [DT11] : le **démon central** [Dij74] où l'exécution ne concerne qu'un acteur à la fois ; le **démon synchrone** [Her90] qui impose que tout le monde travaille obligatoirement à l'unisson ; le **démon réparti** [KY97] où un sous-ensemble des participants

8. On peut faire l'analogie entre les spécifications passives et actives d'une part, et les protocoles proactifs et réactifs de routage dans les réseaux sans fil d'autre part : un protocole proactif [CJ003] maintient des tables de routage en permanence pour permettre des décisions immédiates d'acheminement de messages ; un protocole réactif [PBRD03] calcule un chemin lorsqu'un message a effectivement besoin d'être diffusé.

prend part à l'action simultanément ; le **démon probabiliste** [DTY08] dont le choix à chaque étape des acteurs potentiels est fait selon une loi de probabilité.

2.1.7 Pannes

Les fautes (pannes et attaques) sont naturellement imprévisibles. En conséquence, concevoir un système sans prendre en compte les fautes susceptibles de survenir et s'attendre à ce qu'il y résiste spontanément, est un rêve aussi stupide et inaccessible que d'avoir des enfants en permanence sages. En effet, un système à la fois mal programmé et susceptible d'être attaqué par des ennemis tout puissants ne fonctionnera jamais. Tolérer les fautes c'est s'attendre à ce qu'il en surviennent et accepter que certaines spécifications soient momentanément violées. Pour tolérer des fautes, il faut donc les modéliser et les intégrer à la spécification qui sera effectivement satisfaite. Je vais en rappeler quatre modélisations [Tix06] : les **pannes crash** modélisent les cas où l'unité de calcul disparaît tout simplement et tous les calculs en cours la concernant sont, de fait, interrompus. Ce sont ces pannes qui sont considérées dans la littérature classique [Lyn96, AW98] sur la robustesse [9]. Les **pannes transitoires** corrompent la mémoire du système de manière plus ou moins étendue mais sont assez rares pour qu'on puisse les modéliser en supposant qu'elles n'existent plus après un certain moment. Ce sont les pannes qui sont considérées dans le domaine de l'auto-stabilisation [Dij74, Dol00, Tix09]. Les **pannes intermittentes** [DT02] regroupent en particulier les duplications, déséquensements et pertes de messages. Les **pannes Byzantines** [LSP82, Dol82] modélisent les malveillances ou les attaques ; elles ont la puissance maximale, et sont le plus souvent liées à des preuves d'impossibilité.

2.2 Exemples de problèmes à résoudre et quelques solutions

Dans cette section, je prends pour trame des anecdotes de l'auberge espagnole, sous forme d'un chapelet de paragraphes non numérotés, puis suit une succession de sous sections numérotées où chaque fonctionnement présenté au sein de l'auberge espagnole est traduit par son pendant en algorithmique répartie. Les différents problèmes qui sont traités ou qui illustrent les résultats présents dans les articles du chapitre « *Agrafage* » (page 51) et ceux assez substentiels pour prendre place dans le chapitre « *Construire l'avenir* » (page 43) vont ainsi être abordés.

Assemblées Consultatives de Randonneurs Organisateurs « Natures Y Mimant Et Slamant ». Devant notre auberge se trouve une table de concertation entre les randonneurs. Elle est installée là depuis le fondement de l'auberge ; on y échange et on y grave les lois qui font fonctionner l'auberge. À cette table, on veut assurer que, le plus souvent, la politesse est respectée c'est-à-dire qu'un seul randonneur s'exprime à la fois. Les randonneurs vont et viennent autour de la table, s'expriment, écoutent les autres, s'expriment à nouveau et écoutent les autres autant de fois qu'ils le désirent et quittent la table quand ils le souhaitent. Autour de cette table il y a toujours du monde et chacun prend la parole. Pour assurer qu'un seul randonneur parle à la fois un bâton de parole circule autour de la table [Dij74]. Si un randonneur détient le bâton de parole, il peut parler ; dès qu'il le lâche, il perd le privilège de pouvoir s'exprimer jusqu'au moment où

9. En particulier l'article [FLP85] prouve que lorsque le système est asynchrone, le problème du consensus est impossible à résoudre même en présence d'une seule panne crash.

il récupère un bâton de parole. Avant de quitter la table ou lorsqu'il a fini de parler, tout randonneur transmet le bâton de parole à son voisin de table. Tout randonneur qui se trouve en possession de deux bâtons de parole en jette un dans la forêt. Tout randonneur qui rejoint la table de concertation le fait avec un bâton de parole en main qu'il ramasse au préalable dans la forêt. À certains moments, suite à l'entrée de quelqu'un ou parce qu'un malicieux casse le bâton en deux pour n'en transmettre que la moitié, plusieurs personnes parlent à la fois, mais le plus souvent, il n'y qu'une parole émise. Il est important de comprendre que même si l'objectif est *global* (un seul randonneur parle à la fois autour de la table, et tout randonneur qui le souhaite finit par pouvoir parler), les interactions techniques entre randonneurs (je te passe le bâton ou je reçois un bâton de toi) sont seulement *locales*.

Démocratie, Anarchie, Gérontocratie. L'anarchie est une jolie idée mais a souvent un prix non négligeable [10]. Choisir un chef [Lan77, IR81] peut être parfois très utile, par exemple pour avoir quelqu'un pour apaiser les conflits entre cuisiniers portant sur le problème primordial de la dose de sel qu'il faut saupoudrer sur une côte de bœuf sortant du grill ou le poids réel du saumon servi au prochain repas de Sancho Panza [dC05]. Il existe plusieurs méthodes pour choisir un chef. On peut décider que ce sera toujours le doyen d'âge (c'est la gérontocratie) ou on peut organiser des élections (c'est la démocratie). Dans le premier cas, c'est toujours la même personne qui est élue [AK93] même si on utilise des procédures différentes ou que l'on répète le procédé Dans le deuxième cas, deux procédures peuvent donner des résultats différents et deux exécutions indépendantes de la même procédure peuvent donner des résultats différents [BGJ07] (une contrainte peut même être que l'élection soit équitable, c'est-à-dire que tout le monde ait la même chance d'être élu).

Alerte « Oh Cheval ! ». Un scandale vient d'éclater à l'auberge : de la viande de cheval a été retrouvée dans les sauces précuisinées à base de viande de bœuf. Du coup il devient necéssaire de retirer des buffets tous les plats utilisant un tel ingrédient et il faut également stopper la production pour éviter leur consommation (et la tromperie) par les convives. Afin que tout le monde dans l'auberge soit prévenu, il est nécessaire de diffuser aussi fiablement, efficacement et rapidement que possible l'information et de n'accepter une nouvelle commande pour ce type de plats qu'une fois la chaîne de production sécurisée (c'est-à-dire une fois les ingrédients à base de viande de cheval détruits).

Groupement d'Intérêts. Les convives de notre auberge espagnole [Kla02] apprécient de s'organiser selon des communautés (ou des structures virtuelles) qui représentent leurs centres d'intérêts ; par exemple le Cercle des poètes disparus [Kle98], la Confrérie des Faiseurs d'Andouilles [Rab46] et autres arbres à spaghetti [Gri67]. Cette organisation ne repose pas nécessairement sur leur proximité géographique ou leurs compétences, mais facilite leur communication, leur bien-être ou bien la diffusion d'informations auxquelles ils portent un intérêt.

Auberge Espagnole des Randonneurs En Société. Il arrive que même une structure répartie comme celle de l'auberge espagnole doive produire des connaissances globales sur sa configuration courante, comme par exemple :

10. Le prix de l'anarchie est le sujet de la définition 26 page 51 du rapport d'Habilitation de Johanne Cohen [Coh09], c'est un aspect de la théorie des jeux sans corrélation avec le sujet présent mais très intéressant en soi.

1. Qui est présent à l'auberge aujourd'hui ?
2. Combien de personnes participent aux tâches communes ?
3. Quel est l'éventail des compétences des personnes présentes ?
4. Quel est l'état du stock d'ingrédients ?
5. Combien de commandes sont en souffrance ?

Collecter des informations réparties permet de répondre à ce genre de questions. Il importe bien sûr de veiller à l'*exactitude* (les informations concernant un individu sont correctes) et la *complétude* (les informations de tous les individus actuellement présents et seulement ceux-ci sont recensées) de ce qui est collecté, sans quoi l'intérêt de ces connaissances globales en serait fortement amoindri.

Fromage Ou Autres Desserts. Ce soir à l'auberge, c'est un repas spécial. Tout le monde se verra servi un plat identique, la paëlla. Par contre, pour respecter l'équilibre alimentaire il faut conclure ce repas par un produit laitier : fromage ou dessert ? Chacun a son avis mais une décision commune doit être prise. Pas facile de mettre tout le monde d'accord surtout si tout le monde ne comprend ou ne réfléchit à la même vitesse et si certains convives sont susceptibles de quitter l'auberge au beau milieu de ce processus de concertation et de prise de décision.

Régimes Alimentaires Spéciaux. Depuis peu à l'auberge on constate des cas d'allergies aux arachides, d'intolérance au gluten, des demandes religieuses de viande cachère ou hallal, des exigences de végétariens ou de végétaliens. Ainsi, un convive allergique aux arachides ne peut rester à proximité d'autres convives qui travaillent en utilisant des arachides, ni travailler avec des arachides lui-même. Il devient nécessaire de faire en sorte que de tels conflits entre voisins (qu'il soient issus d'impératifs médicaux ou confessionnels) soient réglés, si possible de manière locale.

2.2.1 Exclusion mutuelle

Comme la prise de parole autour de la table de concertation de l'auberge, le problème de l'exclusion mutuelle est défini selon deux exigences : (**vivacité**) tout participant qui veut entrer en section critique peut le faire après un temps d'attente fini et (**sûreté**) lorsqu'un participant est en section critique, il est le seul à le faire. Dans l'analogie de l'auberge espagnole, la section critique représente la prise de parole, l'autorisation d'entrer en section critique la détention du bâton de parole. Le mode de fonctionnement de la table de concertation expliqué dans le paragraphe « *Assemblées Consultatives de Randonneurs Organisateurs « Natures Y Mimant Et Slamant »* » page 8 est la traduction directe du premier algorithme auto-stabilisant d'exclusion mutuelle introduit par Dijsktra [Dij74]. Dans l'algorithme de Dijsktra, des processus organisés en anneau détiennent un entier et peuvent seulement lire l'entier de leur voisin de gauche et modifier le leur, les interactions sont seulement locales. Le privilège (le bâton de parole) est symbolisé par une différence entre l'entier détenu par le processus et l'entier détenu par son voisin (sauf pour un unique processus distingué où c'est l'égalité entre l'entier détenu par le processus et l'entier détenu par son voisin qui indique la présence du privilège). La taille de la mémoire de chaque processus doit être supérieure (en nombre d'états) au nombre de participants dans l'anneau. Quelle que soit la configuration initiale du système (c'est-à-dire quelles que soient les valeurs initiales des entiers détenus par chacun des processus), après un temps de convergence fini, un seul processus détient le privilège (Sûreté) et celui-ci circule indéfiniment entre les processus (vivacité). De plus, du moment que la taille de

l'entier reste cohérente avec le nombre de processus, les agents peuvent arriver ou quitter le système celui-ci atteindra après le temps de convergence une configuration correcte à partir de laquelle (en l'absence de nouvelle arrivée, de nouveau départ ou de nouvelle corruption transitoire de la mémoire) le système satisfait sa spécification (l'exclusion mutuelle).

La solution proposée par Dijkstra satisfait une spécification dont la formulation est passive. En effet, c'est un algorithme d'exclusion mutuelle à privilège où un privilège est disponible de manière permanente à qui veut l'utiliser ce qui est à distinguer des algorithmes d'exclusion mutuelle à demande de section critique (où un utilisateur demande l'entrée en section critique et où on garantit que s'il l'obtient il est seul à la détenir et qui l'obtiendra à chaque fois qu'il en fera la demande après une attente raisonnable). La spécification de l'exclusion mutuelle à demande de section critique est active. Dans tous les cas, une spécification d'exclusion mutuelle est une spécification dynamique : le privilégié ou le détenteur de section critique change périodiquement entraînant en permanence des changements de configurations.

Dans l'exemple présenté, le graphe de communication est en anneau. La communication est asynchrone par registre partagé entre deux processus voisins sur l'anneau. Le démon est réparti : plusieurs processus peuvent agir en même temps. Les pannes considérées sont des pannes transitoires aussi étendues que possible mais ne concernant que la mémoire. La solution proposée est auto-stabilisante.

2.2.2 Élection de leader

Choisir un chef, comme on l'aborde au paragraphe « *Démocratie, Anarchie, Gérontocratie* » page 9, peut avoir une spécification passive ou active. Avec un algorithme qui satisfait une spécification passive, chaque processus est capable sans communication de savoir s'il est le chef ou pas. Avec une spécification active, il peut être nécessaire d'effectuer une nouvelle élection (impliquant l'intégralité des participants) pour répondre à la question : « suis-je le chef ? » Dans le contexte de l'auto-stabilisation, la plupart des travaux pour l'élection d'un chef considèrent une spécification passive.

Une des solutions auto-stabilisantes les plus simples [AG94] pour élire un chef consiste à se baser sur les identifiants et à construire un arbre couvrant enraciné au processus dont l'identifiant est maximum dans le réseau. Chaque processus maintient ainsi l'identifiant maximum connu ainsi que sa distance au processus dont l'identifiant est maximum. Si mon propre identifiant correspond à l'identifiant maximum connu, je suis l'élu, sinon je ne suis pas élu. Corriger les distances erronées de manière auto-stabilisante peut se faire simplement : si je suis le processus dont l'identifiant est le maximum connu de moi et de mes voisins, alors la distance est 0 ; sinon mon identifiant maximum connu est le maximum connu de mes voisins, et ma distance à ce processus la distance minimale indiquée par mon (ou mes) voisin(s) plus 1. Le problème principal posé par cette approche est de garantir que l'on se débarrasse nécessairement des identifiants qui n'existent pas (et qui ont été introduits dans les mémoires de processus par une défaillance transitoire). Par exemple, si un identifiant inexistant mais plus grand que tous ceux qui existent effectivement sur un anneau apparaît, le calcul des distances tel qu'indiqué précédemment croît indéfiniment, et personne n'est élu. Il faut donc prévoir un mécanisme pour briser ces cycles infinis illégitimes. Si on dispose d'une borne supérieure sur le diamètre du réseau [11], il est possible de briser de tels

11. L'identifiant maximal connu peut constituer une bonne approximation de cette borne [BDT99] sur le diamètre puisque dans une configuration légitime, il est supérieur à la taille du réseau.

cycles en invalidant tout identifiant maximum connu dont la distance supposée est supérieure à la borne et en le remplaçant par une valeur qui correspond à un véritable processus (par exemple, son propre identifiant).

Dans l'exemple prédédent, la spécification est exprimée de manière passive, le graphe de communication est un graphe non-orienté connexe arbitraire, la communication est asynchrone par registres partagés entre processus voisins dans le graphe de communication. Les pannes considérées sont des pannes transitoires aussi étendues que possible mais ne concernant que la mémoire. La solution proposée (avec la borne) est auto-stabilisante.

2.2.3 Propagation d'information avec retour

Pour diffuser une information au sein d'une structure répartie comme notre auberge espagnole, on peut avoir l'utilité des algorithmes de propagation d'information avec retour. La spécification d'un tel problème est la suivante : un initiateur démarre une phase de diffusion en envoyant une information (dans notre exemple du paragraphe « *Alerte « Oh Cheval ! »* » page 9, l'information est que les chevaux se sont pris pour des bœufs) à tous les participants. Après avoir reçu l'information, les participants s'organisent pour accuser réception du message. La propagation d'information est considérée comme effectuée lorsque l'initiateur a reçu tous les accusés de réception.

Ce problème joue un rôle particulier dans le domaine de l'auto-stabilisation, notamment du fait que sa spécification est naturellement active (un initiateur lance la diffusion et détermine que le comportement attendu est terminé). Il constitue une sorte de « benchmark » pour la stabilisation instantanée [BCV03, CDPV03, CDV05b, CDV05a, BDV05, CDV06, CDPV06, BDPV07, CDPT07, CDV09b, CDV09a, CDL$^+$10, DDNT10] dont nous reparlerons plus en détails dans la section « *Comparaison avec d'autres modes de stabilisation* » page 26.

2.2.4 Construction de structure

De nombreux algorithmes répartis présupposent une structure particulière du graphe de communication sous-jacent comme un anneau [Dij74, BP89, BCD95], un DAG [DDT99] (*Directed Acyclic Graph* ou graphe orienté acyclique) ou un arbre [GGH$^+$95, BGKP99, HH03b]. À la manière de nos convives qui se regroupent et s'organisent selon leurs centres d'intérêts pour communiquer plus facilement dans l'histoire du paragraphe « *Groupement d'Intérêts* » page 9, de telles structures peuvent être virtuelles, avoir été construites et être maintenues par un autre algorithme.

Dans le cadre de l'auto-stabilisation, la maintenance de structures topologiques virtuelles se fait habituellement par l'intermédiaire de spécifications passives et joue un rôle considérable, que ce soit parce que de nombreux algorithmes auto-stabilisants sont conçus pour des topologies spécifiques [Dij74, BP89, BCD95, DDT99, GGH$^+$95, BGKP99, HH03b], ou que l'enjeu même du problème consiste à construire des infrastructures pertinentes pour l'organisation globale, notamment dans le cadre des réseaux pair-à-pair [CDPT07, DK08]. Plusieurs de mes travaux, et en particulier ceux liés aux r-opérateurs, permettent la construction de structures arborescentes diverses de manière automatiquement auto-stabilisante. Plus de détails seront donnés dans la section « *Preuve d'auto-stabilisation* ».

2.2.5 Recensement

Une solution possible pour répondre aux questions du paragraphe « *Auberge Espagnole des Randonneurs En Société* » page 9 est d'effectuer un recensement.

Le recensement (*census* en anglais) consiste à recueillir localement, sur chacun des nœuds du système, l'image globale du système. Défini à l'origine pour connaître l'identité de tous les participants au réseau sur lequel il fonctionne, un algorithme de recensement est en fait beaucoup plus puissant si, à cette identité, on adjoint une valeur propre au nœud, puisqu'il permet alors qu'une information locale liée à un participant devienne une information globale dans tout le réseau.

L'essence même du recensement va à l'encontre des principes du réparti, puisqu'on globalise des informations qui sont, au départ, éparpillées. Il a cependant une double utilité :

1. Il permet la réutilisation d'une solution connue dans le domaine séquentiel dans un contexte réparti. Il suffit de collecter toute l'information nécessaire et d'exécuter l'algorithme séquentiel sur les informations collectées. La solution obtenue concernant le nœud qui a effectué le calcul est alors utilisée comme sortie de l'algorihtme réparti pour ce nœud [12].
2. Il permet d'avoir rapidement des bornes supérieures en temps et en espace pour la résolution d'un problème dans un contexte réparti voire auto-stabilisant. Si la solution *ad hoc* envisagée est moins performante que la solution immédiatement obtenue à partir d'un algorithme de recensement, on peut l'écarter.

La section « *Passage de messages* » page 36 reviendra sur l'une de mes contributions : un algorithme de recensement auto-stabilisant sur des graphes de communication arbitraires et possiblement orientés, qui tolère de surcroît les pertes, duplications, et déséquencements de messages.

2.2.6 Consensus

Comme dans le paragraphe « *Fromage Ou Autres Desserts* », le problème du consensus est celui de prendre une décision. Il est défini par l'apport de chacun des participants d'une proposition (par exemple binaire, 0 ou 1) suivi d'une décision de tous les participants sur l'une des propositions comme valeur de décision. Ce choix doit respecter les contraintes suivantes :

- (**Terminaison.**) Tout les participants doivent décider une valeur ;
- (**Accord.**) Tous les décideurs doivent décider de la même façon (c'est-à-dire qu'ils doivent décider la même valeur) ;
- (**Intégrité**) La valeur décidée doit être l'une de celles proposées.

L'un des résultats fondateurs en algorithmique répartie (et lauréat du Prix Dijkstra en 2001) est dû à Michael J. Fischer, Nancy A. Lynch et Mike Paterson qui ont prouvé [FLP85] que dans un système réparti asynchrone, l'éventualité d'une unique panne définitive rend le consensus impossible.

Ce résultat a suscité une intense dynamique de recherche, toujours très soutenue, pour tenter de contourner cette impossibilité. Dans le cadre des systèmes qui peuvent être sujets à des pannes crash définitives et à des défaillances transitoires, la terminaison explicite de tous les participants corrects est impossible (si la mémoire est initialement corrompue pour deux processus sur deux valeurs distinctes et qu'elle leur indique qu'une décision irrévocable a été prise, la spécification d'origine ne peut trivialement pas être satisfaite). Un résultat récent [DKS10] établit une possibilité de connexion entre le consensus, qui implique une terminaison, et l'auto-stabilisation, qui ne peut proposer de termiaison que sous forme

12. Le protocole OSPF [Moy98] pour le routage intradomaine dans Internet est fondé sur ce principe : toutes les relations d'adjacence entre les nœuds du réseau sont diffusées à tous les participants, le graphe est reconstruit sur chaque nœud, et un protocole séquentiel de calcul des plus courts chemins est exécuté sur le graphe résultant. Les entrées correspondant aux autres nœuds du réseau sont mises à jour en conséquence sur le nœud courant.

d'algorithme silencieux. Il considère une série répétée et infinie d'appels au consensus, dont un suffixe satisfait la propriété initiale (accord, terminaison, intégrité). Une autre approche [AFJ06] est de relâcher la condition de terminaison pour se satisfaire d'une terminaison implicite (il n'y a plus de décision ferme mais une terminaison implicite si le même accord est maintenu à l'infini).

2.2.7 Coloration

Colorier le graphe de communication, c'est simplement associer à chaque sommet une couleur de telle manière que deux sommets adjacents n'aient pas la même couleur. Tout graphe est coloriable avec $\delta+1$ couleurs (o δ est le degr maximal des sommets. Ce problème, que nous avons mis dans notre auberge au service de supposés régimes alimentaires spéciaux (page 10), est particulièrement intéressant dans le contexte réparti des réseaux de communication sans fil avec allocations de fréquences [MLG06] (deux émetteurs proches l'un de l'autre doivent avoir des fréquences distinctes pour que les communications restent de qualité) ou de créneaux de temps [HT04] (à chaque couleur correspond un créneau temporel pour émettre un message et si deux voisins qui émettent sur la même fréquence le font à des instants différents, la communication s'effectue correctement).

Dans le contexte de l'auto-stabilisation, la coloration des nœuds adjactents au moyen de couleurs distinctes est caractéristique des problèmes dont la spécification passive est vérifiable localement (c'est-à-dire, en regardant seulement les voisins). Nous reviendrons dans la section ≪ *Aide à la conception d'algorithmes auto-stabilisants* ≫ page 39 sur mes contributions en rapport avec la localité.

Chapitre 3

Comprendre le monde

Mieux vaut comprendre peu que comprendre mal.

Anatole France

Dans ce chapitre nous allons, en trois parties, « comprendre le monde » que j'explore avec mes recherches. Je délaisse ici la vulgarisation anthropomorphique pour adopter un niveau de langage plus scientifique mais pas encore technique (la technique étant réservée aux résultats déjà publiés dans la partie agrafage du document).

Nous allons d'abord aborder (section 3.1) le concept d'auto-stabilisation en le déconnectant du cadre réparti dans lequel il a été défini et où il présente un réel intérêt. En effet, en présentant l'auto-stabilisation dans un cadre centralisé, le lecteur en saisira le principe et comprendra quels sont les points importants à prouver pour montrer qu'un algorithme est auto-stabilisant ou pour le rendre auto-stabilisant.

Une fois le concept d'auto-stabilisation mis en place, je décris (section 3.2) l'algorithmique répartie en général et j'explicite pourquoi l'auto-stabilisation est à mes yeux la propriété primordiale dans ce domaine.

Enfin, dans la section 3.3 page 34, je présente mes apports au domaine qui me paraissent les plus significatifs.

3.1 Comprendre l'auto-stabilisation

Dans cette section, je veux rester, au début, très concrète et je présente un exemple de compteur de nombres premiers sur 4 bits et un exemple de compteur de nombres pairs. Ces deux exemples vont me permettre d'établir ce qu'est le concept d'auto-stabilisation 3.1.2 puis d'illustrer les différentes idées pour faire acquérir la propriété d'auto-stabilisation dans la section 3.1.3.

Avant toute chose définissons l'auto-stabilisation. Rappelons qu'une configuration constitue un état global du système et qu'une exécution est une séquence maximale de configurations telle que le passage d'une configuration à la suivante se fait par l'exécution de l'algorithme par une ou plusieurs machines. La spécification est une propriété sur les configurations (spécification silencieuse) ou sur les exécutions (spécification dynamique).

Définition de l'auto-stabilisation. Un algorithme (ou un composant) est auto-stabilisant pour une spécification s'il est possible de définir un sous-ensemble des ses configurations (appelé ensemble de ses configurations légitimes) qui sont telles que :

1. (**correction**) partant d'une configuration légitime, l'algorithme (ou le composant) vérifie sa spécification ;
2. (**convergence**) partant de n'importe quelle configuration, l'algorithme (ou le composant) en fonctionnement normal (c'est-à-dire en l'absence de nouvelles pannes qui lui feraient atteindre n'importe quelle configuration) converge vers une configuration légitime.

3.1.1 Lecture des figures

Toutes les réflexions de cette section 3.1 sont illustrées par des figures qui ont toutes volontairement le même aspect visuel (elles se ressemblent mais sont toutes différentes).

La face de clown de la figure 3.1 est la légende de ce code graphique. Ses yeux sont les symboles pour les configurations légitimes (son œil gauche étant une configuration initiale), son nez est le code pour représenter le profil des configurations illégitimes et sa bouche est la convention visuelle pour matérialiser l'action de passer d'une configuration à l'autre. Ses oreilles, ses cheveux et le contour de son visage ne cohabitent avec le reste que pour des raisons esthétiques mais n'ont aucune signification. Ainsi les configurations légitimes seront toujours représentées par un carré dont le bord est épaissi. Les configurations illégitimes seront elles symbolisées par un carré entouré d'un gribouillage qui met en évidence leur caractère incorrect (et le gribouillage seul s'il s'agit d'un profil de comportement). À l'intérieur des carrés qui évoquent les configurations, on note des valeurs qui permettent de les distinguer les unes des autres (en conséquence, le gribouillage est vide s'il s'agit d'un profil de comportement et contient un carré s'il s'agit d'une configuration illégitime).

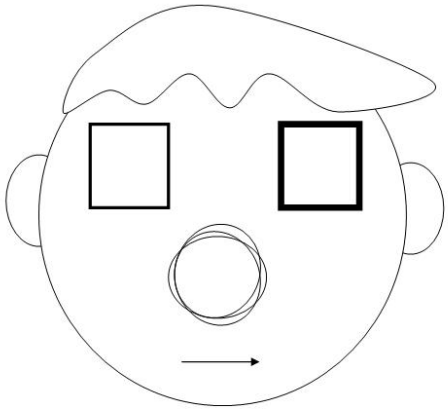


FIGURE 3.1 – Légende pour la représentation des spécifications ou des exécutions autour de l'auto-stabilisation.

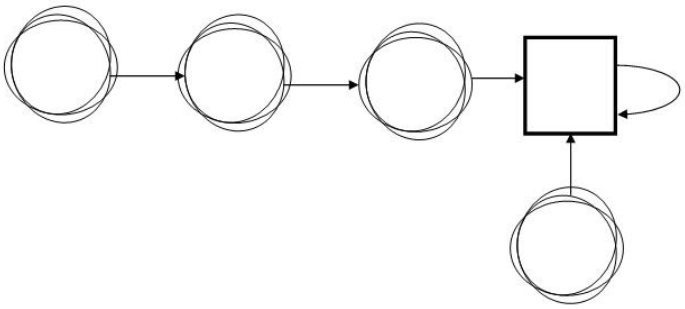
Dans la section 3.1.2 page ci-contre, les configurations légitimes sont les nombres premiers, dans la section 3.1.3 page 22 ce sont les nombres pairs. Respectivement, dans les carrés représentant les configurations illégitimes de la section 3.1.2 sont inscrits des nombres non premiers sur 4 bits et dans ceux des configurations illégitimes de la section 3.1.3 page 22 sont les nombres impairs sur 3 bits.

La possibilité d'un changement de configuration en fonctionnement normal est représentée par une flèche.

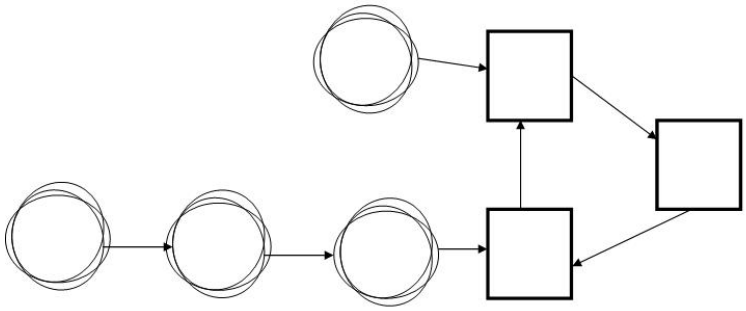
L'enjeu de l'auto-stabilisation est que, partant d'une configuration légitime (carré épaissi), les flèches ne permettent d'atteindre que des configurations légitimes (carrés épaissis), tandis que partant de n'importe quelle configuration légitime ou illégitime, toute succession de flèches mène nécessairement à une configuration légitime (carré épaissi).

La figure 3.2 présente les profils des composantes connexes des fonctionnements auto-stabilisants selon qu'ils ont une spécification statique ou dynamique [1].

1. Ces profils sont conformes aux dessins d'algorithmes concrets disponibles dans l'annexe de ma thèse [Del95] mais sous une autre légende que celle du clown.



(a) cas d'une spécification statique.



(b) cas d'une spécification dynamique.

FIGURE 3.2 – Profils de fonctionnements auto-stabilisants.

Les figures 3.2a et 3.2b ne sont que des profils et les chaînes ou les cycles de longueur 3 peuvent être plus courts ou plus longs et le nombre de chaînes pointant vers le cycle peut être augmenté. Par contre ce qu'il faut retenir c'est que, pour une spécification statique, tout cycle de configuration légitime (exprimé sur les variables de sortie de l'algorithme) est de longueur 1 alors que, pour une spécification dynamique, les cycles peuvent être de longueur plus grande que 1. Dans tous les cas il ne peut y avoir de cycle à l'intérieur des configurations illégitimes [2] [3].

Rappelons, avant de quitter cette section, que ce qui vient d'être dit ici reste vrai si on quitte le cadre centralisé [4]. Une configuration devient alors l'état combiné de plusieurs composants comme nous le verrons dans la section 3.2 page 30.

3.1.2 Nombres premiers : compteur auto-stabilisant ou pas

Je vais introduire ici l'auto-stabilisation avec un exemple concret et totalement original, mais accessible à n'importe quel étudiant de première année d'Informatique. Supposons qu'un architecte veuille construire un composant qui énumère les nombres premiers [5] sur 4 bits et qu'il dispose à cet effet de 4 bascules JK, d'une horloge et d'un nombre limité de portes logiques.

2. Rappelons ici que les profils étudiés le sont sous le démon supporté par l'algorithme et que les configurations représentées ne le sont que sur les variables de sortie, sinon l'algorithme n'est pas auto-stabilisant.

3. On exclut ici les exécutions probabilistes où des cycles sont possibles mais qui seront rompus par l'aléatoire.

4. Dans le cas où une hypothèse d'équité est nécessaire au bon fonctionnement de l'algorithme réparti, des cycles de configurations illégitimes peuvent apparaître sur la représentation statique du système donné précédemment mais ils sont brisés dans toute exécution [Tix09].

5. Un nombre est premier s'il admet exactement deux diviseurs 1 et lui-même. En conséquence, 1 n'est pas premier.

Le composant est spécifié pour énumérer de manière infinie tous les nombres premiers sur 4 bits en passant d'un nombre premier à celui qui lui est immédiatement supérieur à chaque pas de fonctionnement et en retournant au premier des premiers (le nombre 2) dès qu'il a atteint le plus grand des premiers sur 4 bits, c'est-à-dire 13 ou D si on parle en hexadécimal. La figure 3.3 donne une spécification visuelle des configurations par lesquelles le composant doit passer.

Selon que la spécification du compteur est passive (voir définition 2.1.3 page 6, le compteur produit indéfiniment en changeant à chaque top d'horloge la suite des nombres premiers sur 4 bits et l'utilisateur consulte lorsqu'il le souhaite la sortie du compteur) ou active (l'utilisateur active l'horloge pour modifier la sortie en obtenant le nombre premier immédiatement supérieur à la sortie courante, ou 2 si la sortie courante était D), les flèches représentent chacune soit un top d'horloge, soit une action de l'utilisateur.

Devant cette spécification, notre architecte décide alors de faire un compteur synchrone dont chacune des sorties Q_i détermine le bit de poids 2^i. Après une étude et en fonction des composants physiques dont il dispose il choisit de construire son circuit grâce au schéma de la figure 3.6 page 20. Les quatre bascules sont synchronisées sur une horloge ou un composant actionnable par l'utilisateur du circuit (selon que la spécification est passive ou active, voir 2.1.3 page 6). Ce circuit répond bien à la spécification mais il peut atteindre par l'effet d'une faute n'importe quel nombre sur 4 bits, dont potentiellement un nombre qui n'est pas premier.

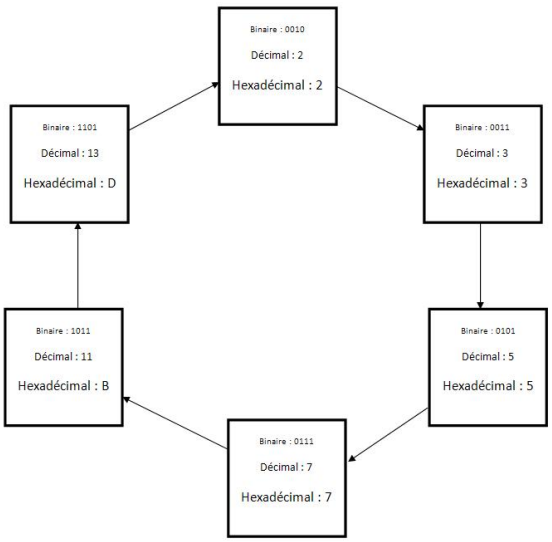


FIGURE 3.3 – Spécification visuelle du ≪ compteur de nombres premiers sur 4 bits ≫.

La figure 3.4 page suivante montre, conformément au code graphique que nous avons présenté dans la section 3.1.1 page 16, la suite des nombres qui sont générés par ce composant en fonction des valeurs (c'est-à-dire configurations) initiales de celui-ci. Ce composant, s'il est forcé à une valeur autre que 0, 4, 8, C ou F, énumère bien les nombres premiers 2, 3, 5, 7, B, D puis retourne à 2 et ce indéfiniment ; par contre s'il est forcé à 0, 4 ou 8, il reste bloqué sur cette valeur ; s'il est forcé à F, il passe par une valeur intermédiaire avant de vérifier sa spécification. **La figure 3.4 illustre clairement que le composant de la figure 3.6 page 20 n'est pas auto-stabilisant.**

Pour un coût similaire l'architecte aurait pu choisir le composant de la figure 3.7 page 21 qui, lui, est auto-stabilisant puisque quelle que soit la valeur de forçage, il est toujours actif et finit nécessairement par énumérer les nombres premiers dans l'ordre voulu, comme précisé par la figure 3.5.

La valeur de forçage, ou valeur initiale, est une valeur qui peut être produite après une panne puisqu'elle correspond à une valeur existante sur 4 bits. Le composant auto-stabilisant garantit que quelle que soit la valeur initiale du composant

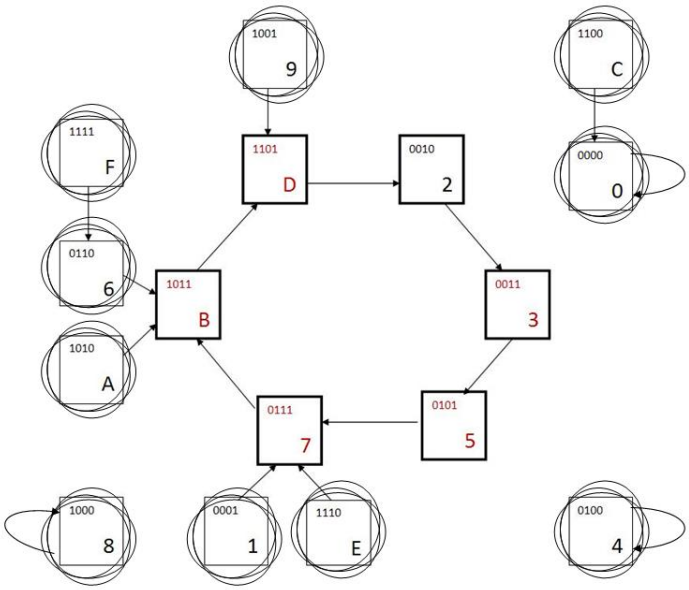


FIGURE 3.4 – Fonctionnement du circuit non auto-stabilisant pour énumérer les nombres premiers sur 4 bits.

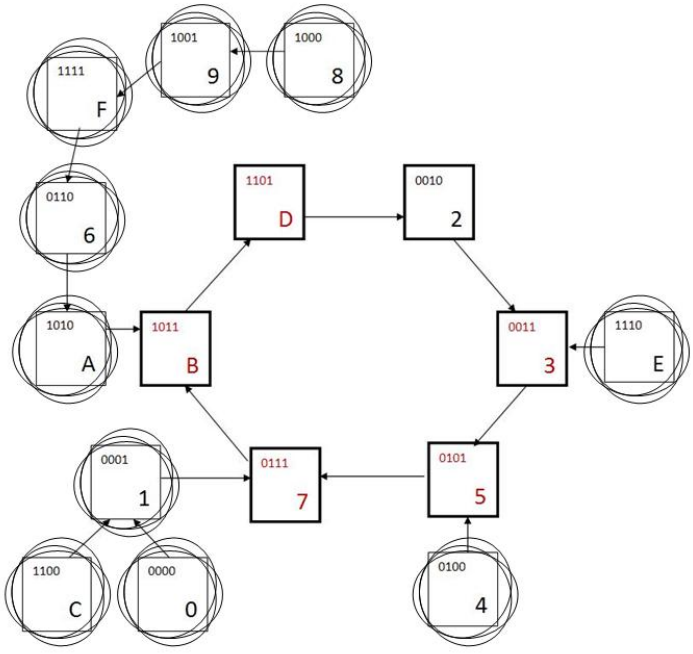


FIGURE 3.5 – Fonctionnement du circuit auto-stabilisant pour énumérer les nombres premiers sur 4 bits.

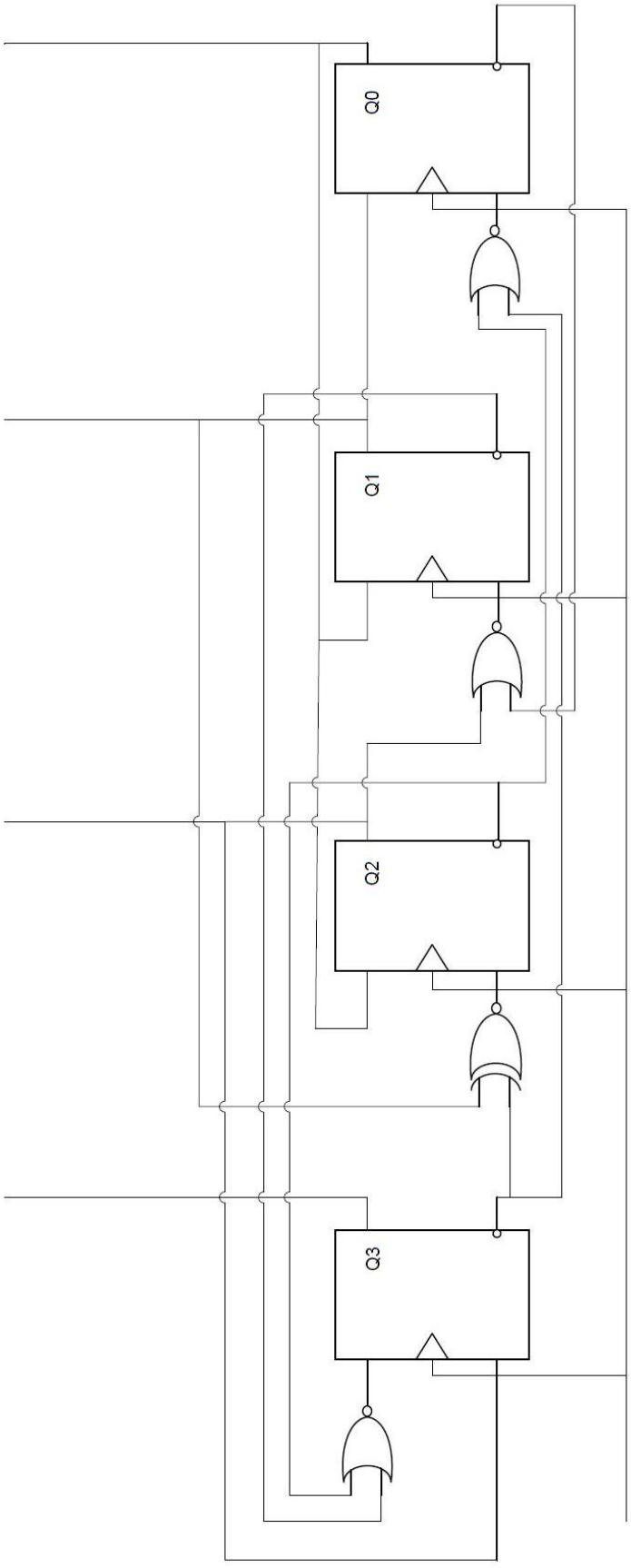


FIGURE 3.6 – Circuit d'un compteur non auto-stabilisant de nombres premiers sur 4 bits ($Q_3Q_2Q_1Q_0$).

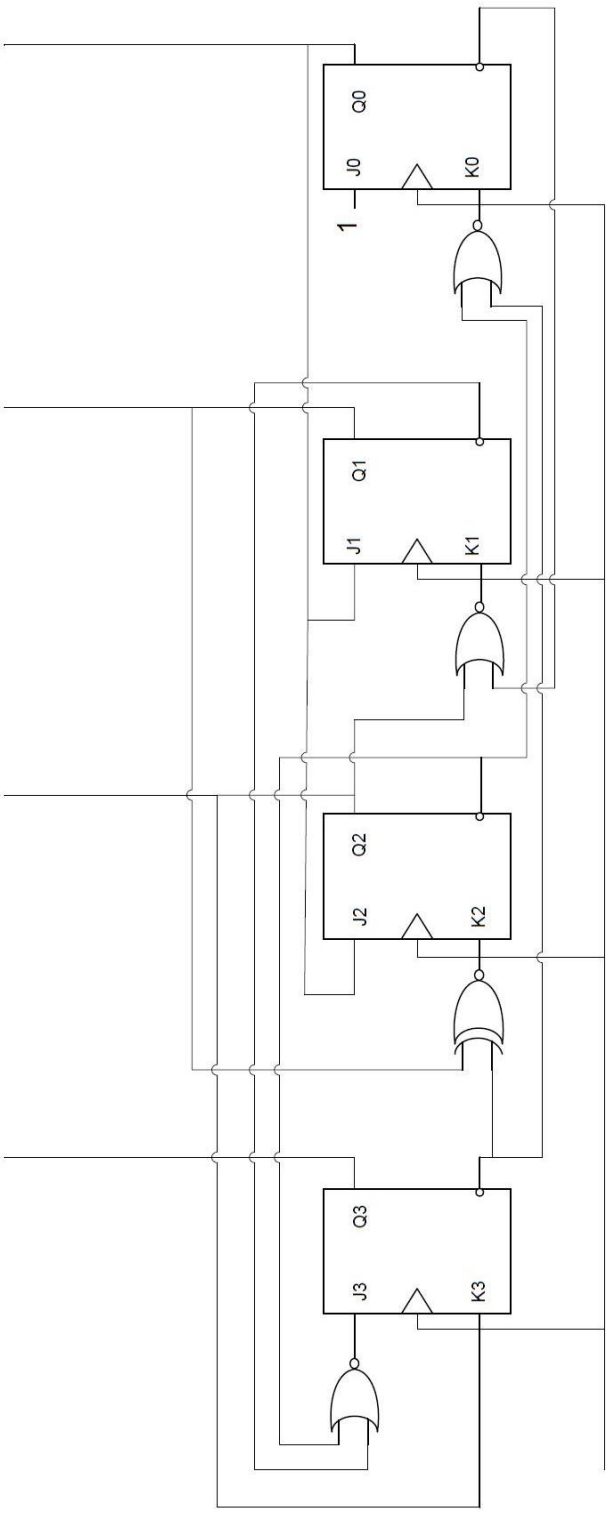


FIGURE 3.7 – Circuit auto-stabilisant pour énumérer les nombres premiers sur 4 bits.

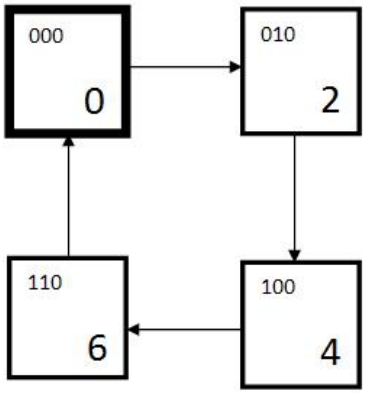


FIGURE 3.8 – Fonctionnement normal du composant « Compteur de nombres pairs » sur 3 bits.

et après un temps de stabilisation le composant va vérifier sa spécification, comme l'illustre la figure 3.5 page 19. Notons que le temps de stabilisation (longueur de la plus longue chaîne de configurations illégitimes) est de 5, il est atteint lorsque le composant est forcé à la valeur 8.

Dans cette section, nous avons expliqué le concept d'auto-stabilisation qui est une propriété de convergence vers des configurations à partir desquelles (en l'absence de nouvelles fautes) tout se passe correctement. Nous avons donné une caractérisation visuelle pour la mettre en valeur dans le cas où on est en mesure de visualiser toutes les configurations et les interactions entre les configurations. Nous avons évoqué la problématique du surcoût supposé de l'auto-stabilisation sur un exemple où il est justement inexistant.

3.1.3 Trois pratiques pour acquérir l'auto-stabilisation

Dans les paragraphes qui suivent, nous abordons comment concevoir des modules auto-stabilisants. Pour cela, je propose un jouet : le compteur de nombres pairs. J'en présente une version non auto-stabilisante et quatre manières de le rendre auto-stabilisant. Trois de ces manières seront reconnues à la fin de la section comme des pratiques transposables en algorithmique répartie (d'où le titre de la section courante), la quatrième nous permettra d'introduire les concepts de stabilisation instantanée (snap stabilization) et de stabilisation idéale (ideal stabilization).

Présentation du jouet. Supposons, d'une part qu'on ait besoin d'un composant spécialisé qui énumère indéfiniment tous les entiers pairs qui puissent être codés sur n bits, et d'autre part que pour concevoir ce composant on dispose de n positions à même de mémoriser chacune un bit (0 ou 1). L'implantation du composant pourrait être d'utiliser toutes les positions, de supposer qu'elles sont initialisées à 0, et de représenter avec la $i^{\text{ème}}$ position le bit de poids 2^i. Partant de l'entier i après une impulsion (top d'horloge ou action d'un utilisateur), notre composant doit produire l'entier $i + 2$, jusqu'à atteindre l'entier $2^n - 2$ (le plus grand entier pair sur n bits) dont le successeur serait 0.

En fonctionnement normal, un tel compteur fonctionne indéfiniment, retournant à zéro après avoir produit le plus grand entier pair possible sur n bits (voir figure 3.8). Pour garantir que seuls des entiers pairs sont produits, un tel composant

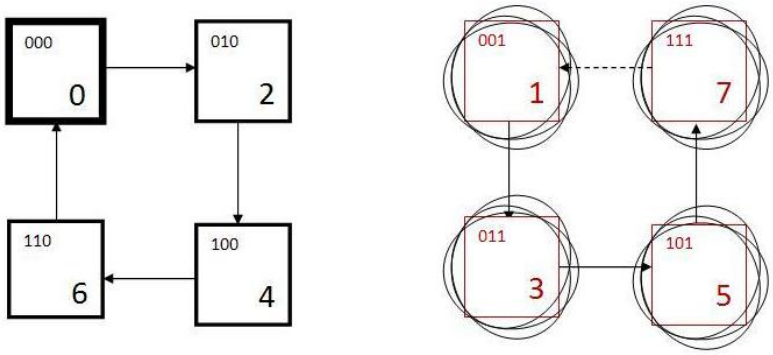


FIGURE 3.9 – « Compteur de nombres pairs » sur 3 bits avec la possibilité d'une corruption de mémoire.

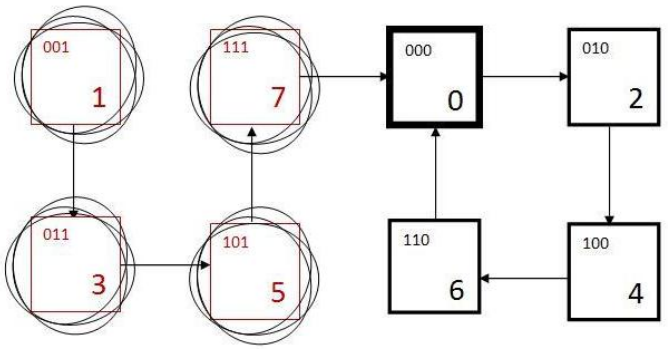


FIGURE 3.10 – « Compteur de nombres pairs » sur 3 bits auto-stabilisant - version débordement.

est amené à faire potentiellement varier tous les bits de poids fort ($2^i, i \in [1, n-1]$) mais jamais le bit de poids faible (2^0) qui reste constamment à zéro.

Si une panne transitoire inverse un bit quelconque (sauf le bit de poids faible), le compteur passe d'un entier pair à un autre non immédiatement supérieur mais continue ensuite à se comporter correctement, il continue donc à respecter la spécification qui est d'énumérer indéfiniment tous les entiers pairs possibles. Par contre, dès qu'une panne transitoire inverse le bit de poids faible, le composant cesse de se comporter correctement puisque, n'ayant aucune raison d'inverser le bit de poids faible en fonctionnement normal, il n'énumérera alors que des entiers impairs. La figure 3.9 permet de visualiser ce problème. Si une faute faisait qu'une des configurations illégitimes était atteinte aucun comportement correct n'existerait plus. Pour qu'un tel compteur soit auto-stabilisant il faudrait qu'il soit construit de manière à ce que toute production accidentelle d'un entier impair induise, après un nombre fini d'étapes, la production d'un entier pair. Plusieurs solutions sont envisageables pour implanter un tel compteur auto-stabilisant. J'en présente ici quatre que j'estime significatives.

Utiliser le contrôle du débordement fourni par l'addition. Une première approche est celle de la programmation. Notre composant effectue en permanence l'opération $i+2$ sur n bits. Cette opération va fatalement produire un débordement et, en cas de débordement, on peut imposer un retour à la valeur initiale zéro. Ainsi le fonctionnement du nouveau composant est celui de la figure 3.10 où la flèche en pointillé dans la figure 3.9 est redirigée vers une configuration légitime.

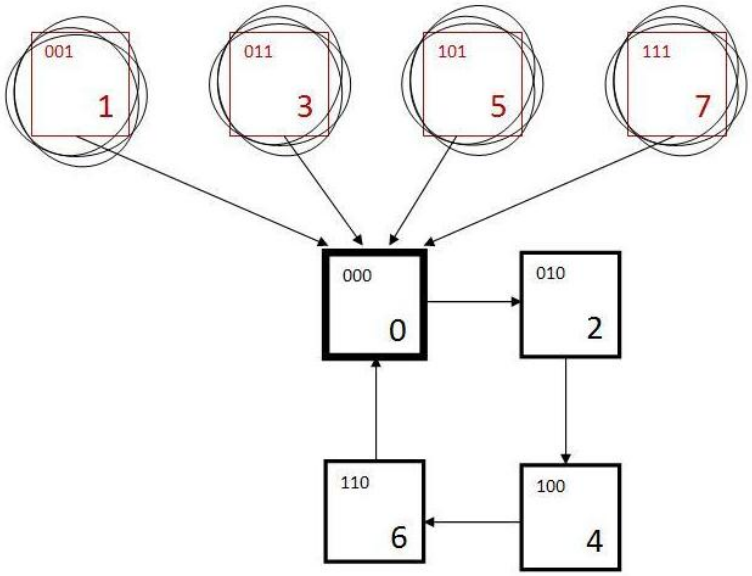


FIGURE 3.11 – Implantation du composant ≪ Compteur de nombres pairs ≫ sur 3 bits - version parité.

Il est alors auto-stabilisant puisque, de toute configuration on atteint une configuration légitime et que d'une configuration légitime on obtient un comportement correct (l'énumération des entiers pairs). Par contre son temps de stabilisation est grand puisque, si l'entier impair 1 est produit, tous les nombres impairs seront parcourus avant d'atteindre le plus grand entier $2^n - 1$ et par la suite une phase stabilisée.

Utiliser le contrôle de la parité du nombre. La deuxième méthode est de construire le compteur de manière à ce que la production de n'importe quel entier impair induise que son successeur soit zéro (voir figure 3.11). Dans ce cas le temps de stabilisation [6] est de 1. La détection d'une mauvaise configuration est ≪ globale ≫ : elle concerne le nombre tout entier [7]. De même, la correction est ≪ globale ≫, elle concerne tous les bits du nombre.

Utiliser le contrôle du bit de poids faible. La troisième façon (figure 3.12 page suivante) serait de construire le composant de manière à ce que le bit de poids faible soit régulièrement rafraîchi (et donc remis à zéro s'il est mal positionné entre deux raffraîchissements). Dans ce cas, le temps de stabilisation est de 1 et on ne contrôle la valeur que d'un bit pour détecter un problème. La détection et la correction sont alors ≪ locales ≫ (elles concernent uniquement un bit facilement identifiable) puisque tout nombre impair a un nombre pair à distance 1 dans l'espace des configurations.

Utiliser moins de positions. La quatrième solution illustrée par la figure 3.13 page 26 est de constater que le nombre de nombres pairs sur n bits est de 2^{n-1}. En conséquence, $n - 1$ positions suffisent à l'implantation d'un tel compteur et une telle implantation n'admet aucune configuration illégitime. Cette remarque n'est pas une simple optimisation mais un concept plus profond qui se décline en

6. il est en fait égal au temps nécessaire pour calculer la parité du nombre.

7. Cette notion de globalit n'est qu'une vue de l'esprit, le contrle de parit fait ici intervenir le nombre cod mais pas directement le bit de poids faible qui indique la parit

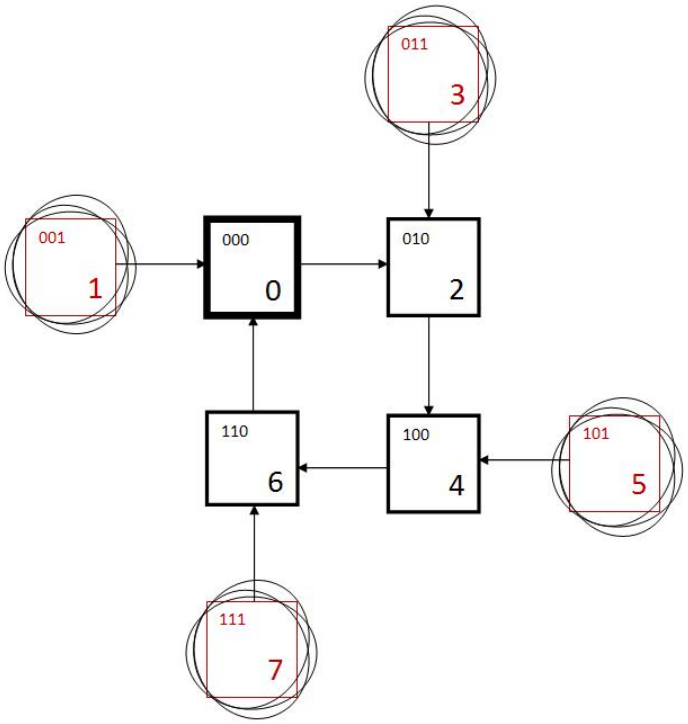


FIGURE 3.12 – Implantation du composant « Compteur de nombres pairs » sur 3 bits- version bit de poids faible.

algorithmique répartie sous deux noms : la stabilisation instantanée [BDPV99, BDV05, CDV09b, DDNT10] ou la stabilisation idéale [NT11] selon que la spécification est active ou passive.

Discussion sur les pratiques

Dans la première solution, on utilise le mécanisme de débordement de l'addition pour détecter l'erreur et la corriger, on fait donc appel à un détecteur de fautes extérieur à l'algorithme de production des nombres pairs. Cette méthode a été mise en exergue pour symboliser celle qui est déclinée dans l'article « transient fault detectors » [BDDT98, BDDT07] consultable en annexe A.1 page 51. Un détecteur de défaillances transitoires pour une spécification n'utilise que les variables de sortie de l'algorithme pour détecter une inconsistance et permettre une correction ultérieure (par exemple, par une réinitialisation vers une configuration légitime, comme c'est le cas dans l'exemple).

La deuxième solution illustre la pratique où la détection de l'erreur par l'algorithme est plus ou moins locale, mais où sa correction consiste à faire une réinitialisation. Dans l'exemple, tous les bits du nombre sont concernés et remis à zéro. L'auto-stabilisation par réinitialisation en cas de détection d'erreur est l'un des mécanismes les plus anciens pour garantir l'auto-stabilisation dans un cadre réparti [APSVD94], par exemple via des transformations algorithmiques [KP93].

La troisième solution peut être considérée comme une version purement locale de la précédente (la détection et la correction restent locales, un seul bit est concerné). Dans un contexte réparti, une meilleure (c'est-à-dire plus faible) localité dans la détection et la correction des fautes transitoires est souvent synonyme de meilleures performances [APSV91, KP99, AD02] du fait d'un coût plus faible en communications.

La quatrième solution me semble être la plus élégante pour implanter un

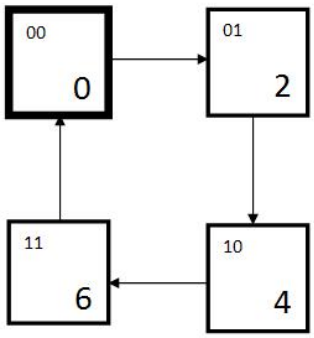


FIGURE 3.13 – Implantation du composant ≪ Compteur de nombres pairs ≫ sur 3bits avec seulement 2 bits.

tel composant, mais elle requiert une réflexion sur la signification des configurations présentes dans la spécification (les nombres pairs inférieurs à n) plutôt que sur les opérations qui intuitivement devraient être implantées (ici l'opération $+2 \bmod 2^n$). Elle sert ici d'introduction à la section 3.1.4.

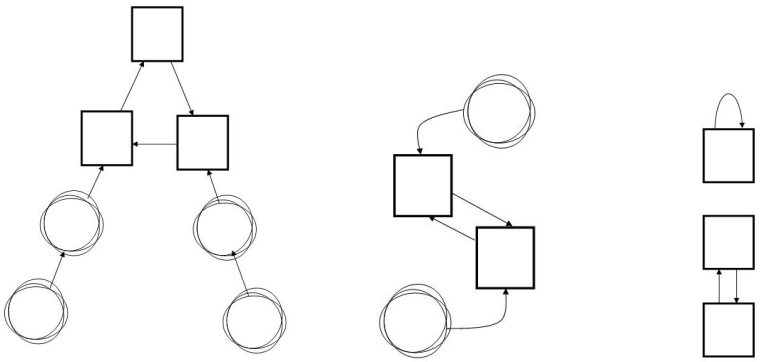
Vue sous cet angle, l'auto-stabilisation est donc quelque chose de très simple avec de nombreuses solutions envisageables. Toutefois, même si pour fabriquer un vrai système auto-stabilisant il faut que tous les acteurs le soient et en particulier le plus petit composant centralisé, il est important de prendre conscience que l'auto-stabilisation a un réel intérêt dans le contexte de l'algorithmique répartie ou du calcul distribué. La compréhension du monde réparti est justement l'objet de la section 3.2 page 30.

3.1.4 Comparaison avec d'autres modes de stabilisation

D'autres modes de stabilisations sont apparus dans la littérature, et peuvent être plus contraignants [DPV11b] (c'est-à-dire que la condition à satisfaire est plus restrictive que celle de l'auto-stabilisation) ou moins contraignants [DPV11a] (c'est-à-dire que l'on autorise alors des comportements que l'on ne tolèrerait pas d'un algorithme auto-stabilisant). Les approches plus limitatives ont été développées pour pallier aux critiques habituellement faites aux algorithmes auto-stabilisants (en particulier, l'absence de garanties sur la sûreté ou sur les performances dans les cas usuels). Les approches moins restrictives ont été motivées dans la plupart des cas par des résultats d'impossibilité dans le cadre général. Sans être exhaustive, j'évoque dans le reste de la section la pseudo-stabilisation, la stabilisation idéale, et la stabilisation instantanée, qui seront utiles par la suite.

La **pseudo-stabilisation** [BGM93] ne peut être définie sur les configurations puisqu'elle ôte au concept d'auto-stabilisation, tel qu'il est défini initialement, la notion de configuration légitime. Un algorithme est pseudo stabilisant si toutes ses exécutions admettent un suffixe qui vérifie sa spécification. Le concept de pseudo-stabilisation reprend l'esprit de définition de l'auto-stabilisation (ultimement, le comportement est celui attendu par la spécification) en faisant disparaître la notion même de configuration (qui permet l'écriture de preuves par invariants). D'un point de vue pratique, la pseudo-stabilisation correspond à l'intuition (erronée) que de nombreux praticiens ont de l'auto-stabilisation, mais a également permis de montrer que plusieurs protocoles réseaux (dont par exemple celui du bit alterné) sont, dans une certaine mesure, stabilisants.

La **stabilisation idéale** (ideal stabilization) [NT11] définit les algorithmes pour lesquels toutes les configurations sont légitimes.



(a) Autostabilisation. (b) Snap stabilisation. (c) Idéale stabilisation.

FIGURE 3.14 – Profils des concepts.

La **stabilisation instantanée** (snapstabilisation) [BCV03, CDPV03, CDV05b, CDV05a, BDV05, CDV06, CDPV06, BDPV07, CDPT07, CDV09b, CDV09a, CDL+10, DDNT10] garantit que le système vérifie la spécification dès la première requête après la fin des fautes, elle aurait donc un temps de stabilisation de zéro. Dans les discussions informelles entre chercheurs dont j'ai été témoin, la stabilisation instantanée a longtemps été confondue avec la notion de stabilisation idéale. Certains faisant le raccourci que si le temps de stabilisation est nul c'est qu'il ne peut exister de configuration illégitime. Pour une spécification passive la stabilisation instantanée n'est effectivement pas distingable de la stabilisation idéale, par contre pour une spécification active, elle est tout à fait distingable. En effet, le principe de la stabilisation instantanée est de se préparer, après une faute, à répondre instantanément correctement aux requêtes qui surviennent. Si toutes les configurations sont légitimes, la préparation est déjà faite. Par contre s'il existe des configurations illégitimes, il faut garantir que la première requête de l'utilisateur les fera quitter.

Sur les figures 3.14 sont repris les profils des exécutions pour la stabilisation idéale, l'auto-stabilisation et la stabilisation instantanée. Le profil retenu pour la stabilisation idéale fait volontairement apparaître deux composantes connexes dans les cycles de configurations légitimes (ce qui n'est pas toujours le cas en réalité). Ce choix a été fait, d'une part, pour rappeler que l'ensemble des configurations légitimes n'est pas nécessairement connexe et, d'autre part, pour mettre un point sur le i. Cette remarque reste vraie pour l'auto-stabilisation et pour la stabilisation instantanée même si leur profil ne le montre pas. De même, les cycles dans les trois cas peuvent être plus courts ou plus longs selon l'algorithme mais doivent rester de longueur 1 si la spécification est statique. Enfin les chaînes de configurations illégitimes de l'auto-stabilisation peuvent être allongées mais pas celle de la stabilisation instantanée. En effet, pour la stabilisation instantanée les chaînes de configurations légitimes vues sur les variables de sorties sont nécessairement de longueur 0 ou 1 pour une spécification active et de longueur 0 pour une spécification passive [8].

Si on reprend le problème de la propagation d'information avec retour (qui à été largement étudiée dans le domaine de la stabilisation instantanée et la parabole de la viande de cheval), l'information à diffuser peut être : « Alerte, arrêtez de manger du bœuf précuisiné car c'est du cheval ! ». Si l'algorithme est

8. Attention cette remarque est sujette à discussion en algorithmique répartie car on ne considère ici que des configurations globales sur les variables de sortie de l'agorithme. En algorithmique répartie, les chaînes de longueur 1 ne seront alors que celles qui modifient la variable de sortie concernée par la requête active de l'algorithme instantanément stabilisant.

auto-stabilisant, même si avant toute conception de repas les processus exécutent un pas de l'algorithme de diffusion, les convives de l'auberge vont continuer de manger du cheval pendant un certain temps avant que la stabilisation ne soit achevée car ils ne seront informés de l'alerte qu'après le temps de stabilisation de l'algorithme de diffusion. Si l'algorithme est instantanément stabilisant, l'information transmise arrive à destination dès la première invocation de l'algorithme de diffusion et aucun plat à base de cheval n'est alors conçu après l'alerte.

3.1.5 Preuve d'auto-stabilisation

Pour prouver qu'un algorithme est auto-stabilisant, plusieurs méthodes cohabitent dans la littérature.

La plus répandue, et pourtant la moins efficace, est de faire une preuve *ad hoc* de correction puis de convergence [DT01, BJM06]. Dans ce cas on considère que la nouvelle spécification de l'algorithme est sa spécification auto-stabilisante et on prouve cette spécification avec les méthodes de preuves classiques en algorithmique répartie.

Une autre méthode de preuve [Tel94, DT98, TG99, DT02, DDT06] est de donner une mesure (un poids) aux configurations et de faire le travail de preuve sur la mesure elle même (par exemple, la mesure de toute configuration légitime est de zéro, et toute exécution partielle de l'algorithme fait décroître la mesure). Cette technique est souvent appelée dans la litérature la technique des *fonctions de potentiel*. Cette technique permet des preuves élégantes mais reporte la difficulté de la preuve sur le choix d'une mesure pertinente. Des exemples d'application de cette technique sont disponibles dans certains les articles du chapitre « *Agrafage* » page 51, des sections A.3 et A.4.

La dernière possibilité est de ne pas avoir de preuve à faire, en concevant un nouvel algorithme à partir de briques déjà existantes (c'est-à-dire d'algorithmes auto-stabilisants pour des sous-problèmes du problème principal) que l'on compose par des méthodes qui préservent la propriété d'auto-stabilisation [Var97, DH99, BGJ01, YKO+06, DH07, BPBRT10]. Le stade ultime de cette approche est de définir des propriétés sur les fonctions exécutées par un processus qui garantissent que l'ensemble du sytème est auto-stabilisant, quel que soit le graphe de communications ou l'ordonnancement du système. Les r-opérateurs [DT01, DDT06] décrits section « *Aide à la conception d'algorithmes auto-stabilisants* » page 39 permettent une telle approche.

3.1.6 Étude des performances

L'étude des performances d'un algorithme auto-stabilisant peut se faire selon plusieurs critères :

- Le temps de stabilisation (c'est-à-dire la longueur de la plus longue chaîne de configurations illégitimes) ;
- L'espace mémoire ;
- La quantité de messages échangés et la quantité d'informations dans chaque message ;
- Le démon minimal sous lequel il est auto-stabilisant et plus généralement les hypothèses minimales qui permettent l'auto-stabilisation ;
- Son surcoût par rapport à un algorithme classique (c'est-à-dire, un algorithme qui n'est pas auto-stabilisant).

Le dernier critère, celui du surcoût peut sembler intéressant en soi, mais il est à contrebalancer avec l'avantage qu'on gagne à utiliser un algorithme auto-stabilisant plutôt qu'un algorithme qui ne l'est pas. En effet, si un algorithme n'est pas auto-stabilisant et que des pannes transitoires surviennent, il ne fonctionne

tout simplement plus (c'est à dire qu'il ne vérifie plus *jamais* sa spécification). A contrario, l'algorithme auto-stabilisant va continuer de fonctionner et son comportement sera correct dès que les pannes auront cessé et que le temps de stabilisation se sera écoulé. En présence de pannes transitoires, le sucoût d'un algorithme auto-stabilisant est donc nul. En comparaison, le surcoût d'un algorithme classique par rapport à un algorithme auto-stabilisant en présence de pannes transitoires est infini.

Dans la pratique, si un système ne fonctionne pas du fait de l'occurrence de pannes, et sous réserve que ce mauvais fonctionnement soit observable de l'extérieur du système, il est alors relancé par l'opérateur (humain) qui l'utilisait. Cela fait du système « algorithme plus opérateur » un système auto-stabilisant par reset, où on va tolérer un certain nombre de dysfonctionnements qui devront être mis en évidence par l'opérateur et résolus par une remise à zéro de l'algorithme jusqu'à atteindre une exécution correcte de l'algorithme, le tout sans avoir pris consience du fait que ce type de pratique impose régulièrement (après chaque salve de pannes) des périodes où le système ne vérifie pas sa spécification (cette non-vérification est même nécessaire à la détection du mauvais comportement). Le gain énorme de l'auto-stabilisation est, d'une part de ne pas avoir besoin d'opérateur extérieur pour relancer le système, et, d'autre part d'avoir la conscience du fait qu'il est possible que la spécification initiale soit non respectée pendant quelques instants (cette conscience est d'ailleurs explicitée lorsqu'on écrit une preuve d'auto-stabilisation, puisque la spécification qui est effectivement prouvée est celle qui autorise le système à afficher un comportement temporairement incorrect après la survenue de fautes transitoires).

3.1.7 Réflexion sur l'infini

Un ordinateur est un objet fini qui contient un espace mémoire fini. Il se sert de cette mémoire pour représenter (modéliser) l'information d'un monde (souvent mathématique) qui, d'un point de vue pratique, est intrinsèquement infini. Il y a donc dans cette modélisation une partie de l'information que l'on cherche à représenter perdue par le modèle.

Par exemple, il arrive très souvent dans la pratique qu'on ait besoin d'un compteur infini pour faire fonctionner un système. La pratique en algorithmique répartie classique est de dire : « prenons un compteur de 64 bits et initialisons le à zéro, le temps qu'il énumère 2^{64} valeurs, suffisamment de temps se sera écoulé pour que l'ordinateur qui effectue le calcul soit obsolète depuis de nombreuses générations et ait été remplacé par un autre dont le compteur a été initialisé à zéro ; ces 2^{64} valeurs sont donc assez nombreuses pour représenter l'infini en pratique ». Or, cette intuition est fausse dès que des pannes transitoires peuvent frapper le système. En effet, si une panne transitoire force la valeur d'un tel compteur à une valeur très grande, $2^{64} - 1$ par exemple, l'infini est atteint et plus rien ne peut être garanti (en particulier, pas la propriété que les nombres grandissent en permanence quand on les incrémente de 1). L'auto-stabilisation pousse à être conscient de ce problème lors de la conception du système et permet de garantir que, même avec une mémoire finie incorrectement initialisée, la perturbation sera passagère et que tout rentrera ensuite dans l'ordre. Ainsi les « bons algorithmes » auto-stabilisants (du point de vue de leur implantation en pratique sur un ordinateur) sont ceux qui n'utilisent que des variables finies.

3.2 Comprendre l'algorithmique répartie

De nombreux systèmes informatiques sont aujourd'hui organisés selon un schéma réparti : ils sont constitués d'unités indépendantes (ordinateurs, processeurs, processus, capteurs, téléphones portables etc.) se coordonnant au moyen de communications pour collaborer à une tâche.

Les principales motivations ayant mené à la conception de tels systèmes sont les suivantes [Tel01] :

- **Échange d'informations** : l'émergence du besoin d'échanger des informations a eu lieu dans les années soixante où les principales universités ont commencé à disposer d'ordinateurs centraux. Les coopérations inter-universitaires contribuèrent à l'élaboration des réseaux longue distance. Aujourd'hui, ces échanges s'effectuent principalement entre ordinateurs individuels interconnectés. Ils vont en s'accroissant avec la démocratisation du réseau *Internet* et l'avènement des téléphones portables et autres supports interconnectés.
- **Partage des ressources** : les différences de coût et les durées relatives d'utilisation entre les ordinateurs et les périphériques (imprimantes, unités de sauvegarde) ont historiquement conduit à la mise en place d'un système de partage des éléments les plus coûteux. Cette façon de procéder permet de générer de moindres coûts par rapport à des systèmes centralisés et, par ailleurs, elle permet une bonne extensibilité en cas d'avancées technologiques (ajout progressif de composants).
- **Fiabilité accrue par la redondance d'informations** : les systèmes répartis rendent possible le bon fonctionnement général du système, même en cas de défaillance de certains composants.
- **Performance accrue par la parallélisation** : la présence de plusieurs unités de calcul ouvre la possibilité de diminuer le temps de latence dans le cas de gros calculs en divisant ces derniers et en exécutant des portions sur des unités de calcul distinctes. C'est le cas pour les ordinateurs parallèles, mais aussi pour les réseaux physiques lorsqu'on envoie une tâche s'exécuter sur un serveur via une antenne relais afin d'obtenir très vite une information que le mini terminal qu'on possède serait incapable de calculer par lui même ou, enfin, pour le cloud où on accède depuis plusieurs terminaux à toutes ses données mise à jour en temps réel.

La programmation de systèmes répartis est basée sur l'utilisation d'algorithmes corrects, flexibles et efficaces : les algorithmes répartis [Lyn96, Gou98, AW98, Dol00, Pel00, Tel01, Ray13b, Ray13a].

On confond désormais la notion d'algorithme réparti et la notion de protocole sous le terme générique d'algorithme réparti. Cette confusion ne fait pas perdre de généralité puisque la différence entre les deux notions ne dépend que du niveau d'abstraction choisi. En effet, un protocole tel qu'on le décrit habituellement n'est qu'une fonctionnalité disponible à un certain niveau d'abstraction. On parle par exemple de protocole de communication point à point ou de protocole de diffusion. Si on se place au niveau de la conception (abstraite) de ces protocoles, ils ne sont que des algorithmes répartis, par contre au niveau (concret) où on les utilise (comme fonctionnalités à l'intérieur d'un système réparti), on les distingue en les appelant protocoles.

Syntaxiquement, les algorithmes répartis sont des collections de codes locaux incluant des instructions de communication. Lorsqu'on se place au niveau local (celui des codes locaux), on parle d'*états* et d'*événements*, alors que, lorsqu'on se place au niveau global (celui de l'algorithme réparti), on parle de *configurations* et d'*actions*.

Dans la section 3.1 page 15 nous avons détaillé par l'exemple le concept d'auto-stabilisation en ne présentant que des exemples centralisés (les compteurs de nombres premiers et les compteurs de nombre pairs). Si on voulait transposer ces exemples de compteurs en algorithmique répartie, il faudrait considérer, par exemple, que chacun des bits est géré par une unité de calcul autonome et que ces unités de calculs se synchronisent (dans les exemples de la section précédente, une telle synchronisation pourrait être orchestrée par le top d'horloge). Toutefois, l'exemple est restreint car la communication reste centralisée et n'est que dans un sens (toutes les unités de calcul travaillent à l'unisson). L'image présentée dans la section 2.1 page 4 est plus intéressante pour présenter les aspects selon lesquels les algorithmes répartis diffèrent des algorithmes centralisés. Nous listons ces aspects dans les sections 3.2.1 à 3.2.5 page suivante. La section 3.2.6 page 33 les reprend en les particularisant du point de vue de la conception des algorithmes auto-stabilisants.

3.2.1 Absence de connaissance de l'état global et difficulté d'initialisation

Dans le cas des algorithmes centralisés, les choix peuvent s'effectuer sur la base d'une connaissance de l'état global du système. À l'inverse, dans le cas des algorithmes répartis, chaque composant ne connaît que son état local et ne peut habituellement pas avoir accès à la connaissance de l'état global (la *configuration*). En conséquence, le contrôle et les choix ne peuvent généralement pas se faire sur la base d'informations globales [HM90].

Si, toutefois, il est indispensable de connaître aussi bien que possible la configuration, ce n'est qu'à l'aide d'un protocole spécifique, dit d'*instantané* [CL85] [9], que l'information peut être obtenue. Dans ce cas, l'information récoltée peut être obsolète, puisque l'algorithme d'instantané est lui-même fondé sur des échanges d'informations qui prennent du « temps ». Avant qu'une information sur l'état d'un composant (ordinateur, processeur ou processus) ne parvienne jusqu'au composant qui la demande, le premier composant peut avoir une nouvelle fois changé d'état.

L'initialisation (ou la réinitialisation) d'un système réparti peut être vu comme le problème dual de l'obtention d'un instantané. Il s'agit de placer, suite à la requête d'un utilisateur externe ou d'une machine participante, toutes les machines dans un état particulier bien connu, pour construire une configuration initiale. Le mettre en œuvre requiert un effort particulier [AG94].

3.2.2 Absence d'existence d'ordre total

Dans le cas des algorithmes centralisés la relation d'ordre entre les actions est totale, elle est induite par une notion temporelle classique. Dans le cas des algorithmes répartis, la notion de temps n'existe que du point de vue local (entre événements sur un même composant). Une autre notion, celle de la causalité (par exemple l'émission d'un message précède sa réception), induit également une relation d'ordre entre actions. Ainsi, en combinant les relations d'ordre temporel locales et celles induites par la causalité, on obtient une relation d'ordre partiel [Lam78] sur les actions mais aucun ordre total ne peut généralement directement être mis en évidence.

9. L'instantané (*snapshot*) dont on parle ici n'est pas à rapprocher du concept de stabilisation instantanée dont on a discuté dans la section 3.1.4 page 26.

3.2.3 Obligation de prendre en compte les fautes

Dans le cas d'un algorithme centralisé, quand une panne crash ou plus généralement une faute permanente survient, il ne fonctionne plus. La tolérance aux fautes transitoires est parfois possible (nous l'avons vu dans la section « *Trois pratiques pour acquérir l'auto-stabilisation* » page 22, mais la volonté du concepteur d'algorithme d'intégrer un mécanisme pour les tolérer dépend à la fois du degré de confiance qu'on exige du système et de la probabilité de survenue des fautes transitoires.

Dans le cadre réparti, et principalement dans les systèmes répartis actuels où les systèmes sont à grande échelle et constitués de composants hétérogènes, les fautes (pannes ou attaques) [10] ne sont pas une éventualité mais une certitude. Il faut donc nécessairement les prendre en compte. Un sous-domaine de l'algorithmique répartie est donc dédié à l'étude d'algorithmes qui résistent à certains types de fautes. On fait généralement la distinction entre les algorithmes robustes [Ray10b, Ray10a, CGR11] (qui résistent à des fautes définitives des participants) et les algorithmes auto-stabilisants [Dol00] (qui résistent à n'importe quel type de faute transitoire), même si des approches multitolérantes sont régulièrement envisagées [AH93, BKM97, DT02, DPBT11].

3.2.4 Non-déterminisme des applications

Un programme informatique peut faire intervenir des variables d'entrée (fournies par l'utilisateur), des variables internes servant aux calculs et des variables de sortie (lues par l'utilisateur). En général, ce programme est fait pour résoudre un problème, c'est-à-dire que l'on demande que les variables de sortie soient déterminées en fonction des variables d'entrée selon une certaine logique définie par la spécification du problème. Dans le cas d'un algorithme centralisé, il est possible (mais pas toujours facile) de confronter, en vue de tests, les variables d'entrée et les variables de sortie d'un algorithme particulier, afin de s'assurer de son bon fonctionnement.

Par contre, si l'on considère un algorithme réparti, les variables de sortie sont, pour de nombreuses spécifications, situées sur des sites distants, et les calculs internes font intervenir des communications entre les sites. L'ordonnancement des actions (qui obéit à un ordre partiel) et les temps de communications induisent un non-déterminisme avec lequel il est nécessaire de composer lors de la conception des algorithmes. Cette difficulté est mise en évidence par les preuves de correction des algorithmes répartis : tant que l'on n'a pas prouvé la correction d'un algorithme réparti comme satisfaisant sa spécification pour *toutes* ses exécutions possibles, rien ne peut garantir qu'il aura le comportement attendu quand on l'exécutera dans un environnement réel. Cette explosion combinatoire des cas à prendre en compte (suite aux multiples exécutions possible à partir d'une configuration bien connue) est rendue encore plus difficilement gérable dans le cas de l'auto-stabilisation, car *toutes* les configurations initiales possibles deviennent autant de configurations initiales potentielles.

3.2.5 Difficulté de la connaissance des acteurs

Dans le cas des systèmes répartis à grande échelle on n'a pas toujours connaissance de l'identité des participants, et connaître la taille du réseau ou la liste des participants à un instant donné (même s'ils ont tous une identité distincte)

10. Une attaque est un comportement erroné dont l'initiateur est malveillant, une panne est une modification des mémoires du système qui relève d'un dysfonctionnement non intentionnel, nous regroupons les deux concepts sous le terme de fautes.

est donc une valeur qui doit être calculée si on en a besoin [CL85, DT02] (voir également la section « *Absence de connaissance de l'état global et difficulté d'initialisation* » page 31).

À cette première difficulté s'ajoute celle que, dans de nombreux cas, ces informations sont fluctuantes, voire inconnues, même de manière locale (par exemple, les participants vont et viennent continuellement comme dans un système pair-à-pair, ou bien les processus participants sont anonymes et/ou uniformes). Les hypothèses faites sur les communications inter-processus sont alors cruciales. En effet, **le principe même d'un algorithme réparti est de garantir un comportement global alors que les interactions entre les composants ne sont que locales**. Selon le modèle considéré, il est possible, soit de connaître l'identité réelle de ses voisins [AG94, NT09] (c'est-à-dire l'identifiant unique qui caractérise chaque voisin), soit de connaître chaque voisin seulement via le canal par lequel l'information est échangée avec lui [AK93, YK96a, YK96b, DT02, DDT06] (voir également A.1, A.3, A.4, A.5), voire de ne pas être en mesure de distinguer si deux messages reçus par un même processus sont en provenance de la même entité [DMRT11] (voir également A.2). [11]

3.2.6 Importance de l'auto-stabilisation dans le cadre réparti

Dans un cadre réparti, on peut résumer l'intérêt de l'auto-stabilisation par les points suivants :

1. On peut tolérer *toutes* les pannes transitoires, indépendamment de leur cause (processus, lien de communication, perturbation extérieure, etc.), de leur nature ou de leur étendue. Cette importance de l'auto-stabilisation dans le cadre de la tolérance aux pannes a été mise en évidence par Leslie Lamport lors de sa présentation plénière à PODC'83 [Lam84] puis par Jayaram et Varghese [VJ00] qui ont prouvé que de simples pannes crash avec reprise pouvaient mener un algorithme réparti dans une configuration arbitraire, à partir de laquelle seul un algorithme auto-stabilisant est capable de revenir à un comportement correct.
2. À la différence des algorithmes répartis classiques, pour lesquels on trouve fréquement des preuves de correction *opérationelles* (telle action implique telle autre action dans le futur, qui à son tour occasionne...) qui sont difficiles à lire, à comprendre, et à vérifier, les algorithmes répartis auto-stabilisants appellent à la rédaction de *preuves basées sur des invariants globaux* ou des mesures sur les configurations, dont les propriétés mathématiques de convergence sont à la fois plus simple à comprendre et mieux définies [12].
3. Le raisonnement à un niveau global, lors de la conception d'un algorithme auto-stabilisant et de sa preuve de correction, conduit souvent à se débarasser de tout ce qui n'est pas indispensable pour l'écriture de la preuve (variables redondantes, hypothèses superflues, propriétés inutiles) pour aboutir à des algorithmes les plus *simples* possibles, plutôt que les plus performants possibles (pour une certaine métrique, qui peut être plus ou moins pertinente selon le système réparti que l'on vise). Cette simplicité, recherchée pour pouvoir exprimer simplement des preuves de correction rigoureuses, est parfois à l'origine d'une efficacité inattendue [13].

11. Dans la littérature, on trouve aussi quelques cas où un nœud n'a aucun sens de l'orientation [BDKM96].

12. Plusieurs algorithmes annoncés comme auto-stabilisants mais dont les arguments de preuve de correction étaient rédigés de manière opérationelle se sont révélés faux quelques années plus tard [BCPV03, BPV04].

13. Le concept de stabilisation instantanée [BDPV99, BDPV07] par exemple, aurait été

3.3 Comprendre mes contributions

Maintenant que sont présentés la notion d'auto-stabilisation et le contexte du réparti, je vais lister les contributions principales que j'estime avoir apporté à ce domaine scientifique. Je range ces contributions en sept points du plus technique au plus général.

3.3.1 Élégance des preuves

Ayant une culture mathématique, j'aime faire des preuves techniquement fiables mais aussi lisibles. Nous avons déjà vu (page 28) que les mesures sur les configurations, aussi appelées fonctions de potentiel, constituaient une technique privilégiée pour satisfaire ce souci d'élégance. Les fonctions de potentiel avaient, jusqu'à présent, été utilisées pour prouver l'auto-stabilisation dans un modèle de communication par registres partagés [Tel94], qui implique principalement que l'espace des configurations est fini. J'ai étendu les preuves par fonctions de potentiel au modèle de communication par passage de messages [DT02, DDT06] (voir également les sections A.3 page 67 et A.4 page 79), où il existe en général un nombre infini de configurations (si les canaux de communication peuvent contenir un nombre arbitrairement grand de messages en transit). Cette extension passe par la définition de classes d'équivalence de configurations et d'une injection de ces classes d'équivalence vers les valeurs de la fonction de potentiel. À ma connaissance, ces techniques de preuve appliquées aux algorithmes auto-stabilisants par passage de messages sont les premières du genre.

Pour juger de l'apport de la méthode, il est intéressant de comparer le résultat initial sur l'auto-stabilisation des r-opérateurs [DT01] écrit pour un modèle de communications à registres partagés, dont les preuves sont longues et difficiles à lire – nombreuses notations et récurrences imbriquées – et sa version généralisée au modèle de communication par passage de messages [DDT06] incluant pertes, duplications et déséquencements (qui prend donc en compte un nombre d'exécutions possibles plus grand), dont la preuve est à la fois plus compacte et, je crois et j'espère, plus simple à comprendre.

3.3.2 Travail sur l'identité

Nous avons vu dans les sections « *Absence de connaissance de l'état global et difficulté d'initialisation* » page 31 et « *Difficulté de la connaissance des acteurs* » page 32 que la notion d'identité est cruciale.

Je m'y suis intéressée, d'une part, d'un point de vue de l'obtention de connaissances globales (les noms de *tous* les utilisateurs [DT02]) dans un contexte réparti classique où on connaît déjà son voisinage et, d'autre part, en envisageant une connaissance locale dans un contexte moins classique, celui des communications sans fil, où un problème est justement la découverte du voisinage.

Connaissances globales des identités. Cette connaissance doit naturellement être mise à jour lorsque des arrivées et des départs d'utilisateurs surviennent. La propriété d'auto-stabilisation est alors utilisée dans un but de mise à jour des informations plutôt que dans un but de tolérance aux pannes. Il faut donc s'attendre à ce que les défaillances ne soient plus transitoires mais intermittentes (de temps à autre, un nouvel utilisateur rejoint ou quitte le système réparti).

découvert suite à l'élaboration d'un algorithme auto-stabilisant comportant le nombre minumal d'états possibles par processus. Dans un autre registre, l'utilisation de r-opérateurs [DT01, DDT06] donne « gratuitement » lors de l'exécution dans un modèle à passage de messages la tolérance aux pertes, duplications et déséquencements.

Si le temps de stabilisation est élevé (par exemple, supérieur au temps moyen entre deux apparitions ou disparitions d'utilisateurs), le système reste dans des configurations illégitimes en permanence et n'est pas très utile. Si le temps de stabilisation est très petit par rapport au même temps moyen, le système contient des informations correctes la plupart du temps dans toute exécution.

Le temps de stabilisation de l'algorithme que nous avons proposé [DT02] est optimal pour le problème considéré (proportionel au diamètre du réseau). Par ailleurs, nous utilisons un modèle à passage de messages le moins restrictif possible : la topologie est quelconque, les liens de communication peuvent être unidirectionels [14] et les messages peuvent être perdus, déséquencés, ou dupliqués.

Connaissance des identités du voisinage Les communications sans fil ne relèvent plus du contexte réparti classique et présentent un défi particulier pour l'établissement de connaissances sur son voisinage par un processus. Quand on reçoit un message, on ignore qui en est l'expéditeur (les adresses réseau utilisées peuvent être falsifiées et on ne peut identifier un expéditeur à un lien de communication comme dans un réseau filaire). Comme dans un réseau filaire anonyme, on n'a pas de moyen de connaître l'identité de ses voisins et on ne peut pas non plus savoir combien ils sont si deux messages ont été émis par la même entité. Ces caractéristiques facilitent grandement les attaques contre un système réparti où les communications sont sans fil. Un type d'attaque particulier est celui des attaques Sybilles [Dou02] [15], où un même individu joue le rôle de plusieurs, dans le but de tromper les processus corrects : par exemple, un individu pourrait envoyer des messages de reconnaissance du réseau en utilisant plusieurs adresses réseau différentes pour faire croire aux processus corrects se trouvant à portée que leur voisinage est très dense. Ce type d'attaque empêche typiquement les protocoles d'accord Byzantin de fonctionner car ils sont, en général, basés sur l'hypothèse qu'une minorité de processus sont Byzantins ; avec une attaque Sybille, un seul Byzantin peut devenir toute une armée.

J'ai abordé la question de l'identification du voisinage en présence de processus malveillants de deux façons [VNTD08, DMRT11].

D'abord, comme l'attaque Sybille sans hypothèse supplémentaire est impossible à résoudre [Dou02], nous avons proposé, par analogie avec les détecteurs de défaillance [CT96, CHT96] dans le contexte de l'algorithmique robuste dans les systèmes répartis asynchrones avec crash, des *détecteurs d'Univers*. Informellement, un détecteur d'Univers [VNTD08] détermine, quand on lui présente plusieurs univers (par exemple des voisinages possibles qui, pris indépendamment, sont cohérents, mais qui, considérés deux à deux, sont incohérents entre eux), quel est l'univers qui correspond à la réalité. Nous en avons défini plusieurs variantes (comme pour les détecteurs de défaillances crash) en faisant varier les contraintes sur l'exactitude et la complétude. Il s'avère que le détecteur le plus faible, pour résoudre le problème de l'obtention du voisinage en cas d'attaque Sybille dans un système complètement asynchrone, est extrêmement puissant et impossible à implanter sans infrastructure cryptographique parfaitement sûre.

Ensuite, nous avons considéré un modèle d'exécution où les processus malveillants ne pouvaient envoyer des messages sur le support de communication sans fil qu'au même rythme que les processus corrects [DMRT11] (un malveillant ne peut donc tenter de créer une nouvelle identité qu'une fois par tour de communication), et nous avons considéré que l'identité des processus était confondue avec

14. Cet aspect présente un intérêt, par exemple, dans le cas de réseaux communicants sans fil, car des puissances d'émission différentes ou des obstacles peuvent conduire à ce qu'un processus p puisse communiquer avec un processus q, mais pas l'inverse.

15. Les attaques Sybilles ont été initialement définies dans le contexte des réseaux pair-à-pair.

ses coordonnées géographiques (c'est-à-dire, les coordonnées géographiques d'un processus – obtenues par un système GPS par exemple – lui servent d'identifiant). Obtenir un voisinage correct revient donc à obtenir les coordonnées géographiques réelles des processus qui nous envoient des messages. Du fait des propriétés physiques du signal transmis (et en particulier son affaiblissement dans l'espace et le fait qu'il ne peut être reçu avant d'avoir été émis), il est possible d'inférer des propriétés géométriques sur les positions géographiques qui peuvent être occupées par les processus dont on reçoit des messages. Par exemple [DMRT11] si on utilise l'intensité du signal reçu, quatre processus corrects (et non colinéaires), qui ne sont pas situés sur le même cercle, peuvent débusquer une minorité de processus malveillants qui mentent sur leur position réelle ; si on utilise un mécanisme d'aller-retour de communication, six processus corrects (et non colinéaires), qui ne sont pas situés sur la même hyperbole, peuvent débusquer une minorité de processus malveillants. Cette dernière contribution montre que, même si les communications sans fil renforcent la puissance des entités malveillantes sur certains côtés (possibilité de plusieurs identités en particulier), elles l'amoindrissent aussi si on prend en compte les propriétés physiques des communications.

3.3.3 Passage de messages

L'auto-stabilisation ne fait aucune hypothèse sur la nature ou l'étendue des pannes qui surviennent dans le système, à la condition qu'elles demeurent transitoires, c'est-à-dire qu'elles cessent de survenir après un temps donné. La plupart des travaux existants considèrent que les communications sont parfaitement fiables et établies dans un modèle de haut niveau où les communications entre voisins sont atomiques et instantanées. Pour implanter ces algorithmes dans un véritable réseau ou système distribué [16], il est nécessaire de recourir à des primitives de communications qui préservent la propriété d'auto-stabilisation, mais qui sont capables de s'exécuter dans des modèles répartis à granularité plus fine (qui correspondent aux réseaux réels). Une grande part de mes travaux a rapport avec la conception et la preuve d'algorithmes auto-stabilisants dans des systèmes répartis où les machines communiquent au moyen de passage de messages. D'après moi, ce modèle constitue un bon compromis entre la possibilité d'écrire des preuves mathématiquement satisfaisantes (et humainement lisibles) et la possibilité d'une implantation réelle.

En particulier, je me suis intéressée à plusieurs problèmes qui demandent une coordination globale des machines (c'est-à-dire les problèmes qui nécessitent d'obtenir des informations de toutes les autres machines du réseau, comme la construction d'un arbre de routage suivant une métrique particulière [GÖ3, DDT06] ou le recensement des machines dans un réseau [BDT99, DT02]) dans un contexte où les canaux de communication qui véhiculent les messages entre les machines ne sont pas fiables (des messages peuvent être perdus, dupliqués, déséquencés, que ce soit pendant la phase stabilisée ou pendant la phase de stabilisation), tout en préservant la propriété d'auto-stabilisation pour tolérer des corruptions initiales arbitraires de toutes les machines et de tous les contenus des canaux de communication.

Nous avons proposé une première approche [DT02], universelle (cette no-

16. Un système distribué est un système réparti, un algorithme réparti est un algorithme distribué et vice versa. La traduction française de « distributed » avait été définie comme étant « réparti » dans la même dynamique que « computer science » avait été baptisé « informatique » et non « science des computeurs » et que « computer » se traduit « ordinateur » et ni « computeur » ni « calculateur ». Aujourd'hui, on glisse vers « distribué » quand on veut insister sur le côté pratique du propos et on conserve « réparti » quand on veut parler français ou qu'on veut rappeler l'aspect sérieux et théorique du domaine.

tion d'universalité sera précisée dans la section« *Aide à la conception d'algorithmes auto-stabilisants* » page 39), qui donne une solution globale à tous les problèmes considérés tant que les machines possèdent des identifiants uniques. Les hypothèses faites sur le voisinage sont particulièrement faibles : un nœud ne connaît que le nombre de ses canaux entrants (numérotés arbitrairement et de manière purement locale), et aucun de ses canaux sortants. Si le graphe est fortement connexe, tous les problèmes impliquant la totalité des processus peuvent être résolus, sinon les problèmes impliquant les ancêtres du graphe de communication peuvent l'être, ce qui est le mieux que l'on puisse faire.

Nous avons ensuite affiné le concept [DDT06] en raisonnant sur les propriétés algébriques qui permettent l'acquisition implicite de la propriété d'auto-stabilisation en dépit de communications non fiables. Cette solution considère que les calculs effectués sur chaque nœud peuvent être exprimés sous la forme d'un opérateur défini sur les valeurs reçues des voisins, d'une part, et sur les constantes propres au nœud, d'autre part. Si l'opérateur est un infimum [Tel01], la stabilisation peut être assurée lorsque l'opérateur est binaire, associatif, commutatif et idempotent. Par exemple, si chaque nœud calcule le minimum des valeurs de ses antécédants, alors, au bout d'un temps fini, plus aucune nouvelle valeur ne sera produite. Ces infimums ne stabilisent cependant pas en présence de cycles dans le graphe de communication pour certaines valeurs initiales. Une généralisation des infimums, appelé r-opérateurs strictement idempotents [DT01], qui conduit à l'auto-stabilisation du système réparti dans tous les cas de figure est la suivante :

1. Pour un entier $n > 1$ donné, il existe une fonction $f : \mathbb{S}^n \mapsto \mathbb{S}$ définie par : $f(x_0, \ldots, x_n) = x_0 \oplus r_1(x_1) \oplus \cdots \oplus r_n(x_n)$;
2. L'ensemble de définition $\mathbb{S}$ est fini, ou $\mathbb{S}$ est infini et toute suite strictement croissante est non bornée ;
3. $\oplus$ est un infimum qui induit une relation d'ordre total $\preceq_\oplus$ sur $\mathbb{S}$ (*i.e.* $x \preceq_\oplus y \equiv x \oplus y = x$) ; on note $e_\oplus$ le plus grand élément de $\mathbb{S}$;
4. Les fonctions r_i sont des homomorphismes de $(\mathbb{S}, \oplus)$ ($r_i(x \oplus y) = r_i(x) \oplus r_i(y)$) ;
5. Pour toute fonction r_i et pour tout $x \in \mathbb{S} \setminus \{e_\oplus\}$, $x \prec_\oplus r_i(x)$ (*i.e.* $x \preceq_\oplus r_i(x)$ et $x \neq r_i(x)$).

Par exemple, que l'opérateur $(x, y) \mapsto \min(x, y + 1)$ vérifie ces propriétés et fournit un algorithme auto-stabilisant ; il permet de calculer la distance à la source (c'est-à-dire le processus dont la constante vaut zéro) la plus proche. Il était connu que l'auto-stabilisation était garantie quand la communication s'effectuait par registres partagés (que l'atomicité soit forte [DT03] ou faible [DT01]). Mon travail [DDT06] montre que la propriété peut être étendue aux communications par passage de messages, même en cas de pertes, duplications, ou déséquencements.

Enfin, je me suis intéressée à la propriété de stabilisation instantanée [BDPV07]. On a vu dans la section 3.1.4, « *Comparaison avec d'autres modes de stabilisation* » page 26 qu'un algorithme instantanément stabilisant doit répondre à la première requête qui lui est adressée de façon correcte (c'est à dire que le premier résultat fourni à l'utilisateur est correct vis-à-vis de la spécification du service). Jusqu'alors, tous les travaux présentaient des algorithmes instantanément stabilisants dans un modèle à communications locales *parfaites* (en une étape atomique, un nœud est capable de connaître *de manière immédiatement fiable* l'état du registre partagé de chacun de ses voisins et de mettre à jour son propre état). On voit qu'il y a là un problème potentiel d'implantation, car il n'existait jusqu'alors aucun algorithme capable de donner une telle garantie dans un modèle d'exécution réaliste comme celui de la communication par passage de

messages (c'est-à-dire, un algorithme instantanément stabilisant de communication entre voisins). Ma contribution a été, entre autres, [DDNT10] (voir la section A.5 « *Répondre immédiatement* » page 91 pour plus de détails sur les autres contributions), de permettre une telle implantation.

3.3.4 Multi-tolérance aux fautes

Dans le paragraphe « *Passage de messages* » page 36, j'ai déjà expliqué comment le passage à un modèle de communication par passage de messages (quand les travaux antérieurs considéraient un modèle de communication moins réaliste) permettait de faciliter l'implantation effective des solutions envisagées. Dans le cas des solutions auto-stabilisantes [DT02, DDT06], l'apport n'est pas seulement un changement de modèle, mais aussi un nouveau type de tolérance aux fautes : les canaux de communications ne sont pas fiables, ils peuvent perdre, dupliquer ou déséquencer les messages en transit [Lyn96]. Bien sûr, comme au paragraphe « *Travail sur l'identité* » page 34, on pourrait utiliser la propriété d'auto-stabilisation pour tolérer ces défaillances intermittentes en les considérant comme transitoires mais, si la moindre perte de message devait entraîner la restabilisation complète du système (occasionnant par là même de nombreuses retransmissions de messages, et donc de nombreuses nouvelles pertes), peu d'utilisateurs trouveraient cette infrastructure utile. La propriété que nous prouvons est plus forte : les pertes, duplications, et déséquencements ne perturbent en aucune manière le système (en particulier, ils ne modifient pas la valeur de la fonction de potentiel utilisée pour prouver leur convergence). Ils peuvent donc se produire pendant la phase de stabilisation et pendant la phase stabilisée. On a donc bien une multi-tolérance aux pannes transitoires et aux pannes intermittentes sur les canaux de communications dans la même exécution.

La tolérance aux pannes crash définitives et aux pannes transitoires est immédiate si les processus crashés peuvent être immédiatement considérés comme supprimés du système (c'est à dire, on dispose d'un détecteur de pannes crash parfait). Dans le cas contraire, de nombreux résultats d'impossibilité apparaissent [AH93]. Le cœur de ces résultats d'impossibilité réside dans le fait que l'on suppose que, dans une seule exécution, des pannes transitoires et des pannes crash définitives peuvent se produire. Une de mes contributions [DDP09] a été, tout d'abord, de considérer que si, dans une exécution donnée, on a soit des occurences de pannes crash définitives, soit des occurences de pannes transitoires (mais pas les deux dans la même exécution) alors il est possible d'obtenir un algorithme à la fois auto-stabilisant (quand des défaillances transitoires surviennent) et robuste (quand des défaillances définitives surviennent) de manière algorithmiquement très simple. De plus, comme un algorithme robuste, cet algorithme garantit la sûreté des résultats obtenus pour les mêmes occurences de pannes définitives. Il présente l'avantage, par rapport à un algorithme seulement robuste aux pannes crash définitives, de revenir à un comportement correct et robuste une fois le temps de stabilisation écoulé quand des défaillances transitoires surviennent (l'algorithme robuste non stabilisant, lui, ne satisfait *plus jamais* sa spécification en cas d'occurences de pannes transitoires). Cette unification montre que les deux approches sont complémentaires et qu'elles ne doivent pas être opposées, comme c'est trop souvent le cas.

3.3.5 Sûreté des applications

L'idée de tolérance aux pannes prônée par l'auto-stabilisation est optimiste [Tel00] (une machine essaie toujours d'exécuter son code quelles que soient les circonstances, en espérant que les choses rentrent dans l'ordre toutes seules). En

cela, elle est maintenant très répandue dans nombre de protocoles réseaux qui utilisent la notion de « best-effort » [Her03, Per00] (on fait toujours de son mieux, en espérant que cela soit suffisant pour les couches supérieures qui utilisent le service qu'on offre). Dans certains types de systèmes répartis, où les propriétés de sûreté des calculs effectués sont importantes (les machines parallèles, les grilles de calcul par exemple), l'auto-stabilisation ne rencontre pas encore un grand enthousiasme. Pourtant, les architectures multi-cœurs et les infrastructures de calcul scientifique qui sont actuellement envisagées comportent un nombre de composants tellement élevé que même un taux d'erreur très faible entraîne sur la globalité du calcul des défaillances très fréquentes (plusieurs par minutes) que les algorithmes déployés ne savent pas contrôler. On arrive ici à un problème insoluble car on veut exécuter un algorithme « sûr » dans un environnement « défaillant ». Les algorithmes « sûrs » classiques (non auto-stabilisants) fonctionnent en faisant l'hypothèse que l'environnement est sûr (correctement initialisé et sans défaillance). Les algorithmes auto-stabilisants « classiques » fonctionnent en ne faisant aucune hypothèse sur les défaillances qui ont pu se produire, mais ne donnent pas de garantie à l'utilisateur (ils ne peuvent pas donner de garantie puisqu'initialement le système peut être dans un état arbitraire). Certains de mes résultats ont en commun d'améliorer des algorithmes auto-stabilisants pour y ajouter des propriétés de sûreté :

1. Si le système n'est pas entièrement défaillant (par exemple, une minorité de machines ont leur mémoire corrompue), alors il est possible de donner des garanties de sûreté (c'est le cas de l'approche proposée dans [HG05, GCH06, DDP09]). Le principe est est assez simple : les variables utilisées sont suffisament redondantes pour permettre la récupération de la « vraie » valeur quand des bits sont inversés (à la manière des codes correcteurs d'erreurs). Opérer une telle correction avant chaque pas de calcul de l'algorithme principal permet de se prémunir contre les défaillances transitoires avec grande probabilité si on considère qu'elles se produisent suivant une loi de probabilité.
2. Si l'utilisateur du système considère que faire l'hypothèse d'un « ou exclusif » entre l'occurence de pannes crash définitives et pannes transitoires est raisonnable, j'ai expliqué au paragraphe « *Multi-tolérance aux fautes* » page 38 que des propriétés supérieures à celles des systèmes robustes classiques, d'une part, et à celles des systèmes auto-stabilisants classiques, d'autre part, pouvaient être obtenues.
3. Si la spécification du système est active (une requête est faite à l'une des machines du système réparti qui doit retourner une réponse à cette requête), alors il est possible de donner des garanties de sûreté à l'utilisateur qui effectue la requête (c'est le cas de l'approche proposée dans [DDNT10] et au paragraphe « *Passage de messages* » page 36). La stabilisation instantanée [BDPV07] garantit, en effet, que la première réponse à une requête est correcte, indépendamment de l'état dans lequel se trouve le système quand la requête est effectuée.

3.3.6 Aide à la conception d'algorithmes auto-stabilisants

Si des pannes, même transitoires, frappent le système, du fait du non-déterminisme des exécutions et de la dispersion des informations, il devient compliqué de déterminer, avec une connaissance seulement locale, si le résultat obtenu est vrai ou faux. Dans l'article, « Transient Fault detector » [BDDT98, BDDT07] (voir A.1 page 51), nous avons montré comment certains problèmes exigent une vue globale pour que l'obtention d'une configuration légitime puisse être testée,

tandis que d'autres nécessitent seulement le recours à une vérification locale. Ainsi, quand on ne dispose que des variables de sortie (celles qui apparaîssent dans la spécification du problème), savoir si l'exécution d'un algorithme réparti de construction d'arbre a bien mené à la construction d'un arbre demande une vision à distance $n/4$ (où n est le nombre de participants), savoir si l'exécution d'un algorithme réparti d'élection a bien mené à l'élection d'un unique candidat demande une vision globale du réseau : n, alors que savoir si l'exécution d'un algorithme réparti de coloration de graphe a bien mené à une coloration du graphe ne requiert que la vision des couleurs des voisins directs. Une telle mise en lumière du lien entre la spécification d'un problème et les connaissances qui sont nécessaires pour s'assurer de sa correction (en l'absence de connaissance sur l'algorithme qui sera exécuté pour obtenir cette solution) permet, par exemple, d'obtenir des transformations automatiques pour qu'un algorithme réparti « classique » [AD02, DS03] puisse devenir un algorithme réparti auto-stabilisant. Par exemple, il est possible de définir un testeur réparti d'algorithmes d'élection (ce testeur serait générique puisqu'il se base uniquement sur les variables de sortie de l'algorithme) qui, en cas de détection d'un problème, enclencherait une procédure de réinitialisation [AG94]. Cette brique logicielle n'étant pas spécifique à un algorithme particulier, elle pourrait même être vendue sur étagère. La contrepartie de cette généricité est le coût très élevé en terme de ressources (mémoire, informations échangés, etc.). Rien n'empêche cependant de spécialiser le détecteur de défaillances transitoires pour un algorithme particulier [BDDT07] : dans le cas de la construction d'un arbre, si au lieu de se baser sur seulement un pointeur vers un voisin (variable de sortie qui est la seule à intervenir lors de la construction d'un arbre), on dispose également d'une variable « distance à la racine » (l'algorithme spécifique qui construit l'arbre utilise cette variable additionnelle pour construire l'arbre), il n'est plus nécessaire de communiquer qu'avec ses voisins pour détecter une inconsistance.

Si l'approche par détection de fautes transitoires puis réinitialisation n'est pas envisageable pour des raisons de performance (c'est probablement le cas si la distance de vision nécessaire est trop importante), je peux alors conseiller à un utilisateur de se demander si sa spécification peut être calculée à l'aide d'un r-opérateur strictement idempotent (voir section « *Passage de messages* » page 36). Si c'est le cas, aucun travail supplémentaire n'est à fournir : l'algorithme est auto-stabilisant quel que soit le graphe de communication, orienté ou non, quel que soit le mode de communication (registres partagés, passage de messages), et tolère même des aléas sur les communications (pertes, duplications, déséquencements). Par ailleurs, aucun message ou variable supplémentaire n'est nécessaire pour garantir l'auto-stabilisation, celle-ci est *sans surcoût*.

Enfin, dans le cas où le problème de l'utilisateur revient à un problème de calcul de point fixe sur un graphe, et qu'on en connaît une solution centralisée, il est possible d'utiliser l'approche mentionnée dans la section « *Recensement* » page 12 pour en obtenir une version à la fois répartie et auto-stabilisante à moindre effort [17].

Bien évidement, hormis l'approche basée sur les r-opérateurs et quelques problèmes et algorithmes particuliers qu'il est peu coûteux de vérifier, ces techniques génériques sont susceptibles de fournir des solutions très peu performantes dans une situation réelle. La poursuite de la recherche vers des solutions *ad hoc* mais performantes reste d'actualité.

17. Cette pratique a d'ailleurs été utilisée dans un de mes articles récents [CD11].

3.3.7 Public différent

Aucune publication scientifique n'en est témoin, et les quelques films pouvant le relater ne peuvent être diffusés à cause du droit à l'image, mais je me suis impliquée dans la vulgarisation des concepts d'algorithmique répartie et d'auto-stabilisation en particulier.

L'algorithme de Dijsktra [Dij74] a été transformé en atelier de slam avec des participants allant de l'élève de 7 ans au retraité de 79 ans. Ce même public a compris les avantages de l'auto-stabilisation et le concept d'asynchronisme grâce à un jeu qui leur permettait de connaître les prénoms de chacun en tolérant les erreurs des uns et la disparition ou l'arrivée des autres (principe calqué sur un des algorithmes que j'ai conçus [DT98]).

Ces ateliers ont eu lieu pendant plusieurs années à la ferme du Moulon puis au LRI dans le cadre de la fête de la science où j'ai trouvé préférable de présenter des travaux de recherche plutôt que des concepts enseignés en première année d'université (voire au lycée) comme c'est majoritairement le cas.

Chapitre 4

Construire l'avenir

L'avenir c'est du passé en préparation.

Pierre Dac

Si toute activité de recherche est par nature impossible à prévoir, je trace ici quelques lignes directrices pour mon travail futur. Elles se déclinent suivant quatre axes, du plus immédiat au plus lointain. Je compte tout d'abord continuer le travail technique déjà entrepris (Section 4.1) et m'attacher à résoudre les questions ouvertes qui me paraissent pertinentes. J'ai l'intention de poursuivre mon ouverture à des modes de pensée différents (section 4.2 page 45) mais néanmoins connectés avec le travail présenté ici, et initié lors de plusieurs collaborations récentes. Les deux autres sections 4.3 page 47 et 4.4 page 48 représentent des ambitions à plus long terme.

4.1 Perspectives d'amélioration des techniques

4.1.1 À la poursuite du r-opérateur universel

Nous savons maintenant [DDT06] que, si l'algorithme réparti exécuté sur chaque nœud du réseau est un r-opérateur strictement idempotent et que l'ordre induit par le s-opérateur associé est total, alors le système est auto-stabilisant quelle que puisse être la topologie sous-jacente dans un modèle à passage de messages où les messages peuvent être perdus, dupliqués, ou déséquencés.

Avec les mêmes hypothèses de communications, nous avons montré [DT02] qu'il est possible, dans un réseau de communications fortement connexe, de résoudre le problème du recensement et de construire des solutions auto-stabilisantes pour tout type de problème global qui calcule un point fixe, à partir des données fournies aux participants du calcul.

Une extension naturelle de ces deux travaux serait de trouver un r-opérateur « universel », c'est-à-dire qui puisse être utilisé pour résoudre tout problème de point fixe [Duc07], dans un modèle de communication réaliste, dans le cas où le graphe de communication est fortement connexe. Un tel opérateur (dont l'ordre induit par le s-opérateur associé est seulement partiel) a été proposé pour un modèle de communication plus contraint [DT03] (où la communication s'effectue par mémoire partagée et où chaque participant peut lire l'état partagé par tous ses voisins de manière atomique), mais je pense que les résultats peuvent être étendus pour des réseaux où les communications sont non seulement plus réalistes (modèle à passage de messages, pas d'atomicité supposée au delà des primitives d'envoi et de réception de messages) mais aussi plus incertaines (pertes, duplications, déséquencement).

4.1.2 PAXOS, 15 ans après

Paxos [Lam98] est un algorithme réparti qui permet de résoudre le problème du consensus répété (c'est-à-dire à plusieurs instances successives du problème du consensus) en présence de pannes crash définitives dans un environnement globalement asynchrone. La propriété de sûreté (deux processus corrects décident la même valeur, et cette valeur est l'une de celles proposées) n'est jamais mise en défaut malgré l'asynchronie. La propriété de vivacité (un processus correct finit par décider d'une valeur) a besoin, pour être satisfaite, qu'une portion de l'exécution soit suffisament synchrone pendant suffisament longtemps.

La motivation principale pour le consensus répété est l'implantation d'une machine à états finis qui serait répliquée sur un grand nombre de sites. Les répliques de la machine fonctionnent en parallèle et doivent avoir le même comportement, c'est-à-dire exécuter les mêmes commandes à chaque étape du calcul. Exécuter une instance du consensus à chaque commande permet de garantir que la succession des commandes prises par chaque réplique de la machine est identique. Puisque les répliques des machines à états sont initialisées dans le même état au début de l'exécution, l'exécution de chaque réplique est identique. La réciproque est trivialement vraie : à chaque étape, il suffit de donner aux répliques la proposition qui sert d'entrée au consensus courant comme commande possible, et la machine à états répliquée résoud le consensus.

Il est naturel de s'interroger sur l'opportunité de résoudre ces problèmes (consensus répété et machine à états répliquée) dans un cadre auto-stabilisant, toujours en présence de pannes crash définitives. Dans ce cas, les répliques ne peuvent plus supposer qu'elles ont été initialisées dans le même état. À chaque pas du calcul, il faut donc se mettre d'accord, non seulement sur la commande à exécuter, mais aussi sur l'état courant dans lequel toutes les répliques sont supposées se trouver. La version auto-stabilisante du consensus répété demande simplement, qu'après un temps fini, toutes les instances de consensus qui sont exécutées répondent à la spécification du consensus. Du fait des initialisations arbitraires des états des répliques, un consensus répété auto-stabilisant ne peut servir en l'état à l'implantation d'une machine à états répliquée auto-stabilisante. En effet, même après stabilisation du consensus répété, seules les *nouvelles* commandes sont synchronisées et uniformes entre les répliques, l'historique des actions passées reste le même (et arbitrairement différent suivant les répliques). Ces deux problèmes, qui étaient équivalents en algorithmique répartie robuste « classique », ne le sont plus du tout dans la version auto-stabilisante du problème.

Dans le cadre de la thèse de Peva Blanchard et en collaboration avec Shlomi Dolev, nous avons proposé [BDBD13] une version *pratiquement* auto-stabilisante [1] de PAXOS. Pour retrouver l'équivalence entre consensus répété et machine à états répliquée malgré les fautes transitoires, nous avons proposé d'inclure l'historique de la réplique dans la proposition en entrée du consensus. Il me paraît important de poursuivre cet effort, en particulier de résoudre la question ouverte suivante : existe-t-il une version de PAXOS tolérante aux pannes crash définitives et *purement* (par opposition à pratiquement) auto-stabilisante ?

1. Par « pratiquement » auto-stabilisante, on entend que la propriété de correction – à partir d'une configuration légitime, seules des configurations légitimes sont atteignables – n'est plus vraie à tout jamais, mais pour un temps très long, par exemple exponentiellement long – 2^{64} exécutions de commandes

4.2 Perspective d'ouverture à la différence

4.2.1 Egoïsme et malveillance

Un point commun à la très grande majorité des travaux de recherche dans le domaine de l'auto-stabilisation est que l'*intégralité* des machines coopère à une tâche commune. L'adversaire place les machines dans des états arbitraires et sélectionne l'ordre d'exécution des machines, de telle sorte que le temps avant de retrouver un comportement correct est le plus long possible, mais les machines elles-mêmes restent unies dans l'adversité.

L'introduction de comportements malveillants (c'est-à-dire Byzantins) dans le contexte de l'auto-stabilisation conduit au problème suivant : il est impossible de distinguer une machine qui diffuse une mauvaise valeur parce qu'elle est mal initialisée d'une machine qui diffuse une mauvaise valeur parce qu'elle veut nuire à l'accomplissement de la tâche commune. De plus, si une machine détecte une inconsistance (entre son état et celui de son voisin), il lui est impossible de distinguer les deux situations suivantes :

1. Son propre état est incorrect et doit être corrigé ;
2. L'état de son voisin est incorrect et son propre état doit être préservé.

Pour garantir l'auto-stabilisation à partir de toute configuration initiale, c'est la première situation qui est choisie en général, ce qui donne aux machines compromises un pouvoir de nuisance substantiel. D'ailleurs, la plupart des travaux qui mélangent auto-stabilisation et comportements Byzantins comprennent des résultats d'impossibilité [AH93] ou restreignent le modèle d'exécution (par exemple, démon central [MT07b]), de communication (par exemple, graphe complet [DW04]), d'attaque (par exemple, au plus une machine Byzantine [DMT12]), où le type de problèmes à résoudre (par exemple, des tâches locales comme le coloriage des nœuds ou le dîner de philosophes [NA02]).

Un pan récent de mes activités traîte de la possibilité que certaines machines aient un comportement partiellement déviant, et en particulier *égoïste*. Ainsi, les machines malveillantes n'ont plus un comportement arbitraire seulement dirigé à la mise en défaut du système, mais poursuivent un intérêt qui leur est propre (par exemple, un prédicat localement calculé à maximiser). Le pouvoir de nuisance de telles machines est évidemment inférieur à celui de véritables Byzantins, ce qui laisse la possibilité d'obtenir des résultats positifs pour des problèmes globaux même avec des graphes de communication peu denses et sans contrainte particulière sur le nombre d'entités égoïstes.

Plus précisément, je me suis intéressée au problème de la construction autostabilisante d'un arbre couvrant, mais en donnant la possibilité aux nœuds d'avoir des intérêts divergents sur la métrique à maximiser [CDGT08]. Ainsi, les machines se rejoignent concernant l'intérêt général (construire un arbre couvrant) mais sont en compétition sur leur intérêt particulier (maximiser la métrique qui leur plaît). La rationalité des comportements égoïstes a permis d'introduire des éléments de théorie des jeux et de compétition dans un contexte traditionnellement centré sur la coopération et l'altruisme. Une première publication sur ces aspects [CD11] m'a permis de trouver une condition suffisante pour l'existence d'une solution auto-stabilisante au problème de la construction d'un arbre de plus courts chemins avec coalitions. Dans ce problème, deux (ou plus) groupes de machines sont interconnectés et cherchent à construire un arbre couvrant. Cependant, les coûts des liens utilisés diffèrent suivant les groupes. Savoir s'il existe un équilibre (c'est-à-dire un arbre stable pour tous les groupes) est notoirement NP-complet [CDGT08] dès qu'il y a au moins deux groupes. Mon travail démontre qu'une condition suffisante pour qu'il existe un équilibre (dérivée d'une condition

de Griffin *et al.* [GSW02] pour la stabilité du routage interdomaine dans Internet) est aussi une condition suffisante pour qu'il existe une solution auto-stabilisante au problème. Caractériser de manière plus approfondie les problèmes pour lesquels l'auto-stabilisation en dépit d'intérêts divergents peut être garantie est un défi passionnant.

4.2.2 Colonie de Zèbres

Les protocoles de population [AAD+06] constituent un modèle théorique récent pour étudier les systèmes répartis constitués de petites entités mobiles qui interagissent (c'est-à-dire communiquent) quand elles se trouvent à proximité l'une de l'autre. Initialement motivé par la pose de petits capteurs mobiles sur des animaux vivants, le modèle a connu un engouement certain de la part de la communauté théorique de part ses propriétés de calcul intéressantes (en gros, le modèle initialement proposé permet de calculer tout prédicat exprimable dans l'arithmétique de Presburger [AAER07]).

L'une des hypothèses les plus importantes pour établir des résultats positifs sur ce qui est effectivement calculable dans le domaine est celle de l'*équité globale* : au cours de l'exécution du système, toute entité interagit obligatoirement avec toutes les autres, et ce, indéfiniment. Je me suis pour ma part intéressée avec Peva Blanchard à l'efficacité des calculs effectués [BBBD12] en liant cette notion d'équité globale à des comportements mobiles dans des réseaux de capteurs *réels*. Plus spécifiquement, l'expérience ZebraNet consiste à placer des capteurs de petite taille sur une colonie de zèbres évoluant au Kenya, pour collecter *via* une station de base diverses informations biométriques liées à chacun des individus de la colonie. A partir des interactions entre les zèbres, il a été possible de quantifier le temps de couverture moyen d'un individu (c'est-à-dire le temps mis pour rencontrer chacun des autres individus de la colonie). Cette caractérisation théorique a permis d'expliquer la perte d'un pourcentage significatif (de l'ordre de 10%) de données dans l'expérience initiale, et de proposer une amélioration du protocole d'échange d'informations pour pallier ce problème.

L'introduction de la tolérance aux panness [DGFG05, DGFGR06, GR09] et de l'auto-stabilisation [AAFJ08, SNY+09, CIW12, MOKY12] dans le contexte des protocoles de populations a pour l'instant considéré les variantes théoriques du modèle (notamment l'hypothèse de l'équité, globale ou locale quand les interactions entre individus sont contraintes par une topologie particulière). Il me parait intéressant de confronter la possibilité d'obtenir des variantes auto-stabilisantes ou tolérantes aux pannes dans le cas où l'équité n'est plus inéluctable mais réaliste et basée sur des observations de réseaux réels.

4.2.3 Formations de Robots

Les réseaux de robots [SY99, FPS12] ont fait l'objet d'une attention importante ces dernières années de la part de la communauté du calcul réparti. La différence essentielle entre le modèle théorique développé pour les étudier et les modèles classiques est que les communications ne sont plus explicites (envoi/réception de messages, lecture/écriture dans une mémoire partagée) mais *implicites* (les robots observent les positions des autres robots dans un référentiel egocentré, et se déplacent sur la base des observations précédentes). On peut bien sûr faire une analogie [MH13] entre la lecture/réception et l'observation d'une part, et entre l'émission/écriture et le déplacement d'autre part, mais deux caractéristiques des réseaux de robots sont inhabituelles dans le contexte réparti classique :

1. Les « valeurs » (c'est-à-dire positions) ont un sens géométrique (même si elles sont exprimées pour un robot particulier dans un référentiel qui lui est propre – en général il n'y a pas d'orientation commune ou même de notion commune de droite et de gauche – et la plupart des approches sont donc basées sur des constructions géométriques, comme le plus petit cercle englobant, l'enveloppe convexe, le centre de gravité, le point Weber, etc.

2. Les robots sont « oublieux » dans le sens où à chaque nouvelle observation, la mémoire du passé dans l'exécution en cours est effacée.

La première caractéristique conduit à se diriger vers des problèmes qui ont également un sens géométrique, comme la formation de motifs géométriques [DFSY10], le rassemblement en un point [CFPS12] ou la dispersion dans un plan [DP09]. La deuxième caractéristique a conduit la communauté à penser que les réseaux de robots étaient « moralement » auto-stabilisants du fait que leur mémoire était réinitialisée régulièrement. Ce dernier point n'est cependant pas totalement vrai car, si on considère les configurations du système comme un tout (ce que l'on fait dans l'auto-stabilisation « classique »), de nombreux articles font des hypothèses cruciales sur le placement initial des robots ; typiquement, dans un algorithme de rassemblement [CFPS12], il est interdit à deux robots d'occuper la même position initialement mais ils seront autorisés à le faire à une date ultérieure (c'est même le but de l'algorithme de rassemblement). Certaines configurations sont donc atteignables mais interdites initialement, ce qui est contraire au principe d'auto-stabilisation. Il y a encore peu de travaux qui mélangent « vraie » auto-stabilisation (au sens des configurations globales) et réseaux de robots [DP12, OT12] et ils considèrent uniquement le cas des robots oublieux. Or, de nouvelles extensions au modèle de base ont été récemment proposées [DFP$^+$12, FSVY13] : un robot peut avoir une mémoire persistante et cette mémoire peut ou non être visible des autres robots. Étudier l'auto-stabilisation de réseaux de robots dans ce modèle enrichi me semble être propice à des découvertes intéressantes.

4.3 Perspective de diffusion plus large

4.3.1 Diffusion à destination des spécialistes

Après ma thèse de troisième cycle soutenue en 1995, mon projet était de diffuser l'idée de l'auto-stabilisation dans le monde académique et le monde industriel. Pendant 15 ans, j'ai continué tranquillement à découvrir de mon côté des algorithmes répartis novateurs mais j'ai pris le temps de « prêcher l'auto-stabilisation et les preuves théoriques » auprès de mes collègues chercheurs proches de la pratique, au point que, maintenant, j'entende dire : « tout le monde parle d'auto-stabilisation ». J'estime aujourd'hui que ce projet a pris du temps mais a réussi (ne serait ce que par le nombre de thèses et d'habilitations à diriger des recherches françaises soutenues faisant référence directement à l'auto-stabilisation à ma connaissance les HDR de Alain Bui (1999), Vincent Villain (1999), Franck Petit (2003), Bertrand Ducourthial (2005), Olivier Flauzac (2005), Sébastien Tixeuil (2006), Colette Johnen (2007), Alain Cournier (2009), Maria Potop-Butucaru (2010), Lelia Blin (2011)).

De même après cette habilitation à diriger les recherches je continuerai à élargir mes connaissances et faire comprendre l'intérêt de cette notion d'auto-stabilisation, de ses limites et de ses atouts.

4.3.2 Diffusion à destination du grand public

Dans la même dynamique que celle du paragraphe précédant, je pense qu'il serait utile de dépenser de l'énergie à comprendre, par exemple par une approche sociologique, pourquoi les mécanismes de pensée de l'algorithmique répartie (localité des informations et des interactions, localité du temps, défaillances inopinées pour un objectif global) sont si étrangèrs aux jeunes scientifiques français et réfléchir à l'élaboration de manières ludiques pour faire connaître ces principes et rendre naturelle cette approche aux générations futures.

De même que les conséquences du hasard sont bien intégrées dans les jeux enfantins de dés comme les petits chevaux ou le jeu de l'oie, que la notion d'ordre ou de séquence est bien répandue grâce aux jeux de cartes traditionnels, il existe probablement des jeux qui aident à comprendre les concepts de l'algorithmique répartie. Pour avoir organisé (comme je le raconte dans la section 3.3.7 page 41) des ateliers sur l'auto-stabilisation et l'algorithmique répartie à destination des jeunes écoliers lorsque j'organisais la fête de la science, j'ai pu constater que plusieurs des notions fondamentales leur étaient très naturelles, car elles correspondent à une réalité physique : chaque écolier est une entité distincte, qui évolue à son rythme (localité du temps), qui communique avec ses camarades en petit comité (localité des interactions), et trouve amusant de participer à un jeu où il faut reconstruire des informations globales alors que celles qui leur sont fournies sont toutes locales, mélangées ou transformées [2].

Il me paraît important de mettre en évidence ces concepts et de travailler à vulgariser et à diffuser ce type de pensée auprès des formateurs d'enseignants des jeunes générations.

4.4 Perspective de l'ordinateur auto-stabilisant

On peut toujours espérer que tous les acteurs de la recherche en calcul réparti se mettent à faire de l'auto-stabilisation. L'intérêt de la notion est déjà important et largement répandu dans les réseaux actuels qui sont basés sur le principe du « *best effort* », un concept proche de l'auto-stabilisation [Per00].

Au delà de cette utopie, il faut être conscient que les ordinateurs d'aujourd'hui (ce qui inclut téléphones portables et autres tablettes) sont conçus à base de composants non auto-stabilisants et utilisent une large quantité d'algorithmes qui ne sont pas non-plus auto-stabilisants. Une perspective à très long terme serait de faire disparaître cet état de fait. L'équipe que dirige Shlomi Dolev à l'Université de Beer Sheva considère que l'objectif de construire un ordinateur auto-stabilisant est possible avec la technologie actuelle, mais pas encore rentable. Je collabore, à ma mesure, avec eux et j'espère avoir l'occasion un jour de voir un tel ordinateur. Étant donnés les enjeux financiers en la matière, il est peu probable que je puisse posséder, avant ma retraite, un ordinateur qui soit totalement auto-stabilisant. Par contre, j'ai déjà croisé un grand nombre d'ordinateurs défaillants et d'interventions humaines pour remettre des systèmes Informatiques en route...

Concevoir des ordinateurs auto-stabilisants, c'est accepter de mettre un peu d'imperfection, donc d'humanité, dans les ordinateurs au lieu de mettre trop de perfection, donc d'inhumanité, dans les humains comme le monde actuel à tendance à le faire.

En conclusion, comme je l'ai entendu récemment sur France Inter : « Le téléphone portable n'est pas né d'un programme gouvernemental pour l'optimisation du vol des pigeons voyageurs. » Mon projet premier reste de continuer à

2. Attention : réfléchir à une approche ludique pour la formation à l'algorithmique répartie n'a rien à voir avec la théorie des jeux telle qu'elle est présentée dans la section 4.2.1.

faire et à faire faire au mieux de mes capacités et de mes compétences un travail scientifique de qualité ; de comprendre et de faire comprendre le monde qui m'entoure ; de défendre l'idée que l'auto-stabilisation est un principe phare de l'informatique moderne.

FIGURE 4.1 – Petit message de Ben

Annexe A

Agrafage

Le grand inconvénient des livres nouveaux est de nous empêcher de lire les anciens.

Joseph Joubert

Dans ce chapitre, je reproduis mes principales publications dans des revues en les introduisant avec leur résumé. Une section est consacrée à chacune des cinq publications mise en valeur dans les pages précédentes de ce manuscrit.

A.1 Cerner les problèmes

Transient Fault Detectors. Joffroy Beauquier, Sylvie Delaët, Shlomi Dolev, and Sébastien Tixeuil. *Distributed Computing*, 20(1) :39-51, June 2007.

– *Résumé :* Dans cet article, nous présentons des détecteurs de défaillances qui détectent des défaillances transitoires, i.e. la corruption de l'état du système sans qu'elle soit due à la corruption du code exécuté. Nous distinguons les tâches qui représentent les problèmes à résoudre, des implantations qui sont les algorithmes résolvant les problèmes. Une tâche est spécifiée par la sortie que doit produire le système réparti. Le mécanisme qui est effectivement employé pour produire cette sortie concerne l'implantation et non la tâche. De plus, nous sommes en mesure de définir des propriétés de localité en distance et de localité en temps pour les tâches que nous considérons. La localité en distance est liée au diamètre de la configuration du système qu'un détecteur de défaillances doit maintenir pour être en mesure de détecter une défaillance transitoire. La localité en temps est liée au nombre de configurations système successives qu'un détecteur de défaillances doit maintenir pour détecter une défaillance transitoire. Tant la localité en temps que la localité en distance peuvent donner aux personnes responsables des implantations des informations concernant les ressources et techniques nécessaires pour implanter effectivement la solution.

Distrib. Comput. (2007) 20:39–51
DOI 10.1007/s00446-007-0029-x

Transient fault detectors

Joffroy Beauquier · Sylvie Delaët · Shlomi Dolev · Sébastien Tixeuil

Received: 15 August 1999 / Accepted: 10 March 2007 / Published online: 5 June 2007

Abstract We present fault detectors for transient faults, (i.e., corruptions of the memory of the processors, but not of the code of the processors). We distinguish fault detectors for *tasks* (i.e., the problem to be solved) from failure detectors for *implementations* (i.e., the algorithm that solves the problem). The aim of our fault detectors is to detect a memory corruption as soon as possible. We study the amount of memory needed by the fault detectors for some specific tasks, and give bounds for each task. The amount of memory is related to the size and the number of views that a processor has to maintain to ensure a quick detection. This work may give the implementation designer hints concerning the techniques and resources that are required for implementing a task.

Keywords Distributed systems · Transient faults · Fault detectors · Self-stabilization

An extended abstract of this paper was presented at the *12th International Symposium on DIStributed Computing (DISC'98).* Shlomi Dolev is partly supported by the Israeli Ministry of Science and Arts grant #6756195. Part of this research was done while Shlomi Dolev was visiting the Laboratoire de Recherche en Informatique (LRI), University of Paris Sud.

J. Beauquier ( ) · S. Tixeuil
University of Paris Sud, LRI-CNRS 8623, INRIA Grand Large, 91405 Orsay, France
e-mail: jb@lri.fr

S. Tixeuil
e-mail: tixeuil@lri.fr

S. Delaët
University of Paris Sud, LRI-CNRS 8623, 91405 Orsay, France
e-mail: delaet@lri.fr

S. Dolev
Department of Mathematics and Computer Science, Ben-Gurion University, Beer-Sheva 84105, Israel
e-mail: dolev@cs.bgu.ac.il

1 Introduction

In a system that may experience transient faults, it is impossible for the processors to "know" that the system is currently in a consistent state: assume that every processor has a boolean variable that is true whenever the processor knows that the system is in a consistent state and is false otherwise. The value of this variable may not reflect the situation of the system since it is subject to transient faults. This is why the processors in self-stabilizing systems must continue the execution of the algorithm forever and never know for sure that the system is stabilized.

In this paper we propose a tool for identifying the inconsistency of a system, namely a transient fault detector. Identification of a transient fault can be coupled with a self-stabilizing reset procedure to obtain a self-stabilizing algorithm from an existing non-self-stabilizing algorithm (see e.g., [9,11]). The requirement that every processor will know whether the system is in a consistent state is relaxed; instead we require that at least one processor identifies the occurrence of a fault when the system is in an inconsistent state. Moreover, the transient fault detector is unreliable since it can detect an inconsistent state as a result of a transient fault that corrupts the state of the fault detector itself. The only guarantees we have are that when the system is not in a consistent state a fault is detected, and when both the system and the fault detector are in a consistent state no fault is detected.

Our focus in this paper is on the implementation of fault detectors and not in the operations invoked as a result of detecting a fault; we just mention two such possible operations, namely: resetting (e.g., [4]) and repairing (e.g., [1,13,19]).

In this paper we present fault detectors that detect transient faults, i.e., corruption of the system state without corrupting the program of the processors. We distinguish *task* which is

the problem to solve, from *implementation* which is the algorithm that solves the problem. A task is specified as a desired output of the distributed system. The mechanism used to produce this output is not a concern of the task but a concern of the implementation. We study transient fault detectors for tasks and for implementations, separately. Designing fault detectors for tasks (and not for a specific implementation) gives the implementation designers the flexibility of changing the implementation without modifying the fault detector.

In addition, we are able to classify both the *distance locality* and the *history locality* property of tasks. The distance locality is related to the diameter of the system configuration that a processor has to maintain in order to detect a transient fault. The history locality is related to the number of consecutive system configurations that a processor has to maintain in order to detect a transient fault.

Both the distance and the history locality of a task may give the implementation designer hints concerning the techniques and resources that are required for implementing the task.

Then we turn to investigate fault detectors for a specific implementation—a specific algorithm. Obviously, one may use a fault detector for the task of the algorithm without considering the data structures and techniques used by the algorithm. However, we are able to show that, in many cases, the amount of resources required is dramatically reduced when we use fault detectors for a specific implementation and not a fault detector for the task.

Related work The term "failure detector" was introduced in a different context in [6], where an abstract failure detector is used for coping with asynchrony and solving consensus. In the context of self-stabilizing systems, checking the consistency of a distributed system was used in [20] where a snapshot of the system is repeatedly collected. Fault detectors, called observers, that are initialized correctly and are not subject to state corruption are used in [22]. Monitoring consistency *locally* for a restricted set of *algorithms* has been suggested in e.g., [2,3,7,17,8,18]. A local monitoring scheme for every on-line and off-line algorithm has been presented in [1]. The local monitoring technique of [1] is a general technique that monitors the consistency of any *algorithm*. The method of [1] uses pyramids of snapshots, and therefore the memory requirement of each processor is related to the size of the system. In this work we present a hierarchy of fault detectors for *tasks* and *algorithms* that is based on the amount of information used by the fault detector. It is interesting that different distributed *tasks* require different amounts of memory for detecting transient faults. We note that a transient fault detector that is designed for a particular *algorithm* may use the variables of the algorithm for fault detection.

The rest of the paper is organized as follows: The system is described in Sect. 2, fault detectors for asynchronous silent tasks are considered in Sect. 3 and for synchronous non silent tasks in Sect. 4, respectively. Fault detectors for implementation (algorithms) are presented in Sect. 5. Implementation details of transient fault detectors appear in Sect. 6 and concluding remarks are presented in Sect. 7.

2 The system

Distributed system Our system settings are similar to those presented in [14]. We consider an asynchronous system of n processors, each processor resides on a distinct node of the system's *communication graph* $G(V, E)$, where V is the set of *vertices* and E is the set of *edges*. Two processors that are connected by an edge of G are *neighbors*. Communication among neighboring processors is carried out by *communication registers*. In the sequel we use the term registers for communication registers. An edge (i, j) of G stands for two registers $r_{i,j}$ and $r_{j,i}$. P_i (P_j) can write in $r_{i,j}$ ($r_{j,i}$, respectively) and both processors can read both registers. The registers in which a processor P_i writes are the registers of P_i.

Let u and v be two nodes of the communication graph, $Dist(u, v)$ denotes the number of edges on a shortest path from u to v (if such a path exists). Let $MaxDist(u)$ be the maximal value of $Dist(u, v)$ over all nodes v in the system. A node u is a *center* of G if there is no node v such that $MaxDist(u) > MaxDist(v)$. The *radius*, r, of G is the value of $MaxDist(u)$ for a center node u of G. The *diameter*, D, of G is the maximal value of $MaxDist(v)$ over all nodes v in the system. Let i be an integer, $Ball(u, i)$ denotes the set B of nodes $b \in B$ such that $Dist(u, b) \leq i$.

Configurations and runs Each processor is a finite state machine whose program is composed of *steps*. Processors have unique identifiers. The *state* of a processor consists of the values of its internal variables and its communication registers i.e., the registers to which it writes. Let S_i be the set of states of the processor P_i. A *configuration*, $c \in (S_1 \times S_2 \times \cdots S_n)$ is a vector of the states of all the processors.

An *asynchronous run* of the system is a finite or infinite sequence of configurations $R = (c_1, c_2, \ldots)$ such that c_{i+1} is reached from c_i by a step of one processor. In such a step, a processor may execute internal computations followed by a read or write operation. This scheduling policy is known as the *read–write* atomicity model. We use the *cycle complexity* (see [10,13,15]) to measure time in asynchronous system. The first cycle of a run is the minimal prefix of the run in which each processor reads the registers of all its neighbors and writes to its registers. The second cycle is the first cycle of the rest of the run, and so on. In an asynchronous run, time i relates to the ith cycle.

A *synchronous run* of the system is a finite or infinite sequence of configurations $R = (c_1, c_2, \ldots)$ such that c_{i+1} is reached from c_i by the following steps of the processors: first every processor reads the register of its neighbors; once every processor finishes reading, the processors change state and write into their registers. In a synchronous run, time i relates to the ith configuration.

Specifications and fault detectors An *abstract run* is a run in which only the values of a subset of the state variables, called the *output variables* are shown in each configuration. The *specification* P of a task T for a given system $\mathcal{S}$ is a (possibly infinite) set of abstract runs of $\mathcal{S}$. For example, the mutual exclusion task is defined by a set of abstract runs, such that in each run in this set at most one processor executes the critical section at a time and every processor executes the critical section infinitely often—existence of an output boolean variable that indicates whether the processor is executing the critical section is assumed.

$\mathcal{S}$ is *self-stabilizing* (in fact *pseudo self-stabilizing* [5]) with relation to P if and only if each of its runs has a suffix in P.

The goal of a *transient fault detector* is to check whether a particular run of the system $\mathcal{S}$ matches the specification P. More precisely, a fault detector is assimilated to a boolean variable that obtains the value $true$ if and only if the specification is satisfied. Our fault detectors use the concept of *views* and *histories*, that are defined for a specification P.

Definition 1 (View) The view $\mathcal{V}_i^d$ at distance d from a processor P_i contains: (i) the subgraph of the system communication graph G containing the nodes in $Ball(i, d)$ and (ii) the set of output variables that are associated with those nodes for a specification P.

Definition 2 (History) The history $\mathcal{V}_i^d[1 \ldots s]$ at distance d from a processor P_i is a sequence of s consecutive views at distance d from P_i.

The history element $\mathcal{V}_i^d[j]$, for a given index j, is the view at distance d from P_i at time j. View $\mathcal{V}_i^d[1]$ is associated with present time, while view $\mathcal{V}_i^d[s]$ is associated with $s-1$ time units in the past.

The fault detectors that we consider in this paper are *distributed* in the sense that they can be invoked by each of the system processors and give a different response to each of them. However, we assume that the response of a fault detector to a processor is uniquely determined by the history of the invoking processor. More precisely, the response of the oracle is *true* if and only if a predicate on the history of the calling process (induced by the specification of the task) is also true.

Definition 3 (Fault detector at a processor) The fault detector $\mathcal{FD}_d^s(P_i)$ at processor P_i is an oracle that can be invoked by P_i and whose binary response is uniquely determined by the local history $\mathcal{V}_i^d[1 \ldots s]$ of P_i.

The result of $\mathcal{FD}_d^s(P_i)$ can be used in any configuration c_j of a system run. The result of the oracle $\mathcal{FD}_d^s(P_i)$ in configuration c_j is denoted by $\mathcal{FD}_d^s(P_i, c_j)$.

Definition 4 (Fault detector in a configuration) The distributed result $\mathcal{FD}_d^s(P_i, c_j)$ of a fault detector in a configuration c_j is the conjunction of the $\mathcal{FD}_d^s(P_i, c_j)$ results for each processor P_i in the system.

Intuitively, if in a given system configuration, the task specification is satisfied, then every oracle invocation at every processor must return *true*. In the case of a system configuration where the task specification is not satisfied, at least one of the fault detectors at a processor must return *false* to the invoking processor.

Task classification A task is (d, s)—local if and only if there exists a fault detector that uses s consecutive views at distance d and is correct for this task, while there exist no correct fault detectors for this task that use at most $s-1$ consecutive views or views at distance at most $d-1$. More formally, the following conditions are verified:

1. For each configuration c of any system $\mathcal{S}$ and for all $1 \leq i \leq n$ $\mathcal{FD}_d^s(P_i, c)$ returns $true$ if c is correct (i.e., matches the task specification), and there exists an index $1 \leq i \leq n$ such that $\mathcal{FD}_d^s(P_i, c)$ returns $false$ if c is incorrect (i.e., does not match its specification).
2. For any fault detector in $\mathcal{FD}_{d-1}^s(P_i, c)$, there exists a configuration c of a particular system $\mathcal{S}$ that is correct (i.e., matches the task specification) and index k such that $\mathcal{FD}_{d-1}^s(P_k, c)$ returns $false$ or there exists an incorrect configuration c' in which for every $1 \leq k \leq n$ $\mathcal{FD}_{d-1}^s(P_k, c')$ returns $true$.
3. For any fault detector in $\mathcal{FD}_d^{s-1}(P_i, c)$, there exists a configuration c of a particular system $\mathcal{S}$ that is correct (i.e., matches the task specification) and index k such that $\mathcal{FD}_d^{s-1}(P_k, c)$ returns $false$ or there exists an incorrect configuration c' in which for every $1 \leq k \leq n$, $\mathcal{FD}_d^{s-1}(P_k, c')$ returns $true$.

Locality criteria It turned out that the fault detection capabilities of a fault detector is related to the amount of information it stores. We distinguish two parameters that are related to the storage used by a fault detector:

- Distance: the distance d of $\mathcal{V}_i^d[1 \ldots s]$, where d is between 0 and $r+1$, and r is the radius of the system.
- History: the number s of views in $\mathcal{V}_i^d[1 \ldots s]$.

The two locality criteria that we consider for tasks refer to the ease that transient faults will be triggered. The smaller the distance or time locality, the less information oracles use to give a correct response.

Roughly speaking, if the result of the oracle at a processor P_i uses a history where only P_i's output variables are present, the predicate that is evaluated by the oracle is *purely local*: it is independent from the neighborhood of P_i. In the particular case when the distance d of the history of each P_i equals 1, the oracle's predicate is *local*. If $d = r + 1$, the predicate is *global*.

3 Fault detectors for silent tasks

In this section we study fault detectors for silent tasks ([12]) where the output variables remain fixed from some point of the run. Since silent tasks are 0—history local, in the following, we assume that the locality property only refers to distance locality. Therefore, we use $\mathcal{V}_i^d$ instead of $\mathcal{V}_i^d[1]$ to denote the history of P_i, and $\mathcal{FD}_d$ instead of $\mathcal{FD}_d^1$.

We now list several silent tasks, present their specification, and identify the minimal distance required for their fault detectors.

3.1 Leader election, center determination, and number of nodes

In this section we present a class of tasks that requires that at least one processor has a view of the entire system. In Sect. 2, we assumed that the $\mathcal{V}_i^d$ views contain the communication graph description up to distance d from P_i. There are two possible basic assumptions about the communication graph information that is stored in the $\mathcal{V}_i^d$ views:

Assumption 1 (Links included) The edge identifiers of the network are stored in the $\mathcal{V}_i^d$ views. In other words, $\mathcal{V}_i^d$ contains the information about the identity of the processors at distance $d+1$ that are connected to each processor at distance d from P_i.

Assumption 2 (Links not included) The edge identifiers of the network are *not* stored in the $\mathcal{V}_i^d$ views. In other words, $\mathcal{V}_i^d$ does not contain the information about the identity of the processors at distance $d + 1$.

Let us see how these assumptions influence the information that can be deduced from the view of a processor (*not* knowing the actual radius of the communication graph):

1. If Assumption 1 holds, then the view $\mathcal{V}_i^r$ (where r is the radius of the communication graph of the system) of a node of the communication graph is sufficient to conclude that P_i *is* (or is not) a center. Indeed, if each edge in $\mathcal{V}_i^r$ is connected to exactly two nodes, then P_i is a center, otherwise, it is not.
2. If Assumption 2 holds, then the view $\mathcal{V}_i^r$ is insufficient to conclude that P_i is (or is not) a center, because it is impossible to distinguish the following two cases:
   (a) the radius of the network is r and $\mathcal{V}_i^r$ contains complete knowledge of the network,
   (b) the radius of the network is $r + 1$ and $\mathcal{V}_i^r$ misses information about at least one node.

   Note that if the view of P_i is $\mathcal{V}_i^{r+1}$, where r is the radius of the system, it is possible to conclude that P_i is a center by checking whether there are processors at distance $r + 1$ from itself.

Now we are ready to consider the first task which is the leader election task.

Leader election, task specification Each processor P_i maintains a local output boolean variable $\mathcal{L}_i$ that is set to *true* if the node is elected and *false* otherwise. There is exactly one node P_l with local variable $\mathcal{L}_l$ set to *true*.

Lemma 1 *If Assumption 1 holds, the leader election task is r distance local.*

Proof We first present an impossibility result for the existence of a fault detector for this task in $\mathcal{FD}_{r-1}$ and then present a fault detector in the set of $\mathcal{FD}_r$ for the task.

Let us consider a system $\mathcal{S}$ such that its communication graph $G = (V, E)$ verifies: $\exists x \in V, \exists y \in V, Dist(x, y) = 2r - 1$, where r is the radius of G. For example, a chain graph of $2r$ nodes verifies this property. Then, for any node v in this graph, either (i) $Ball(v, r - 1)$ contains x but not y, or (ii) $Ball(v, r - 1)$ does not contain x. Indeed, it is impossible that $Ball(v, r - 1)$ contains both x and y, since $Dist(x, y) = 2r - 1$ by hypothesis, which is incompatible with the following derivation:

$$\begin{aligned} Dist(x, y) &\leq Dist(x, u) + Dist(y, u) \\ &\leq r - 1 + r - 1 \\ &\leq 2r - 2 \\ &< 2r - 1 \end{aligned}$$

Let us consider the following three configurations:

- c_1 in which x is elected and all other nodes are not,
- c_2 in which y is elected and all other nodes are not, and
- c_3 in which no node is elected.

Suppose there exists a $\mathcal{FD}_{r-1}$ for the leader election task. The result of $\mathcal{FD}_{r-1}$ in c_1 and c_2, must be $true$ at every node, since the configuration satisfies the leader election specification; on the other hand, in c_3, at least one failure detector at a node must respond $false$.

We now consider every node v in c_3. If $Ball(v, r - 1)$ contains x but not y (case *(i)*), then $\mathcal{V}_v^{r-1}$ is the same in configurations c_1 and c_3, thus $\mathcal{FD}_v^{r-1}(v, c_3)$ returns *true*. If $Ball(v, r - 1)$ does not contain x (case *(i)*), then $\mathcal{V}_v^{r-1}$ is the same in configurations c_1 and c_3, thus $\mathcal{FD}_v^{r-1}(v, c_3)$ returns *true*. Hence, $\mathcal{FD}_{r-1}$ responds $true$ in c_3 in which there is no leader.

This completes the proof that the leader election task is not $r - 1$ distance local.

We now present a fault detector in the set $\mathcal{FD}_r$ for the leader election task. By the definition of r-distance local, every view $\mathcal{V}_i^r$ of processor P_i contains the part of configuration including P_i and its neighbors at distance r. According to Assumption 1, $\mathcal{FD}_r(P_i)$ can check whether it knows the entire system, that is whether P_i is a center of the communication graph. If P_i is not a center, $\mathcal{FD}_r(P_i)$ returns $true$ (does not detect a fault). If P_i is a center, $\mathcal{FD}_r(P_i)$ checks in its r-view whether there is exactly one processor P_l with $\mathcal{L}_l = true$. $\mathcal{FD}_r(P_i)$ detects a fault if and only if the above condition is false. □

Next we present a similar proof for the case in which the view does not include the identity of the processors to which a link is connected, unless the link belongs to a path of length $d - 1$ from a processor. Namely, using Assumption 2.

Lemma 2 *If Assumption 2 holds, the leader election task is $(r + 1)$-distance local.*

Proof We first present an impossibility result for the existence of a fault detector for this task in $\mathcal{FD}_r$ and then present a fault detector in the set of $\mathcal{FD}_{r+1}$ for the task. By the definition of r-distance local, every view $\mathcal{V}_i^r$ of processor P_i contains the portion of the configuration that includes P_i and its neighbors at distance r.

Consider the system $\mathcal{S}_1$ of radius r with $(4r - 2)$ processors represented in Fig. 1. Consider a configuration of this system in which all variables $\mathcal{L}_i$ are set to false. At least one fault detector at a processor must detect a fault, it is easy to see that, because all other processors have a partial view of the system, only $\mathcal{FD}_r(A)$ and $\mathcal{FD}_r(B)$ can possibly do that.

Now consider a second system $\mathcal{S}_2$, of $4r$ processors (with the same radius value r), represented in Fig. 1. In $\mathcal{S}_2$, all variables $\mathcal{L}_i$ are set to *false*, except the variable of C, which is set to *true* (so that a unique leader is elected). In both systems, the views at distance r of B are identical. Thus, in both systems, $\mathcal{FD}_r(B)$ must decide the same. Because in system $\mathcal{S}_2$, $\mathcal{FD}_r(B)$ does not detect a fault, it also does not detect a fault in $\mathcal{S}_1$. By considering a dual system, one can conclude that in $\mathcal{S}_1$, $\mathcal{FD}_r(A)$ cannot detect a fault as well.

The contradiction is complete since, in $\mathcal{S}_1$, no processors detect a fault. Hence, there exists no fault detector in $\mathcal{FD}_r$ for the leader election task.

We now present a fault detector in the set $\mathcal{FD}_{r+1}$ for the leader election task. By the definition of $(r + 1)$-distance local every view $\mathcal{V}_i^{r+1}$ of processor P_i contains the part of the configuration including P_i and its neighbors at distance $(r + 1)$. Thus $\mathcal{FD}_{r+1}(P_i)$ can check whether it knows the entire system, that is, whether P_i is a center. If P_i is not a center, $\mathcal{FD}_{r+1}(P_i)$ never detects a fault. If P_i is a center, $\mathcal{FD}_{r+1}(P_i)$ checks in its $(r+1)$-view whether there is exactly one processor P_l with its variable $\mathcal{L}_l$ set to true. $\mathcal{FD}_{r+1}(P_i)$ detects a fault if and only if the above condition is false.

Observation 1 *For every task whose specification is predicated on the global system configuration, there is a fault detector in $\mathcal{FD}_{r+1}(resp.\ \mathcal{FD}_r)$ for this task if Assumption 2 (resp. Assumption 1) holds. Namely, the fault detector that uses the centers to check whether the global predicate is true in their $(r + 1)$-views (resp. r-views) while the other processors return* true.

In the rest of the paper, we will assume that Assumption 2 holds.

Next, we consider the center determination task.

Center determination, task specification Each processor P_i maintains a local output boolean variable $\mathcal{I}_i$ which is *true* if and only if the processor is a center of the system.

Lemma 3 *The center determination task is r-distance local.*

Proof The proof that there is no fault detector in $\mathcal{FD}_{r-1}$ for the center determination task is by presenting systems $\mathcal{S}_1$ and $\mathcal{S}_2$ depicted in Fig. 2. The configuration that we consider for both systems is that where only the $\mathcal{I}$ variable of C is $true$.

System $\mathcal{S}_1$ is in a correct configuration, where centers are correctly identified, so each transient fault detector responds $true$ in this configuration. System $\mathcal{S}_2$ is in an incorrect configuration, because E is incorrectly identified as a non-center.

Now every view of every node in the N branch is the same in systems $\mathcal{S}_1$ and $\mathcal{S}_2$, so each transient fault detector at these nodes responds $true$. Then every node in the W branch in $\mathcal{S}_2$ has the same view as its counterpart node in the N branch, so each transient fault detector at these nodes responds $true$. Finally every node in the E branch in $\mathcal{S}_2$ has the same view in systems $\mathcal{S}_1$ and $\mathcal{S}_2$, so each transient fault detector at these nodes responds $true$. Consequently, no fault is detected in $\mathcal{S}_2$.

Now we construct a fault detector in $\mathcal{FD}_r$ for the center determination task as follows:

1. if $\mathcal{I}_i = true$ (P_i claims it is a center), then the fault detector at P_i responds $false$ iff:
   (a) $\mathcal{V}_i^r$ contains a processor P_j that is a center in $\mathcal{V}_i^r$ and such that $\mathcal{I}_j = false$,
   (b) $\mathcal{V}_i^r$ contains a processor P_j that is not a center in $\mathcal{V}_i^r$ and such that $\mathcal{I}_j = true$.
2. if $\mathcal{I}_i = false$ (P_i claims it is not a center), then the fault detector at P_i responds $false$ iff:
   (a) $\mathcal{V}_i^r$ contains only processors P_k such that $\mathcal{I}_k = false$,
   (b) $\mathcal{V}_i^r$ contains two processors P_j and P_k such that $\mathcal{I}_j = true$ and $Dist(j, k) \geq r + 1$ in $\mathcal{V}_i^r$.

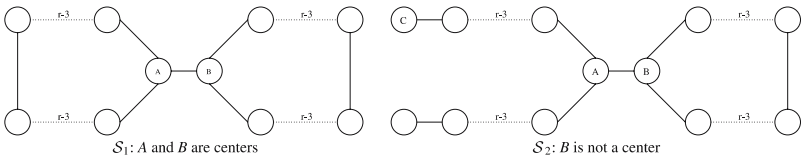


Fig. 1 Leader election

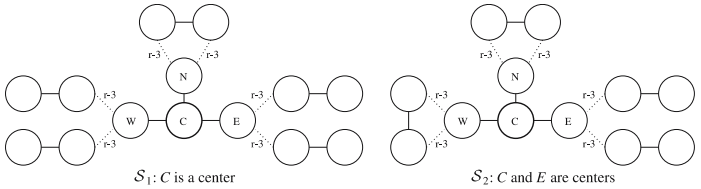


Fig. 2 Center determination

If the system is correct, any center P_i has its $\mathcal{I}_i$ variable set to $true$ and any non-center P_j has its $\mathcal{I}_j$ variable set to $false$. Then none of the centers can detect an inconsistency in its view (that covers the whole network). Any non-center is at most at distance r from a center (by definition of a center) in any view, so no fault detector returns $false$.

If the system is not correct, this means that one of the following two cases occurred: there exists a non center P_j such that $\mathcal{I}_j = true$, or there exists a center P_k such that $\mathcal{I}_k = false$. This leads to three kinds of configurations:

1. There exists a center P_k such that $\mathcal{I}_k = true$: then P_k can detect a fault because its view includes the whole network.
2. There exists no node P_i such that $\mathcal{I}_i = true$, then all fault detectors respond $false$.
3. For any center P_k, $\mathcal{I}_k = false$, and there exists a non center P_j such that $\mathcal{I}_j = true$. Then P_k has the whole network in its view. By definition of a non center, there exists a processor P_i in the network such that $Dist(j, i) \geq r + 1$. So P_k has in its view two nodes at distance more than r and one of them claims it is a center, thus it detects a fault.

It may be surprising that the center determination problem is r-local while the leader election problem is $(r + 1)$-local under the same hypothesis, while the transient fault detector that we presented for leader election checks if the invoking processor is a center of the graph to build its decision.

The reason is the following: to check if the leader election problem is solved using only leader election output variables, a view at distance $r + 1$ is needed; to check if the leader election problem is solved using both the leader election and center output variables, a view at distance r is sufficient. However, the resulting failure detector is not a failure detector for the leader election task, but a failure detector for the center determination *and* leader election task.

Number of nodes Each processor P_i maintains a variable $Number_i$ containing the number of nodes in the system.

Lemma 4 *The number of nodes task is* $(r+1)$*-distance local.*

Proof According to Observation 1, it suffices to prove that there is no fault detector in $\mathcal{FD}_r$ for the number of nodes task. Suppose the contrary holds and consider the system $\mathcal{S}_1$ of $(4r - 2)$ processors and radius r, placed in a configuration in which each processor P_i has its variable $Number_i$ set to $4r$, which is the number of processors in the system $\mathcal{S}_2$, see Fig. 1. Each processor P_i different from A and B (not being a center) has less than $(4r - 4)$ processors in its view at distance r.

It is easy to build a system $\mathcal{S}_1(i)$ with $4r$ processors for each such P_i, with a processor having the same r-view as P_i for any system configuration. For example, see Fig. 3 for nodes at the left of A in $\mathcal{S}_1$. Thus $\mathcal{FD}_r(P_i)$ cannot detect a fault. Hence, only $\mathcal{FD}_r(A)$ and $\mathcal{FD}_r(B)$ (the centers) can possibly detect a fault in system $\mathcal{S}_1$. But A has the same view at distance r in $\mathcal{S}_1$ and $\mathcal{S}_2$. Thus $\mathcal{FD}_r(A)$ does not detect a fault neither in $\mathcal{S}_1$ nor in $\mathcal{S}_2$. For similar reasons, $\mathcal{FD}_r(B)$ does not detect a fault either, a contradiction.

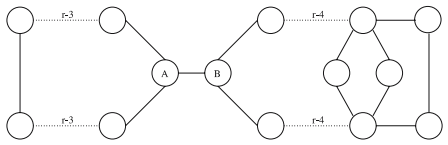


Fig. 3 Number of nodes

3.2 Rooted tree, Hamiltonian circuit, Eulerian circuit and x-tree partition

In this subsection we first present the rooted tree construction task that is $\lceil n/4 \rceil$-distance local.

Rooted tree construction, task specification Each processor P_i maintains a variable $\mathcal{P}_i$ with a pointer to one of its neighbors, chosen to be its parent in the tree. A single processor in the system, P_r which we call the *root*, has a hardwired *nil* value in $\mathcal{P}_r$, while the value of the variables of the other processors define a tree rooted at P_r.

Lemma 5 *The rooted tree construction task is* $\lceil \frac{n}{4} \rceil$*-distance local.*

Proof The proof is by presenting a fault detector in the set of $\mathcal{FD}_{\lceil \frac{n}{4} \rceil}$ and proving impossibility result for the existence of a fault detector for this task that is in $\mathcal{FD}_{\lceil \frac{n}{4} \rceil - 1}$. The fault detector in every non-root processor will check whether a cycle exists or there is evidence that the tree that is connected to the root does not include some processor. The fault detector of the root will check whether the subtree connected to it does not include some processor. Obviously, no fault is detected when the system encodes a tree rooted at P_r. If a processor P_i which is not the root has no parent, this is detected immediately using $\mathcal{V}_i^1$. Then only the case in which a partial tree is rooted at the root and where the other nodes belong to a cycle has to be considered.

A fault detector that uses views of radius $\lceil \frac{n}{4} \rceil$ can detect cycles that are up to $2 \times \lceil \frac{n}{4} \rceil$ nodes long. Consequently, a cycle with $2 \times \lceil \frac{n}{4} \rceil + 1$ processors cannot be detected. Consider a graph that has such a cycle and a subtree rooted at the root node. This subtree contains at most $n - 2 \times \lceil \frac{n}{4} \rceil - 1$ nodes, so its diameter is at most $n - 2 \times \lceil \frac{n}{4} \rceil - 2$. Let B be the set of nodes that are in the subtree connected to the root or are direct neighbors of a node in this subtree. The diameter of the part of the communication graph that connects the nodes in B (the graph in which only the edges connecting nodes of B appear) is at most $n - 2 \times \lceil \frac{n}{4} \rceil$. If a center P_c of the B set has a view at distance $\lceil \frac{n-2\times\lceil \frac{n}{4} \rceil}{2} \rceil$, it can view all nodes in B. Since $\lceil \frac{n-2\times\lceil \frac{n}{4} \rceil}{2} \rceil \leq \lceil \frac{n}{4} \rceil$, all the nodes of B are included in the view of P_c.

Now, if *all* nodes that are direct neighbors of a processor in the subtree (but do not belong to the subtree) have not chosen their parent among the subtree nodes, then the transient fault detector at the center processor can detect a fault appropriately, since it detected processors that are not connected to the root.

To prove that the task is not $(\lceil \frac{n}{4} \rceil - 1) - local$ we consider a system configuration c in which a chain of $\lceil n/2 \rceil$ processors are connected to the root and the rest of the processors form a cycle. In such a configuration, at least one transient fault detector at a processor must detect a fault; we prove that every such possible detector will also detect a fault in a configuration that encodes a tree.

Assume that the communication graph of the system in which a cycle exists includes in addition to the above chain and cycle a (non-tree) edge between a processor in the chain and a processor in the cycle (thus it is possible to have a tree spanning the graph).

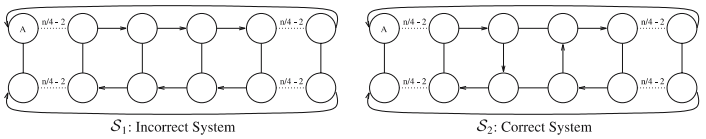
Assume that a fault detector $\mathcal{FD}_d(P_i)$ at processor P_i in the cycle identifies a fault in c (*i.e.*, $\mathcal{FD}_d(P_i, c)$ returns *false*). Then there exists at least one processor P_j in the cycle that is not included in $\mathcal{V}_i^d$ (where $d = (\lceil \frac{n}{4} \rceil - 1)$). There exists a configuration c' that includes an additional edge from P_j to the root. P_j may have chosen this edge to be a tree edge and therefore no cycle exists in c', but P_i will still have the same view $\mathcal{V}_i^d$ in c' and therefore $\mathcal{FD}_d(P_i)$ will detect a fault in c' (*i.e.*, $\mathcal{FD}_d(P_i, c)$ returns *false*).

If the fault detector that indicates a fault is at a processor P_i on the chain that is connected to the root, then there is at least one processor, P_j, on the chain and one processor, P_k, on the cycle that are not included in $\mathcal{V}_i^d$. Thus, $\mathcal{FD}_d(P_i)$ will indicate a fault when the system is in another configuration, one in which there are no cycles, since P_j and P_k are connected by an edge and P_j chooses P_k as its parent.

Hamiltonian circuit construction (*in a Hamiltonian system*) Each processor P_i has distinguished one incoming ($in(i)$) and one outgoing ($out(i)$) edge, that globally induce a Hamiltonian circuit.

Lemma 6 *The Hamiltonian circuit construction task is* $\lceil n/4 \rceil$*- distance local* (*n is the number of nodes*).

Proof Note first that a fault detector at a processor that has not distinguished exactly one incoming and one outgoing edge can detect a fault. If the structure globally induced by the variables $in(i)$ and $out(i)$ is not a Hamiltonian circuit, it has necessarily several circuits covering the system (each processor is in one circuit). But one of the circuits has no more than $n/2$ processors and each processor in this circuit has the entire circuit in its view at distance $\lceil n/4 \rceil$. Thus a fault can be detected by any fault detector in these processors.



$\mathcal{S}_1$: Incorrect System $\mathcal{S}_2$: Correct System

Fig. 4 Hamiltonian circuit

Now, for proving that the task has no fault detector in $\mathcal{FD}_{\lceil n/4 \rceil - 1}$, consider the two systems $\mathcal{S}_1$ and $\mathcal{S}_2$ presented in Fig. 4.

In both $\mathcal{S}_1$ and $\mathcal{S}_2$, processor A has the same $\mathcal{V}^{\lceil n/4 \rceil - 1}$, thus $\mathcal{FD}_{\lceil n/4 \rceil - 1}(A)$ cannot detect a fault in $\mathcal{S}_1$. Since the system is symmetric, no fault detector can detect a fault in $\mathcal{S}_1$, where no Hamiltonian circuit is built.

Eulerian circuit construction (in an Eulerian system) Each processor P_i has $\delta_i/2$ variables (δ_i being the degree of node i in which P_i resides). Each variable l_j contains a pair of indices of edges attached to P_i, such that these edges appear one immediately after the other in the Eulerian circuit.

Lemma 7 *The Eulerian circuit construction task is* $\lceil n/4 \rceil$*-distance local (n is the number of nodes).*

Proof Note first that a fault detector at a processor P_i having an edge appearing in two different l_j can detect a fault.

Then to prove that there is exactly one circuit, we follow the same proof as in the Hamiltonian circuit construction proof, considering the two systems in Figs. 5 and 6.

Processors A and B have the same $\mathcal{V}^{\lceil n/4 \rceil - 1}$ in both systems, thus neither of their fault detectors can detect a fault in the first one.

Since the system is symmetric, no transient fault detector can detect a fault in the first system, where no Eulerian circuit is built.

Next we show that there is a class of tasks that requires a fault detector of radius i (where i is not related to n) for every integer i.

x-Tree partition, task specification Each processor P_i maintains a boolean variable $\mathcal{B}_{ij}$ for each of its attached links (i, j). For any two neighbor nodes i and j, $\mathcal{B}_{ij} = \mathcal{B}_{ji}$. The edges with $\mathcal{B}_{ij} = true$ are called the *border* edges. If the border edges are disconnected then the Tree is partitioned into connected components with a diameter of no more than x, where x is a positive integer smaller than n (See [16]).

Lemma 8 *The x-Tree partition is* $\lfloor \frac{x}{2} \rfloor$*-distance local.*

Proof The proof is by proving there exists no fault detector for the x-Tree partition task in the $\mathcal{FD}_{\lfloor \frac{x}{2} \rfloor - 1}$ set and by showing the existence of a fault detector in the $\mathcal{FD}_{\lfloor \frac{x}{2} \rfloor}$ set.

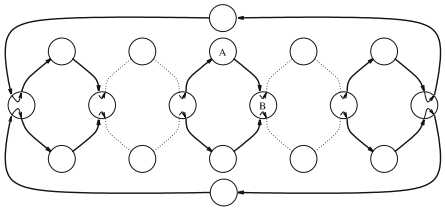


Fig. 5 No Eulerian circuit

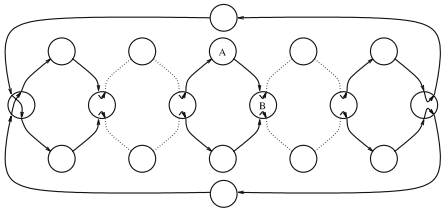


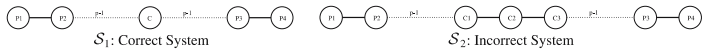
Fig. 6 Eulerian circuit

The impossibility result is found by considering the two systems $\mathcal{S}_1$ (which is correct) and $\mathcal{S}_2$ (which is incorrect) presented in Fig. 7 and by showing that if a fault detector responds $true$ in the first system, it also responds $true$ in the second system. In the two systems, thicker edges denote borders.

We consider $p = \lfloor \frac{x}{2} \rfloor$ (so that $x = 2 \times p$ if x is even and $x = 2 \times p + 1$ if x is odd). Suppose that in $\mathcal{S}_1$, the processor C has a view at distance $p - 1$. Then $\mathcal{FD}_{p-1}(C)$ does not see processors P_2 and P_3. Since the first system is correct, no transient fault detector detects a fault. Now in $\mathcal{S}_2$, processors C_1, C_2 and C_3 have the same view as processor C in $\mathcal{S}_1$. All other nodes in $\mathcal{S}_2$ have the same view as in $\mathcal{S}_1$, so neither of the fault detectors at these processors can detect a fault. As a result, no detector detects a fault in $\mathcal{S}_2$.

Now we construct a transient fault detector for the x-Tree partition task that is in $\mathcal{FD}_{\lfloor \frac{x}{2} \rfloor}$ at processor P_i. This fault detector responds $true$ except in the following two cases:

1. $\mathcal{V}_i^p$ contains j and k such that $\mathcal{B}_{jk} \neq \mathcal{B}_{kj}$.
2. $\mathcal{V}_i^p$ contains x nodes $i_1, i_2, \ldots, i_x$ such that for any j in $[1 \ldots x]$, $\mathcal{B}_{i_j i_{j+1}} = false$.



$\mathcal{S}_1$: Correct System $\mathcal{S}_2$: Incorrect System

Fig. 7 x-Tree partition

If the configuration is correct, then for any two nodes j and k, we have $\mathcal{B}_{jk} = \mathcal{B}_{kj}$, so rule 1 does not apply. Also, since any chain such that any process P_i in the chain has one of its $\mathcal{B}_{i*}$ set to $false$ is of length less than the diameter of a connected component, it is also less than x, thus rule 2 does not apply either. As a consequence, in a correct configuration, the fault detector responds $true$.

If the configuration is not correct, it means that either two neighboring nodes j and k do not agree on a common border edge, or that there exists a connected component of diameter strictly greater than x. The first case is trivially solved by rule 1. The second case can be rewritten as there exists a chain of $x + 1$ processors $P_{i_1}, P_{i_2}, \ldots, P_{i_{x+1}}$ such that for any j in $[1 \ldots x]$, $\mathcal{B}_{i_j i_{j+1}} = false$. By definition, a center node c of this chain has in its view $\mathcal{V}_c^p$ all processors $P_{i_1}, P_{i_2}, \ldots, P_{i_{x+1}}$, and by rule 2, $\mathcal{FD}_p(c)$ can detect a fault.

3.3 Maximum independent set, coloring, and topology update

Maximum independent set, task specification Each processor P_i maintains a local boolean variable $\mathcal{IS}_i$. No two neighbors may have their variable set to *true*. In addition, every processor P_i with $\mathcal{IS}_i = false$ has at least one neighbor P_j with $\mathcal{IS}_j = true$.

Lemma 9 *The maximum independent set task is* 1*-distance local.*

Proof The proof is by presenting a fault detector in the set of $\mathcal{FD}_1$ and proving impossibility result for the existence of a fault detector for this task in $\mathcal{FD}_0$.

By the definition of 1-distance local, $\mathcal{FD}_1(P_i)$ can verify that: if $\mathcal{IS}_i = true$, then $\forall j \in Neighbors_i$, $\mathcal{IS}_i \neq \mathcal{IS}_j$, and if $\mathcal{IS}_i = false$, then $\exists j \in Neighbors_i$, $\mathcal{IS}_i \neq \mathcal{IS}_j$. The fault detector at P_i will indicate the occurrence of a fault in case any of the above properties does not hold. The above test ensures that the value of all the $\mathcal{IS}$ variables constructs a maximum independent set.

By the definition of 0-distance local, no fault is detected in a configuration in which the $\mathcal{IS}_i$ variable of every processor P_i holds *true*.

A similar proof holds for the coloring task that we now present.

Coloring, task specification Each processor P_i maintains a variable $\mathcal{C}_i$ representing its color. In addition, for every two neighboring processors P_i and P_j it holds that $\mathcal{C}_i \neq \mathcal{C}_j$.

Lemma 10 *The coloring task is* 1*-distance local.*

Proof The proof is by presenting a fault detector in the set of $\mathcal{FD}_1$ and proving the impossibility result for the existence of a fault detector for this task in $\mathcal{FD}_0$.

By the definition of 1-distance local, $\mathcal{FD}_1(P_i)$ can verify that:

$$\forall j \in Neighbors_i, \mathcal{C}_i \neq \mathcal{C}_j,$$

The fault detector at P_i notices the occurrence of a fault in case the above property does not hold.

By the definition of 0-distance local, no fault is detected in a configuration in which the $\mathcal{C}_i$ variable of every processor P_i holds the same value $\mathcal{C}$.

Topology update, task specification Each processor P_i maintains a local variable $\mathcal{T}_i$, containing the representation of the communication graph, say by using a neighboring matrix or the list of the communication graph edges.

Lemma 11 *The topology update task is* 1*-distance local.*

Proof The proof is by presenting a fault detector in the set of $\mathcal{FD}_1$ and proving the (obvious) impossibility result for the existence of a fault detector for this task in $\mathcal{FD}_0$.

By the definition of 1-distance local, $\mathcal{FD}_1(P_i)$ can verify that $\mathcal{T}_i = \mathcal{T}_j$ for every neighboring processor P_j. The fault detector at P_i notices the occurrence of a fault in case there exists a neighbor for which the above equality does not hold. The above test ensures that the value of all the $\mathcal{T}$ variables is the same. In addition, the fault detector at P_i checks whether the local topology of P_i (that is included in $\mathcal{V}_i^1$) appears correctly in $\mathcal{T}_i$. This test ensures that the (common identical) value of $\mathcal{T}$ is correct, since every processor identified its local topology in $\mathcal{T}$.

By the definition of 0-distance local, no fault is detected in a configuration in which the $\mathcal{T}_i$ variable of every processor P_i holds the local topology of P_i i.e., P_i and its neighbors, without the rest (non empty portion) of the system.

4 Fault detectors for non-silent tasks

In this section we consider the set of non-silent tasks. Unlike the previous section that considered fault detectors for asynchronous (as well as synchronous) systems, in this section we consider fault detectors for synchronous systems. We present tasks that are s-history local, with $s > 1$. Here, s defines the

size of the history $\mathcal{V}_i^d[1 \ldots s]$ of each processor P_i. The system is synchronous and each view in $\mathcal{V}_i^d[1 \ldots s]$ is related to a different time. This array is thereafter referred as the *local history* of processor P_i. Each $\mathcal{V}_i^d$ is a view on every component of the system up to distance d from P_i.

We now list several non-silent tasks, present their specification, and identify the minimal history required for their fault detectors. We start with a trivial bounded privilege task.

4.1 Bounded privilege

Bounded privilege, task specification Each processor P_i maintains a local boolean variable $\mathcal{P}riv_i$. For each processor P_i $\mathcal{P}riv_i$ is set to true exactly once (another variant is at least once) in every c synchronous steps ($c \geq 2$).

Lemma 12 *The bounded privilege task is* 0*-distance local, c-history local.*

Proof A local history of $c-1$ views such that in each view the output variable $\mathcal{P}riv_i$ is false does not give an indication on task violation. On the other hand it is clear that a local history of c views is sufficient for fault detection.

4.2 Bounded privilege dining philosophers

Bounded privilege dining philosophers, task specification Each processor P_i maintains a local boolean variable $\mathcal{P}riv_i$. For each processor P_i $\mathcal{P}riv_i$ is set to true at least once in every c synchronous steps ($c \geq 2$). In addition, for every two neighboring processors, P_i and P_j, if $\mathcal{P}riv_i = true$ then

$$\mathcal{P}riv_j = false$$

Lemma 13 *The bounded privilege dining philosophers task is* 1*-distance local, c-history local.*

Proof First we prove that there exists no 0-distance local fault detector for the bounded privilege dining philosophers task.

If there existed such a fault detector, it could not possibly detect that two neighbors P_i and P_j have $\mathcal{P}riv_i = \mathcal{P}riv_j = true$ simultaneously.

Then we prove that there exists no $(c-1)$-history local fault detector for the bounded privilege dining philosophers task.

If $c = 2$, a $(c-1)$-history local fault detector would use only the current view to detect a fault. Then consider two possible runs e_1 and e_2 of a system consisting of two processors P_1 and P_2, as depicted in Fig. 8. In run e_1, $\mathcal{P}riv_1$ and $\mathcal{P}riv_2$ are alternatively set to $true$ every two time units, so this run is correct. Obviously, run e_2 is incorrect since $\mathcal{P}riv_2$ is never set to $true$. The 1-history in any configuration C_n of e_2 is the same as the 1-history in configuration $C_{2\times n}$ of e_1. Since the fault detector must respond $true$ in any configuration of e_1, it must respond $true$ in any configuration of e_2, which is an incorrect run, thus this fault detector is unable to detect a fault.

If $c \geq 3$, then consider a system containing two processors P_1 and P_2 and consider 2 runs of this system as depicted in Fig. 9. Runs e_1 and e_2 consist in a repeated pattern of size c. In runs e_1 and e_2, processors P_1 and P_2 have their $\mathcal{P}riv$ variable set to $true$ every c configurations. In addition, in each configuration of these runs, we never have $\mathcal{P}riv_1 = \mathcal{P}riv_2 = true$. Consequently, runs e_1 and e_2 are correct, so the fault detectors at P_1 and P_2 respond $true$ in any of the configurations of these runs. Now consider run e_f of the same system as depicted in Fig. 10. Run e_f consists of a repeated pattern of size $2 \times c$. In this pattern, the first c configurations are the same as in e_1, while the next c configurations are the same as in e_2. In run e_f, $\mathcal{P}riv_1$ is set to $true$ every c configurations, but $\mathcal{P}riv_2$ is alternatively set to true every $c+1$ configurations, and every $c-1$ configurations so run e_f is incorrect.

In configurations C_1 to C_{c+1} of run e_f, the $(c-1)$-history of both P_1 and P_2 is the same as in configuration C_1 to C_{c+1} of run e_1, so both fault detectors respond $true$ in these configurations of e_f. In configurations C_{c+2} to $C_{2\times c+1}$ of run e_f, the $(c-1)$-history of both P_1 and P_2 is the same as in configuration C_{c+2} to $C_{2\times c+1}$ of run e_2, so both fault detectors respond $true$ in these configurations of e_f.

Using the same reasoning, the $(c-1)$-histories of P_1 and P_2 in configurations $C_{(k-1)\times c+2}$ to $C_{k\times c+1}$ of run e_f are the same as those in run e_1 if k is odd and those in run e_2 if k is even. Since the histories in run e_f could be those of correct runs, no fault is detected in any configuration of e_f, which is an incorrect run.

On the other hand, it is clear that a local history of c views is sufficient for detection of the first predicate of the task specification. In addition, to ensure that no two processors are privileged simultaneously, a view of diameter 1 is sufficient.

4.3 Deterministic non-interactive tasks

In a synchronous run (assuming a synchronous scheduler that activates all enabled processors at each synchronous step) of a non-randomized, non-interactive, bounded space algorithm, some configurations must be reached more than once, and thus the system must repeat its actions infinitely often, in every infinite run. Therefore, there is a bounded *repetition pattern* that can be identified, where the actions of the processors are repeated.

Each processor can have a local history that includes all the views of the repetition pattern in the order they should be repeated. The processors repeatedly send to their neighbors their local history and detect inconsistency if two views that are related to the same time do not agree, or the current value of the output variables are not correct. The distance of the views is a function of the task. Note that, in fact, when the distance of views is equal to the diameter of the system, the above fault detectors may serve as an implementation of the task.

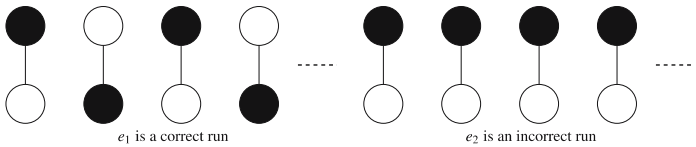


Fig. 8 P_1 and P_2 should be privileged every 2 configurations

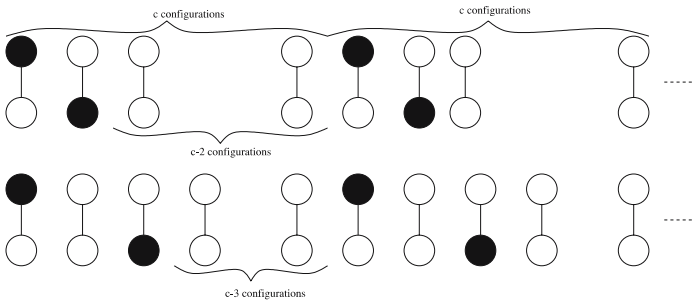


Fig. 9 Runs e_1 and e_2 are correct runs

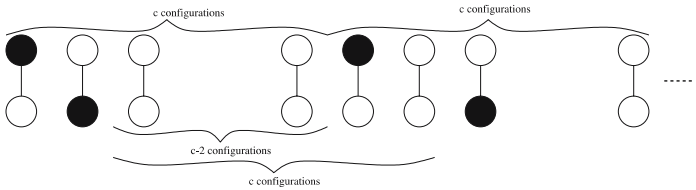


Fig. 10 Run e_f is incorrect

4.4 Fair privilege

Fair privilege, task specification Each processor P_i maintains a local boolean variable $\mathcal{P}riv_i$. For each processor P_i $\mathcal{P}riv_i$ is set to true infinitely often in every infinite synchronous run.

Lemma 14 *For any integers d and l, there exists no d-distance local, l-history local transient fault detector for the fair privilege task.*

Proof Assume that there exist two integers d and l such that there exists a d-distance local, l-history local transient fault detector for the fair privilege task.

Consider a system where, in each synchronous run, processors get the same privilege status (for any processor P_i and any configuration c, all $\mathcal{P}riv_i$ variables are equal). A run of this system is given by the successive values of $\mathcal{P}riv_i$, the variable of a single processor P_i. We consider two runs R_1 (which is correct) and R_2 (which is incorrect) and prove that if the l-history local transient fault detector responds $true$ in R_1, it also responds $true$ in R_2, and thus does not detect the faulty run.

$$R_1 = \underbrace{false, \ldots, false}_{l \text{ times}}, true, true, \ldots$$

$$R_2 = \underbrace{false, \ldots, false}_{l \text{ times}}, false, false, \ldots$$

In run R_1, the $\mathcal{P}riv_i$ variable is set to $true$ infinitely often, so this run is correct. Therefore the transient fault detector at processor P_i must return $true$ in any configuration.

In run R_2, the $\mathcal{P}riv_i$ variable is never set to $true$, so this run is incorrect. Therefore the transient fault detector must respond $false$ in at least one configuration at node P_i.

Configurations c_0 to c_{l-1} are equal in R_1 and R_2, so the histories $\mathcal{V}_i^0[1 \ldots l]$ are equal in those configurations. Since the transient fault detector responded $true$ in configurations $c_0 \ldots c_{l-1}$ in R_1 (which is correct), it also responds $true$ in configurations $c_0 \ldots c_{l-1}$ in R_2. Note that in configuration c_{l-1} of either R_1 or R_2, the variable $\mathcal{V}_i^0[1 \ldots l]$ equals

$$\underbrace{false, \ldots, false}_{l \text{ times}}$$

Now consider configuration c_k of R_2, with $k \geq l$. In configuration c_k, $\mathcal{V}_i^0[1 \ldots l]$ equals $\underbrace{false, \ldots, false}_{l \text{ times}}$, thus the transient fault detector must respond $true$ in configuration c_k.

Since for any configuration of R_2, the l-history local fault detector responds $true$, it does not detect any fault in R_2, which is an incorrect run.

5 Transient fault detectors for algorithms

Unlike the case of fault detectors for tasks, the fault detectors for algorithms (implementations of tasks) may use the entire state of the processors (and not only the output that is defined by the task). For example, in an implementation of the spanning tree construction in which every processor has a variable with the distance from the root, the fault detector may use the distance variable to detect inconsistency: if every processor has a distance that is greater than one from its parent distance, and the root has no parent, then the system is in a consistent state.

A monitoring technique that can be used as a fault detector is presented in [1]. The monitoring technique can detect inconsistency of every on-line or off-line *algorithm*. Since the monitoring technique is universal, it is possible to design more efficient (in terms of memory) fault detectors for specific sets of algorithms.

Hierarchy for fault detectors of algorithms can be defined analogously to the definition of the fault detectors for tasks. In fact, the predicates that were previously defined for the task can be replaced by new predicates for invariants that are preserved by the algorithm in every execution. The choice of the algorithm that implements a task influences the fitting fault detector. For instance, one may suggest using a topology update algorithm to implement the above silent tasks. A topology update algorithm provides each processor with information concerning the entire system, thus every processor may perform *local* computations using this information to elect a leader, to elect an identifier, or to count the nodes in the system. Clearly, the above choice of implementation results in using much more memory than other possible implementations. On the other hand, it is possible to monitor the consistency of this particular implementation by a fault detector in $\mathcal{FD}_1$.

6 Implementing transient fault detectors

In this section, we give a possible implementation for using and maintaining the local variables of the fault detectors, namely the local view (for silent algorithms) and local history (for non-silent algorithms) variables.

6.1 Updating the local views

The policy for updating the processor views is the following. Each processor P_i communicates (repeatedly in asynchronous systems, at each pulse in synchronous systems) with each of its neighbors, P_j—the portion of $\mathcal{V}_i^d$ that is shared with $\mathcal{V}_j^d$. In other words, P_i does not communicate to P_j the view on the system components that are of distance $d+1$ from P_j (according to the communication graph portion in $\mathcal{V}_i^d$). When P_i receives the view $\mathcal{V}_j^d$ from its neighbor P_j, P_i checks whether the shared portions of $\mathcal{V}_i^d$ and $\mathcal{V}_j^d$ agree. P_i outputs a fault indication if these portions do not agree.

It is easy to see that the above test ensures that every processor has the right view of the components up to distance d from itself. Assume that the view of P_i is not correct concerning the variable of some processor P_k. Let P_i, P_{j1}, P_{j2}, …, P_k be the processors along a shortest path (of length not greater than d) from P_i to P_k. Let P_{jl} be the first processor in this path for which $\mathcal{V}_{jl}$ and $\mathcal{V}_i$ hold non-equal values for a variable of P_k. Note that there must exist such a processor since P_i and P_k hold different values for the variables of P_k. Clearly, P_{jl} and $P_{j(l-1)}$ identify the inconsistency.

6.2 Updating the local histories

We define the updating policy of the local histories only for synchronous systems. In each synchronous step the last view $\mathcal{V}_i^d[s]$ becomes the first view, each other view $\mathcal{V}_i^d[j]$ is copied into $\mathcal{V}_i^d[j+1]$, $1 \leq j < s$.

6.3 Towards self-stabilization

The last issue in implementing the fault detector is the action taken upon inconsistency detection. Although it is out of the scope of the paper, we mention the reset (e.g., [20,4]) and repair operations (e.g., [1,21]), both should result in a consistent state of the system and the fault detector. The fault detector is not activated until the reset or the repair actions terminate. In particular, to collect a new view up to distance d, the update algorithm [10,13] can be used for d rounds (using a synchronizer, in the case of asynchronous system) and only then will the fault detector be activated.

7 Concluding remarks

In this paper, we investigated the amount of resources required for implementing transient fault detectors for tasks and algorithms. We presented a hierarchy of transient fault detectors that detect the occurrence of faults in a *single* asynchronous round (in an asynchronous system) or a single synchronous round (in a synchronous system). The benefits of our approach are twofold:

1. It gives the task implementer a formal framework to analyze the expected "cost" of the task, or even the implementation of the task (considered as a refined task), in terms of resources that are required for transient fault detection.
2. It provides a measure of the informal notion of locality from a transient fault detection point of view (i.e., two tasks or two task implementations can now be compared based on the locality of their respective transient fault detectors). This is complementary to the usual notion of locality in distributed computing [23] which relates to the amount of resources needed to *solve* the task.

We suggest two interesting open problems:

1. While this paper concentrated on defining the notion of locality based on the effort needed to *detect* transient faults, an analogous definition based on the effort for *repairing* would be of great interest.
2. The transient fault detectors that we present in this paper could actually be used to detect arbitrary deviation from the specification, not just transient faults. However, the possible transient fault detector implementation that we give in this paper may only be subject to transient faults. Designing detectors that can cope with arbitrary (i.e., malicious) failures would turn our solution into a universal *predicate detector* mechanism.

Acknowledgements We are grateful to the anonymous reviewers who enabled us to improve the contents and presentation of this paper.

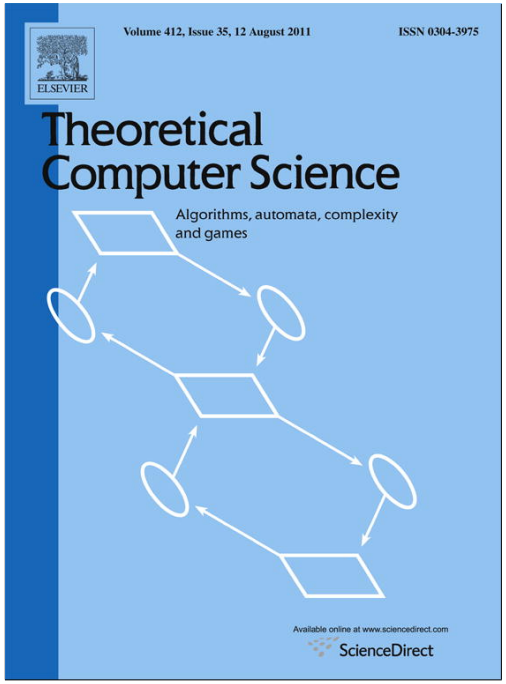
References

1. Afek, Y., Dolev, S.: Local stabilizer. J. Parallel Distrib. Comput. **62**(5), 745–765 (2002)
2. Afek, Y., Kutten, S., Yung, M.: Memory-efficient self stabilizing protocols for general networks. In: van Leeuwen, J., Santoro, N. (eds.) WDAG, Lecture Notes in Computer Science, vol. 486, pp. 15–28. Springer, Berlin (1990)
3. Awerbuch, B., Patt-Shamir, B., Varghese, G.: Self-stabilization by local checking and correction (extended abstract). In: FOCS, pp. 268–277. IEEE (1991)
4. Awerbuch, B., Patt-Shamir, B., Varghese, G., Dolev, S.: Self-stabilization by local checking and global reset (extended abstract). In: Tel, G., Vitányi, P.M.B. (eds.) WDAG, Lecture Notes in Computer Science, vol. 857, pp. 326–339. Springer, Berlin (1994)
5. Burns, J.E., Gouda, M.G., Miller, R.E.: Stabilization and pseudo-stabilization. Distrib. Comput. **7**(1), 35–42 (1993)
6. Chandra, T.D., Toueg, S.: Unreliable failure detectors for reliable distributed systems. J. ACM **43**(2), 225–267 (1996)
7. Delaët, S., Ducourthial, B., Tixeuil, S.: Self-stabilization with r-operators revisited. J. Aerosp. Comput. Inf. Commun. (2006)
8. Delaët, S., Tixeuil, S.: Tolerating transient and intermittent failures. J. Parallel Distrib. Comput. **62**(5), 961–981 (2002)
9. Dijkstra, E.W.: Self-stabilizing systems in spite of distributed control. Commun. ACM **17**(11), 643–644 (1974)
10. Dolev, S.: Self-stabilizing routing and related protocol. J. Parallel Distrib. Comput. **42**(2), 122–127 (1997)
11. Dolev, S.: Self-stabilization. MIT Press, Cambridge (2000)
12. Dolev, S., Gouda, M.G., Schneider, M.: Memory requirements for silent stabilization. Acta Inf. **36**(6), 447–462 (1999)
13. Dolev, S., Herman, T.: Superstabilizing protocols for dynamic distributed systems. Chicago J. Theor. Comput. Sci. **1997** (1997)
14. Dolev, S., Israeli, A., Moran, S.: Self-stabilization of dynamic systems assuming only read/write atomicity. Distrib. Comput. **7**(1), 3–16 (1993)
15. Dolev, S., Israeli, A., Moran, S.: Analyzing expected time by scheduler-luck games. IEEE Trans. Softw. Eng. **21**(5), 429–439 (1995)
16. Dolev, S., Kranakis, E., Krizanc, D., Peleg, D.: Bubbles: adaptive routing scheme for high-speed dynamic networks (extended abstract). In: STOC, pp. 528–537. ACM (1995)
17. Ducourthial, B., Tixeuil, S.: Self-stabilization with r-operators. Distrib. Comput. **14**(3), 147–162 (2001)
18. Ducourthial, B., Tixeuil, S.: Self-stabilization with path algebra. Theor. Comput. Sci. **293**(1), 219–236 (2003). Extended abstract in Sirrocco 2000
19. Ghosh, S., Gupta, A., Herman, T., Pemmaraju, S.V.: Fault-containing self-stabilizing algorithms. In: PODC, pp. 45–54 (1996)
20. Katz, S., Perry, K.J.: Self-stabilizing extensions for message-passing systems. Distribut. Comput. **7**(1), 17–26 (1993)
21. Kutten, S., Patt-Shamir, B.: Time-adaptive self stabilization. In: PODC, pp. 149–158 (1997)
22. Lin, C., Simon, J.: Observing self-stabilization. In: PODC92 Proceedings of the 11th annual ACM symposium on principles of distributed computing, pp. 113–123 (1992)
23. Peleg, D.: Distributed computing: a locality-sensitive approach. SIAM Monogr. Discr. Math. Appl. (2000)

A.2 Démasquer les planqués

Deterministic Secure Positioning in Wireless Sensor Networks. Sylvie Delaët, Partha Sarathi Mandal, Mariusz Rokicki, and Sébastien Tixeuil. *Theoretical Computer Science (TCS)*, 412(35) : 4471-4481, 2011.

– *Résumé :* Le problème de la vérification des positions est une brique de base importante pour un large sous-ensemble d'applications dans les réseaux de capteurs sans fil (WSN). En effet, la performance globale du WSN décroît significativement quand des nœuds dévient de leur comportement attendu et rapportent des informations de localisation erronées de manière à simuler une autre position. Dans cet article, nous proposons le premier algorithme réparti pour l'identification fiable de capteurs malveillants dans un WSN. Notre approche n'est pas basée sur un sous-ensemble de nœuds fiables qui coopèrent et dont le comportement est exemplaire. Ainsi, tout sous-ensemble de nœuds peut essayer de simuler une fausse position. Comme dans les approches précédentes, notre protocole est basé sur des techniques d'évaluation de la distance développées pour les WSN. D'un côté, nous montrons que la technique de mesure de la puissance du signal reçue (RSS) est utilisée, notre protocole peut tolérer $n/2 - 2$ capteurs malveillants. Quand la technique de temps de traversée (ToF) est utilisée, notre protocole tolère au plus $n/2 - 3$ capteurs malveillants. D'un autre côté, nous montrons qu'aucun protocole déterministe ne peut identifier des capteurs malveillants si leur nombre est $n/2 - 1$. Notre schéma est donc presque optimal suivant le critère du nombre de capteurs malveillants tolérés. Nous discutons également de l'utilisation de nos techniques dans un modèle où un sous-ensemble de capteurs est considéré comme fiable et connu de tous. De manière plus spécifique, nos résultats peuvent être utilisés pour minimiser le nombre de capteurs fiables qui sont nécessaires à contrecarrer les actions des capteurs malveillants.



Theoretical Computer Science 412 (2011) 4471–4481

Contents lists available at ScienceDirect

Theoretical Computer Science

journal homepage: www.elsevier.com/locate/tcs

Deterministic secure positioning in wireless sensor networks

Sylvie Delaët [a], Partha Sarathi Mandal [b], Mariusz A. Rokicki [c], Sébastien Tixeuil [d,*]

[a] Univ. Paris Sud, France
[b] Indian Institute of Technology, Guwahati, India
[c] University of Liverpool, United Kingdom
[d] UPMC Sorbonne Universités, Institut Universitaire de France, France

ARTICLE INFO

Article history:
Received 22 April 2010
Received in revised form 27 January 2011
Accepted 5 April 2011
Communicated by R. Klasing

Keywords:
Wireless sensor network
Secure positioning
Distributed protocol

ABSTRACT

The position verification problem is an important building block for a large subset of wireless sensor network (WSN) applications. Indeed, the performance of the WSN degrades significantly when misbehaving nodes report false location information in order to fake their actual position. In this paper we propose the first deterministic distributed protocol for an accurate identification of faking sensors in a WSN. Our scheme does *not* rely on a subset of *trusted* nodes that cooperate and are not allowed to misbehave. Thus, any subset of nodes is allowed to try faking its position. As in previous approaches, our protocol is based on distance evaluation techniques developed for WSN.

On the positive side, we show that when the received signal strength (RSS) technique is used, our protocol handles at most $\lfloor \frac{n}{2} \rfloor - 2$ faking sensors. When the time of flight (ToF) technique is used, our protocol manages at most $\lfloor \frac{n}{2} \rfloor - 3$ misbehaving sensors. On the negative side, we prove that no deterministic protocol can identify faking sensors if their number is $\lceil \frac{n}{2} \rceil - 1$. Thus, our scheme is almost optimal with respect to the number of faking sensors.

We discuss application of our technique in the trusted sensor model. More specifically, our results can be used to minimize the number of trusted sensors that are needed to defeat faking ones.

1. Introduction

The position verification problem is an important building block for a large subset of wireless sensor network (WSN) applications. For example, environment and habitat monitoring [21], surveillance and tracking for military [8], and geographic routing [13], require an accurate position estimation of the network nodes. Most of existing position verification protocols rely on distance evaluation techniques (*e.g.* [1,7,9,19,23,24]). Received signal strength (RSS) [1] and time of flight (ToF) [7] techniques are relatively easy to implement yet very precise (one or two meters). In the RSS technique, a receiving sensor estimates the distance of the sender on the basis of sending and receiving signal strengths. In the ToF technique, the sensor estimates distance based on message delay and radio signal propagation time. Position verification using the aforementioned distance estimation techniques is relatively straightforward provided that *all* sensors cooperate.

In the *secure* positioning problem we can distinguish two sets of nodes: the set of *correct* nodes and the set of *faking* (or *cheating*) nodes. The goal of faking nodes is to mislead correct nodes about their real positions. More specifically, faking nodes cooperate and corrupt the position verification protocol to convince the correct nodes about their faked positions.

* Corresponding author. Tel.: +33 144278764.
E-mail address: Sebastien.Tixeuil@lip6.fr (S. Tixeuil).

0304-3975/$ – see front matter © 2011 Elsevier B.V. All rights reserved.
doi:10.1016/j.tcs.2011.04.010

This problem can be also seen as a variation of the Byzantine agreement problem [14] in which the correct nodes have to decide which nodes are faking their positions. Initially correct nodes do not know which nodes are correct and which nodes are faking. Faking nodes come to play in the WSN in a natural way due to several factors: a sensor node may malfunction, inaccurate position (coordinates) estimation [1,7] may induce incorrect information, or attackers could use fake positions to take down the network.

Related work. Most methods for secure positioning [3,4,15,16] existing in the literature rely on a fixed set of *trusted* entities (or *verifiers*) and distance estimation techniques to filter out faking nodes. We refer to this model as the *trusted sensor* (or *TS*) model. In this model, faking nodes may use attacks not available to regular nodes, such as radio signal jamming or using directional antennas, which permit to implement *e.g.* wormhole attack [11] and Sybil attack [5]. Lazos and Poovendran [15] present a secure range-independent localization scheme, where each sensor computes its position based on received beacon messages from locators. Sensors compute the center of gravity of beacons' intersection region, and the computed location becomes the estimated location of the sensor. Probabilistic analysis of the protocol demonstrates that it is resilient to wormhole and Sybil attacks, with high probability. Lazos et al. [16] further refine this scheme with multilateration to reduce the number of required locators, while maintaining probabilistic guarantees. The TS model was considered by Capkun and Hubaux [4] and Capkun et al. [3]. In [4] the authors present a protocol that relies on a distance bounding technique. This technique was proposed by Brands and Chaum [2]. Each sensor v measures its distance to a (potential) faking sensor u based on its message round-trip delay and radio signal propagation time, thus enabling the faking node u only to *enlarge* the distance to v. Then, if the faking node is located inside the triangle formed by verifiers and its faked position is also located within the triangle, then at least one of the three verifiers detects an inconsistency. The protocol presented in [3] relies on a set of hidden verifiers. Each verifier v measures the arrival time t_v of the (potential) faking node transmission. Verifiers exchange all such arrival times and check consistency of the declared position. The TS model presents several drawbacks in WSN: first the network cannot self-organize in an entirely distributed manner, and second the trusted nodes have to be checked regularly and manually to actually remain trusted.

Relaxing the assumption of trusted nodes makes the problem more challenging, and to our knowledge, has only been investigated very recently by Hwang et al. [12]. We call the setting where no trusted node preexists the *no trusted sensor* (or *NTS*) model. The approach in [12] is randomized and consists of two phases: distance measurement and filtering. In the distance measurement phase, sensors measure their distances to their neighbors, faking sensors being allowed to corrupt the distance measuring technique. In the filtering phase each correct sensor randomly picks up 2 so-called *pivot* sensors. Next each sensor v uses trilateration with respect to the chosen pivot sensors to compute the location of its neighbor u. If there is a match between the announced location and the computed location, the (u, v) link is added to the network, otherwise it is discarded. Of course, the chosen pivot sensors could be faking and lying, so the protocol may only give probabilistic guarantee. The recent work of [22] investigates the related problem of neighbor discovery, where every participant has to find out who are its neighbors. The combination of a deterministic protocol and network asynchrony rules out the existence of a purely algorithmic solution, so they add to each node an *oracle* that permits to distinguish which neighbor claims are correct and which are not. Since the neighbor identification problem can be seen as a subtask of the secure positioning we focus on in this paper (the actual neighbors of a node could be deduced from the neighbors that send securely verified coordinates); this justifies the hypothesis we take here about system synchrony.

Our results. The main contribution of this paper is a *deterministic* secure positioning protocol, FindMap, in the NTS model. To the best of our knowledge, this is the first fully deterministic protocol in the NTS model. The protocol guarantees that the correct nodes never accept faked positions of the cheating nodes. The basic version of the protocol assumes that faking sensors are not able to mislead distance evaluation techniques. The protocol correctly filters out cheating sensors provided there are at most $\lceil \frac{n}{2} \rceil - 2$ faking sensors. Conversely, we provide evidence that in the same setting it is impossible to deterministically solve the problem when the number of faking sensors is at least $\lceil \frac{n}{2} \rceil - 1$. We then extend the protocol to deal with faking sensors that are also allowed to corrupt the distance measure technique (RSS or ToF). In the case of RSS, our protocol tolerates at most $\lfloor \frac{n}{2} \rfloor - 2$ faking sensors (provided no four sensors are located on the same circle and no four sensors are co-linear). In the case of ToF, our protocol may handle up to $\lfloor \frac{n}{2} \rfloor - 3$ faking sensors (provided no six sensors are located on the same hyperbola and no six sensors are co-linear).

Our results have significant impact on the secure positioning problem in the TS model as well. The TS protocol presented by Capkun et al. [3] relies on set of hidden station that detect inconsistencies between measured distance and distance computed from claimed coordinates. The authors propose the ToF-like technique to estimate the distance. Our detailed analysis shows that six hidden stations (verifiers) are sufficient to detect a faking node. The authors also conjecture that the ToF-like technique could be replaced with the RSS technique. Our results answer positively to the open question of [3], improving the number of needed stations to four. Thus, in the TS model our results can be used to efficiently deploy a minimal number of trusted stations.

Our FindMap protocol can be used to prevent Sybil attack [5]. More specifically, each message can be verified if it contains real position (id) of its sender. Each message that is found to contain faked position (id) can be discarded. Thus correct nodes never accept messages with a faked sender position (id).

2. Technical preliminaries

The sensors are located in the plane. We assume the NTS model, unless stated otherwise. That is, initially correct nodes do not have any knowledge about the network. In particular, correct nodes do not know the position of other correct nodes. Each node is aware of its own geographic coordinates, and those coordinates are used to identify nodes. The WSN is partially synchronous: all nodes operate in rounds. In one round every node is able to transmit exactly one message that reaches all nodes in the network. The size of the WSN is n. Our protocol does not rely on n, since participants are unaware of the network size. For each transmission, correct nodes use the same transmission power.

The total number of faking nodes is denoted by f. Faking nodes are allowed to corrupt protocol's messages *e.g.* transmit incorrect coordinates (and thus incorrect identifier) to the other nodes. We assume that faking nodes may cooperate between themselves in an omniscient manner (*i.e.* without exchanging messages) in order to fool the correct nodes in the WSN. In the basic protocol, faking nodes cannot corrupt distance measure techniques. This assumption is then relaxed in Sections 4 and 5 where faking sensors can corrupt the ranging technique. We assume however that each faking node obeys synchrony. That is, each faking node transmits at most one message per round.

We assume that all distance-ranging techniques are perfect with respect to precision. This assumption is further discussed in the conclusion. The distance computed by a node v to a node u based on a distance ranging technique is denoted by $\hat{d}(v, u)$. The distance computed by a node v to a node u using coordinates provided by u is denoted by $d(v, u)$. A particular sensor v *detects inconsistency* on distance (*i.e.* position) of sensor u if $d(v, u) \neq \hat{d}(v, u)$. Our protocols rely on detecting and reporting such inconsistency.

In the remaining of the paper, we use two distance estimation techniques:

1. In the *received signal strength* (*RSS*) technique, we assume that each node can precisely measure the distance to the transmitting node by Friis transmission equation (1) [17]:
$$S_r = S_s \left(\frac{\lambda}{4\pi d}\right)^2 \tag{1}$$
where S_s is the transmission power of the sender, S_r is the receive signal strength (RSS) of the wave at receiver, λ is wave length and d is distance between sender and receiver.
2. The *synchronous time of flight* (*SToF*) technique relies on propagation time of the radio signal. For this technique we assume that sensors are synchronized by global time. Sender u attaches the time of transmission, t_s to the message. Receiver v records the message arrival time t_r of the message. Next v computes the distance $d = t * s$ of u based on time delay $t = t_r - t_s$ of the message and radio signal speed s.
3. The *different arrival time* (*DAT*) technique provides similar guarantees as SToF. The advantage of DAT over SToF is that DAT does not require synchronization. In the DAT technique each sensor transmits its message with two types of signals that differ on propagation speed *e.g.* radio signal (RF) and ultra sound signal (US). Sender sensor u transmits its message with RF and US signal simultaneously. Receiver sensor v, which estimates its distance to sender u, records arrival time t_r of RF signal and arrival time t_u of US signal from u. Then, based on the propagation speed s_r of RF, propagation speed s_u of US and difference of arrival times $t = t_u - t_r$ sensor v can compute distance to sensor u. Eq. (2) show the relation.
$$t = \frac{\hat{d}}{s_r} - \frac{\hat{d}}{s_u}. \tag{2}$$

3. Basic protocol

In this section we present the protocol FindMap that essentially performs by majority voting. The protocol detects all faking sensors provided $n - 2 - f > f$. Thus, the total number of faking sensors is at most $\lceil \frac{n}{2} \rceil - 2$. In this section we consider the relatively simpler case where faking sensors cannot cheat on ranging techniques. That is faking nodes cannot change its transmission power, but they can cooperate and corrupt the protocol. In Sections 4 and 5, the protocol will be extended to the case where faking nodes corrupt the ranging technique. Our key assumption is that no three correct sensors are co-linear. This assumption allows to formulate the following fact.

Fact 1. *If a cheating sensor transmits a message with a faked position then at least one of the three correct sensors can detect an inconsistency provided that no three sensors are co-linear (see Fig. 1).*

Based on Fact 1, we can develop FindMap(k), where k is the maximum number of correct nodes which cannot detect inconsistency. By Fact 1 at most $k = 2$ correct nodes will not detect inconsistency. The protocol operates in two rounds. In *Round 1* all sensors exchange their coordinates by transmitting an initial message. Next, each node v computes the distances $\hat{d}(v, u)$ (from the ranging technique) and $d(v, u)$ (from the obtained node coordinates) of u and compare them. If $\hat{d}(v, u) \neq d(v, u)$ then v accuses u of faking its position. Otherwise, v approves u (v believes that u is correct). To keep record of its accusations and approvals, each node v maintains an array $accus_v$. In *Round 2* each node v transmits its array

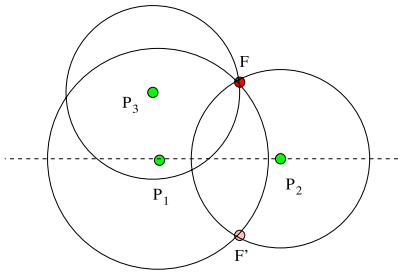


Fig. 1. An example showing F consistently fakes its location to F' against P_1 and P_2. However P_3 always detects an inconsistency since no three correct sensors are co-linear.

$accus_v$. Next, each node v counts accusations and approvals toward the node u including its own messages. Node v finds the node u as faking if the number of accusations is strictly larger than number of approvals minus k.

Protocol FindMap(k) executed by node v

Round 1:
1. v exchanges coordinates by transmitting $init_v$ & receiving $init_u$.
2. for each received message $init_u$:
3. compute $\hat{d}(v, u)$ with ranging technique and $d(v, u)$ using the coordinates of u.
4. **if** $(\hat{d}(v, u) \neq d(v, u))$ **then** $accus_v[u] \leftarrow true$ **else** $accus_v[u] \leftarrow false$

Round 2:
5. v exchange accusations by transmitting $accus_v$ & receiving $accus_u$.
6. for each received $accus_u$:
7. for each sensor r that participated in Round 1
8. **if** $accus_u[r] = true$ **then** $NumAccus_r += 1$ **else** $NumApprove_r += 1$
9. for each sensor u:
10. **if** $(NumAccus_u > NumApprove_u - k)$ **then** v finds u as faking **else** v finds u as correct.

Theorem 1. *Each correct sensor, running protocol* FindMap($k = 2$)*, never accepts position of faking sensor provided $n-f-2 > f$ (at most $\lceil \frac{n}{2} \rceil - 2$ faking nodes) and no three sensors are co-linear.*

Proof. Let us assume $k = 2$, $n - f - k > f$ and no three sensors are co-linear. First we will show that all correct sensors will detect each faking sensor. By Fact 1, at most $k + k$ sensors will approve each faking sensor v (at most f faking sensors and at most k correct sensors), and at least $n - f - k$ correct sensors will accuse each faking sensor v. Thus the inequality $n - f - k > f + k - k$ in line 10 of the protocol will be true due to our assumption that $n - f - k > f$. So faking node v will be identified by all correct nodes.

Next, we have to show that no correct sensor will be found faking. We can see that each correct sensor u can be accused by at most f faking sensors and each correct sensor u will be approved by all $n - f$ correct sensors. Thus the inequality $f > n - f - k$ in line 10 of the protocol will be false due to our assumption $f < n - f - k$. So each correct node u will be identified as a correct one by all correct nodes. □

Next, we show that it is impossible to filter out the faking sensors when $n - 2 - f \leq f$. The assumption that faking sensors cannot corrupt the distance ranging technique makes this result even stronger. Our protocol is synchronous but this impossibility result holds for asynchronous settings also.

Theorem 2. *If $n - f - 2 \leq f$ then faking nodes cannot be identified by a deterministic protocol.*

Proof. The proof is by contradiction. That is we assume that deterministic protocol $\mathcal{P}$, that identifies faking sensors for $n - f - 2 \leq f$, exists. The main idea of the proof is to construct two executions that are indistinguishable to sensors v and u. In the first execution both sensors v and u have to successfully identify faking sensors. In the second execution the correct sensors behave as if they were faking sensors in the first execution and faking sensors behave as if they were correct sensors in the first execution. Thus, in the second execution sensors v and u will find correct sensors as faking. This implies that no deterministic protocol is able to identify faking sensors for $n - f - 2 \leq f$. The details are as follows.

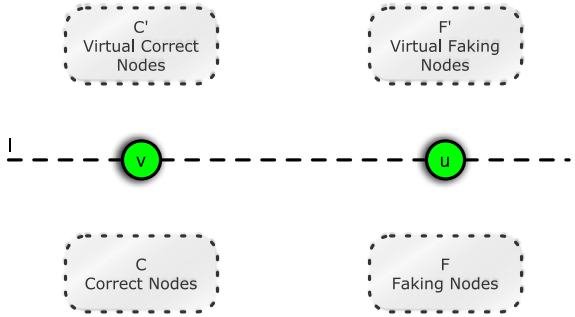


Fig. 2. First execution.

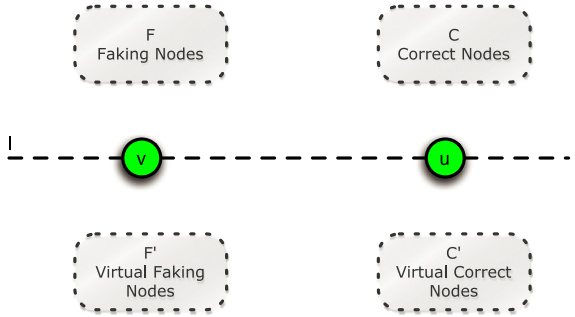


Fig. 3. Second execution.

Let us assume that correct sensors run a protocol $\mathcal{P}$, which allows to detect location of correct sensors and identify the faking sensors even when $n-f-2=f$. In the case where $n-f-2<f$, we make some faking sensors correct to achieve equality, and in the case where n is odd, one of the faking sensors will remain silent. Let us consider the first execution (see Fig. 2). There are two correct sensors v and u located on the straight line l. There are two sets of sensors C-correct sensors and F-faking sensors located on the lower half of the plane. The sizes of the sets are equal $|C| = |F| = f$. The sensors in F are trying to convince sensors v and u that they are located on the other side of the straight line l symmetrically. Each sensor in F behaves as if it was a correct sensor located symmetrically against straight line l. These sensors create set F' of virtually faking sensors. More precisely, F' contains all the faking sensors in F with their faked coordinates. Virtual sensors in F' execute the protocol as if sensors in C were faking and their correct location was in C', which is the symmetric reflection of C against straight line l. Construction of the second execution will clarify why we need such behavior of sensors in F'. We can see that sensors v and u are not able to detect inconsistency directly on the distance of virtual faking sensors since symmetry preserves their distances from v and u. By our assumption about correctness of the protocol $\mathcal{P}$, sensors v and u are able to verify that sensors in F' are faking.

Now let us consider the second execution (see Fig. 3). In the second execution sensors in C and F' are swapped. Thus sensors in F has to be located on the other side of straight line l symmetrically. Now virtual faking sensors in F' can imitate the first execution of the correct sensors in C. Correct sensors in C behave like virtual sensors in F' in the first execution. This is because the virtual sensors in F' in the first execution behaved like correct sensors and additionally they claimed that sensors from C were located in C' (see Fig. 3). F is really located in the previous location of C' and the sensors in C are correct. Thus sensors v and u are not able to distinguish between the first and the second execution. Sensors v and u will have to decide that C is set of faking sensors. This is because v and u have made such a decision in the first execution, v and u are not able to distinguish between these two executions. □

4. Protocol based on RSS ranging technique

In this section we assume that sensors use the RSS technique to measure distance. We assume that each correct sensor has a fixed common transmission signal strength of S_s. The faking sensors can change their transmission signal strength to prevent the correct sensor from computing correct distance. Let F be a faking sensor that changes its signal strength $S_s^{'}$ and

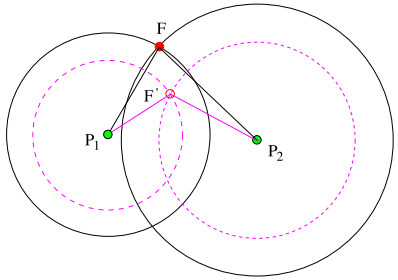


Fig. 4. An example showing a faking sensor F can supply its suitable false position F' to correct sensors P_1 and P_2 by changing its signal strength.

sends a suitable faked position F' to other correct sensors v. Sensor v can estimate the distance, $\hat{d}$ from the received signal strength (RSS) by the Friis transmission equation (1) assuming, the common signal strength S_s has been used.

$$\hat{d}^2 = c\frac{S_s}{S_r} \implies \hat{d}^2 = \frac{S_s}{S_s^{'}}d^2 \tag{3}$$

where $c = \left(\frac{\lambda}{4\pi}\right)^2, S_r = c\frac{S_s^{'}}{d^2}$, and d is the distance from v to the actual position of F.

We show that Protocol FindMap can be adapted to this model provided that $n-3-f > f$, *i.e.* the total number of faking sensors is at most $\lfloor \frac{n}{2} \rfloor - 2$ and no four correct sensors are located on a particular circle. In this variant of the protocol, a sensor v considers sensor u faking if the number of accusation messages for u is at least $\lceil \frac{n}{2} \rceil - 1$.

Lemma 1. *Let F be a faking sensor. If F fakes its position and transmission power then the set of correct nodes that cannot detect inconsistency is located on a circle.*

Proof. Let us assume that faking sensor F, located in (x_f, y_f), changes its signal strength from S_s to $S_s^{'}$ and sends a corresponding faked position (x_f', y_f') to two correct sensors P_1 and P_2. Faked position (x_f', y_f') reported by sensor F is computed so that P_1 and P_2 cannot detect inconsistency. More precisely, (x_f', y_f') is the point of intersection of the two circles centered at P_i, for $i = 1, 2$, with radius $\hat{d}_i$, where $\hat{d}_i$ is the distance measured by P_i with the RSS ranging technique

$$\hat{d_1}^2 = \frac{S_s}{S_s^{'}}{d_1}^2 \quad \text{and} \quad \hat{d_2}^2 = \frac{S_s}{S_s^{'}}{d_2}^2$$

and d_i is the real distance P_i to F (see Fig. 4). We can see that neither P_1 nor P_2 is able to detect the inconsistency of the faked position (x_f', y_f').

Let us observe that each correct sensor P such that $\hat{d_p} = d_p\sqrt{\frac{S_s}{S_s^{'}}}$, where $\hat{d_p}$ is distance computed by P to F with RSS and d_p is the real distance P to F, is not able to detect inconsistency. Therefore, the possible location (x, y) of the point P is $\frac{(x-x_f')^2+(y-y_f')^2}{(x-x_f)^2+(y-y_f)^2} = \frac{S_s}{S_s^{'}} \implies x^2+y^2-2\left(\frac{x_f'-\lambda^2 x_f}{1-\lambda^2}\right)x - 2\left(\frac{y_f'-\lambda^2 y_f}{1-\lambda^2}\right)y + \frac{{x_f'}^2+{y_f'}^2-\lambda^2({x_f}^2+{y_f}^2)}{1-\lambda^2} = 0$ where $\lambda = \sqrt{\frac{S_s}{S_s^{'}}}$.

This is an equation of a circle with respect to the given faked position F' of F and P_1 and P_2 as shown in Fig. 5. Therefore, F pretends the faked position F' to the sensors which are laying only on the particular circle. □

Theorem 3. *If the distance evaluation is done with the RSS technique and no four sensors are located on the same circle and no four sensors are co-linear, then at least one out of four correct sensors detects inconsistency in the faked transmission.*

Proof. By Lemma 1 and our assumption it is said that no four sensors are located on the same circle while three correct sensors accept faked locations. This is because, a circle is uniquely determined by three points on the plane, and the fourth sensor which is not on the circle detects inconsistency. Additionally, the faking sensor can be accepted by at most 3 correct sensors located on the line provided the faking sensor does not change its transmission power and reports its faked position symmetrically against the line. □

Theorem 3 allows to adjust the FindMap protocol.

Corollary 1. *Each correct sensor, running the protocol* FindMap($k = 3$)*, never accepts position of the faking node, in the model where faking sensors can corrupt the RSS ranging technique, provided $n-f-3 > f$ (at most $\lfloor \frac{n}{2} \rfloor - 2$ faking nodes) and no four sensors are located on the same circle and no four nodes are co-linear.*

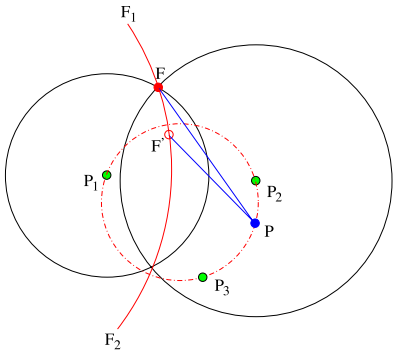


Fig. 5. An example showing a faking sensor F can lie about its position by changing the signal strength to a multiple number of correct sensors which are laying on a particular circle.

Proof. Let us consider a faking sensor F, which fakes its transmission power. By Theorem 3 in each set of four correct sensors at most 3 sensors accept faked positions. Thus, each faking sensor will be accused by at least $n - f - 3$ correct sensors. By inequality $n - f - 3 > f$ the number of correct sensors that accuse F is at least $\lceil \frac{n}{2} \rceil - 1$ and the number of faking sensors is at most $\lfloor \frac{n}{2} \rfloor - 2$. Thus, each faking sensor will be found faking and no correct sensor will be found faking. Additionally, if faking sensor F does not change its transmission power but only lies about its position then at least one among the three co-linear correct sensors will detect inconsistency. □

Theorem 3 can be also applied in the protocol for the model of trusted sensors. In the protocol presented in [3], we can use Theorem 3 to find deployment of the minimum number of hidden stations required to detect faking nodes.

Corollary 2. *If the four hidden stations are neither located on the same circle nor co-linear then one of the stations will always detects a faked transmission.*

More precisely, if a sensor transmits a faked position then at least one of the hidden stations detects inconsistency. Corollary 2 remains true provided the faking node's transmission reaches all hidden stations and it is not allowed to use directional antennas. Since the verifiers are hidden to the faking node in the model of [3], the latter has very low chances to consistently fake its position even with directional antennas.

5. Protocol based on ToF-like ranging techniques

We first discuss how faking sensors can corrupt the two SToF and DAT ranging techniques:

1. In the case where the *SToF* ranging technique is used by sensor u, u first transmits a message attaching the time of transmission t_s into the message. Sensor v, receives the message from sensor u at time t_r, estimates the distance based on delay $t = t_r - t_s$ and radio signal propagation speed s_r, $\hat{d}(v, u) = s_r t$. So, it is possible that a faking sensor can prevent sensor v from computing the real distance by faking the transmission time t_s.
2. In the case where the *DAT* ranging technique is used, sensor u transmits each message simultaneously with two signals (*e.g.* RF and US signals). Sensor v then records the difference of arrival time t between the RF and the US signal. This can be done using only a local clock at v. Thus no global time is required. Then, v computes distance $\hat{d}(v, u)$ based on t, propagation speed s_r of RF signal and propagation speed s_u of US signal. In this case, a faking sensor may prevent a correct sensor v from computing real distance by delaying one of the two simultaneous transmissions.

Now we show that the corrupted SToF and DAT ranging techniques have the same effect on correct sensors.

Lemma 2. *If the ranging is evaluated with the SToF technique, and if the faking sensor F shifts its real transmission time, then all correct sensors compute the real distance to sensor F increased or decreased by the same length b.*

Proof. Let us assume that faked sensor F shifts its real transmission time by t'. Then all the correct sensors will compute the distance modified by $b = s_r t'$, where s_r is the radio signal propagation speed. □

Lemma 3. *If the ranging is evaluated with the DAT technique and faking sensor F introduces shift $t' \neq 0$ between the RF and US transmissions, then all correct sensors compute the real distance to sensor F, increased or decreased by the same length b.*

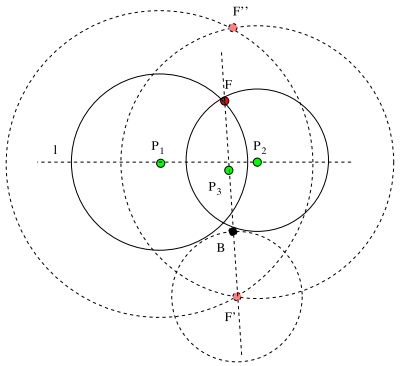


Fig. 6. This figure shows that sensor F can change its position to F' and consistently lie against sensor P_3 which is located in the middle of segment FB. Length of segment $F'B$ is b.

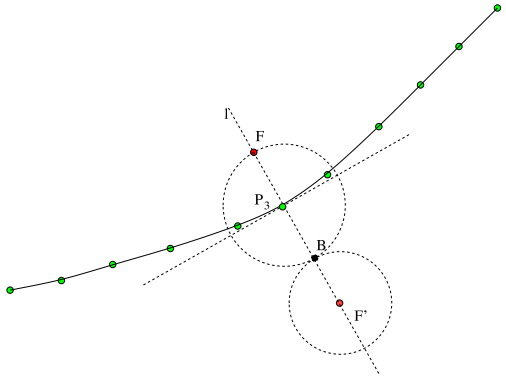


Fig. 7. We assume that $|FF'| > b$. This figure shows set S of correct sensors located on the hyperbola, which cannot detect inconsistency. That is for each correct sensor P located on the hyperbola the distance $|F'P|$ is equal to $|FP| + b$

Proof. Since the faking sensor shifts the two transmissions by time t' then the difference in arrival times of the signals will be $t + t'$ where t is the original difference for $t' = 0$. Each correct sensor will compute $\hat{d}$ based on the following equation.

$$t + t' = \frac{\hat{d}}{s_r} - \frac{\hat{d}}{s_u}.$$

Thus the real distance will be modified by

$$b = \frac{t'}{1/s_r - 1/s_u}$$

in all correct sensors. □

Since the corruption on SToF and DAT has the same result we can formulate the following theorem for both ranging techniques.

Theorem 4. *If the distance evaluation is done with the SToF or DAT techniques and no six sensors are located on the same hyperbola and no six sensors are co-linear, then at least one of the six correct sensors detects inconsistency in faked transmission.*

Proof. Let us assume that faking sensor F enlarges its distance against the correct sensor by b. The case when the sensor reduces its distance is symmetric. By Lemmas 2 and 3 there are at most two faked locations F' and F'' for faking sensor F, which guarantee consistency against sensors P_1 and P_2 (see Fig. 6). Let us assume that sensor F decides for faked location F'.

Now we will find a set of correct sensors, which will not detect inconsistency. We consider two cases:

1. The first case is when distance c between F' and F is strictly larger than b (see Fig. 7). Each correct sensor P, which cannot detect inconsistency on distance to F, has to meet $d(P, F) = \hat{d}(P, F)$. The condition $d(P, F) = \hat{d}(P, F)$ can be transformed

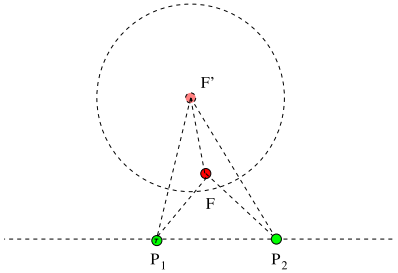


Fig. 8. We assume $FF' \leq b$. This figure shows that faking sensor F cannot change its position to F' consistently against sensors P_1 and P_2. That is $F'P_1 < |FP_1| + b$ or $F'P_2 < |FP_2| + b$ allowing sensor P_1 or P_2 to detect inconsistency.

into the distances on the plane $|F'P| = |FP| + b$. Based on this condition we can come up with a system of equations for sensors in $S = \{P : d(P, F) = \hat{d}(P, F)\}$.

$$\begin{aligned} x^2 + y^2 &= z^2 \\ x^2 + (y - c)^2 &= (z + b)^2 \end{aligned} \tag{4}$$

where $|FP| = z, x, y$ are the coordinates of correct sensor $P \in S$. We assume that $F = (0, 0)$ and $F' = (0, c)$. Next, we can find analytical description of points in S on the Eq. (4).

$$\begin{aligned} &x^2 + (y - c)^2 = (\sqrt{x^2 + y^2} + b)^2 \\ &x^2 + y^2 - 2yc + c^2 = x^2 + y^2 + 2b\sqrt{x^2 + y^2} + b^2 \\ &(-2yc + c^2 - b^2)^2 = 4b^2(x^2 + y^2) \\ &4y^2c^2 - 4yc(c^2 - b^2) + (c^2 - b^2)^2 = 4b^2(x^2 + y^2) \\ &4y^2c^2 - 4yc(c^2 - b^2) + (c^2 - b^2)^2 - 4b^2y^2 = 4b^2x^2 \\ &4(c^2 - b^2)y^2 - 4c(c^2 - b^2)y + (c^2 - b^2)^2 = 4b^2x^2 \\ &(c^2 - b^2)(4y^2 - 4cy + c^2 - b^2) = 4b^2x^2 \\ &(c^2 - b^2)((2y - c)^2 - b^2) = 4b^2x^2 \\ &(c^2 - b^2)(2y - c)^2 - b^2(c^2 - b^2) = 4b^2x^2 \\ &(c^2 - b^2)(2y - c)^2 - 4b^2x^2 = b^2(c^2 - b^2) \\ &\frac{(2y - c)^2}{b^2} - \frac{4x^2}{c^2 - b^2} = 1. \end{aligned}$$

This is an equation of the hyperbola. The five points uniquely determine the hyperbola. Thus, if no 6 sensors are located on a hyperbola then at most 5 correct sensors accept faked positions.

2. The second case is when distance c between F' and F is at most b (see Fig. 8). We will show that P_1 or P_2 will have to detect inconsistency. The distance measured using coordinates by P_i, for $i = 1, 2$, has to be exactly $|FP_i| + b$ to prevent sensor P_i from detecting inconsistency. By triangle inequality we have $|F'F| + |FP_i| \geq |F'P_i|$ for $i = 1, 2$. Thus the distance $|F'P_i|$ measured by P_i with a ranging technique is at most $|FP_i| + b$. Sensor P_i for $i = 1, 2$ will measure required distance when sensors F', F and P_i are co-linear. This will happen for at most one sensor. This is because we assume that no three sensors are co-linear. □

Theorem 4 allows us to modify the protocol FindMap so that it works in the model in which faking sensors can corrupt the SToF or DAT ranging technique.

Corollary 3. *The protocol* FindMap($k = 5$) *identifies all faking sensors, in the model where faking sensors can corrupt the SToF or DAT ranging techniques, provided* $n - f - 5 > f$ *and no six sensors are located on the same hyperbola and no three sensors are co-linear.*

Proof. Let us consider a faking sensor F. Theorem 4 guarantees that in each set of six correct sensors there exists a sensor which detects inconsistency on distance to F. Thus each faking sensor will be accused by at least correct $n - f - 5$ sensors. By inequality $n - f - 5 > f$ the number of correct sensors that accuse F is at least $\lceil \frac{n}{2} \rceil - 2$ and the number of faking sensors is at most $\lfloor \frac{n}{2} \rfloor - 3$. Thus each faking sensor will be found faking, and no correct sensor will be found faking. □

Theorem 4 can be also applied in the protocol for the model of trusted sensors [3]. We can use Theorem 4 to compute the deployment of the minimum number of hidden stations required to detect faking nodes.

Corollary 4. *If the six hidden stations are not located on the same hyperbola and no three stations are co-linear then one of the stations always detect a faking node.*

Corollary 4 is true provided the attacker's transmission reaches all the hidden stations and the attacker is not allowed to use directional antennas. Since the verifiers are hidden to faking nodes, the latter has very low chance to consistently fake its position even with directional antennas.

6. Secure positioning in case of imperfect ranging techniques

In this paper we solve the secure positioning problem assuming a perfect ranging technique. However, for the case of RSS ranging technique, this assumption may not hold due to the presence of noise in the network channel, that would cause in turn signal attenuation not to necessarily follow the Friis transmission equation (1) we have used. It is possible to incorporate an additional noise factor (ε) that can be assumed to follow a normal (Gaussian) distribution with mean 0 and variance σ^2. The Friis transmission equation becomes $S_r = S_s \left(\frac{\lambda}{4\pi d}\right)^2 + \varepsilon$ where $\varepsilon \sim N(0, \sigma^2)$. In this case every estimated distance $\hat{d}(v, u)$ using the RSS technique is expected to lie in the R range, which is a function of distance $d(v, u)$. If $\hat{d}(v, u) \notin R$ then v accuses u of faking its position, otherwise, v approves u.

However, we assume σ^2 to be known by all nodes. If it is unknown, each sensor can estimate it by sending signals from known distances and measuring the deviations in received signal strengths from those expected in an ideal situation. Checking the distribution of these deviations permits to assess whether the error distribution is indeed normal (see [10,20] for further reference to normality tests of error distributions). If the result differs from normality, one can choose a suitable model for the error distribution and deduce the acceptance interval using the quantiles of that distribution. Here, for the sake of simplicity, we assume the error distribution to be normal, which is the most common and popular choice in the literature.

Also, one may occasionally need empirical adjustments to the basic Friis Eq. (1) using larger exponents. These are used in terrestrial models, where reflected signals are likely to result in destructive interference, and foliage and atmospheric gases contribute to signal attenuation [6]. In this case, one can consider S_r/S_s to be proportional to $G_rG_s\left(\frac{\lambda}{d}\right)^m$, where G_r and G_s are mean effective gain of the antennas and m is a scalar that typically lies in the range [2, 4].

7. Concluding remarks

We have proposed a secure deterministic position verification protocol for WSN that performs in the most general NTS model. Although the previous protocol of Hwang et al. [12] for the same problem is randomized and only gives probabilistic guarantees, it is interesting to compare the number of transmitted messages. In [12], each sensor announces one distance at a time in a round robin fashion. Otherwise the faking node may hold its own announcement, collects all correct nodes' information, and sends a consistently faked distance claim. Thus the message complexity of [12] is $O(n^2)$. In our case, $2n$ messages are transmitted in two rounds. Thus our protocol improves both time and power used for transmissions significantly, while providing certain results rather than probabilistic ones.

Our network model assumes that correct nodes are within the range of every other node. We believe that our majority voting heuristic could provide robust results for arbitrary network topology. We can observe that a correct node u identifies its faking neighbor v provided that the number of correct u's neighbors which report inconsistency on v is strictly larger than the number of faking u's neighbors.

Extending our result to WSN with fixed ranges for every node is a challenging task, especially since previous results on networks facing intermittent failures and attacks [18] are written for rather stronger models (*i.e.* wired secure communications) than that of this paper.

Acknowledgements

The first author is is supported by ANR project SHAMAN. Part of this work was done while the second author was a postdoctoral fellow at INRIA in the Grand Large project-team, Univ. Paris Sud, France. Part of this work was done while the third author was a postdoctoral fellow at CNRS in the Parall project-team, Univ. Paris Sud, France. The last author is supported by ANR projects SHAMAN, ALADDIN, and R-DISCOVER.

References

[1] P. Bahl, V.N. Padmanabhan, Radar: an in-building rf-based user location and tracking system, in: INFOCOM, vol. 2, IEEE, 2000, pp. 775–784.
[2] S. Brands, D. Chaum, Distance-bounding protocols, in: EUROCRYPT'93: Workshop on the Theory and Application of Cryptographic Techniques on Advances in Cryptology, Springer-Verlag, Secaucus, USA, 1994, pp. 344–359.
[3] S. Capkun, M. Cagalj, M.B. Srivastava, Secure localization with hidden and mobile base stations, in: INFOCOM, IEEE, 2006.
[4] S. Capkun, J. Hubaux, Secure positioning in wireless networks, IEEE Journal on Selected Areas in Communications: Special Issue on Security in Wireless Ad Hoc Networks 24 (2) (2006) 221–232.
[5] J.R. Douceur, The sybil attack, in: IPTPS'01: Int. Workshop on Peer-to-Peer Systems, in: LNCS, vol. 2429, Springer-Verlag, London, UK, 2002, pp. 251–260.

[6] Bruce Fette, Cognitive Radio Technology, second edition, Academic Press, 2009.
[7] R.J. Fontana, E. Richley, J. Barney, Commercialization of an ultra wideband precision asset location system, in: Ultra Wideband Systems and Technologies, 2003 IEEE Conference on, 2003, pp. 369–373.
[8] T. He, S. Krishnamurthy, J.A. Stankovic, T. Abdelzaher, L. Luo, R. Stoleru, T. Yan, L. Gu, J. Hui, B. Krogh, An energy-efficient surveillance system using wireless sensor networks, in: MobiSys'04: Proc. of the 2nd Int. Conf. on Mobile Systems, Applications, and Services, New York, USA, 2004, pp. 270–283.
[9] J. Hightower, R. Want, G. Borriello, SpotON: an indoor 3d location sensing technology based on RF signal strength, UW CSE 00-02-02, University of Washington, Department of Computer Science and Engineering, Seattle, WA, February 2000.
[10] M. Hollander, D.A. Wolfe, Nonparametric Statistical Methods, Wiley, New York, 1999.
[11] Y. Hu, A. Perrig, D.B. Johnson, Packet leashes: a defense against wormhole attacks in wireless networks, in: INFOCOM, IEEE, 2003.
[12] J. Hwang, T. He, Y. Kim, Detecting phantom nodes in wireless sensor networks, in: INFOCOM, IEEE, 2007, pp. 2391–2395.
[13] B. Karp, H.T. Kung, Gpsr: greedy perimeter stateless routing for wireless networks, in: MobiCom'00: Proc. of the 6th Annual Int. Conf. on Mobile Computing and Networking, ACM Press, New York, USA, 2000, pp. 243–254.
[14] Leslie Lamport, Robert Shostak, Marshall Pease, The byzantine generals problem, ACM Trans. on Programming Lamguages and Systems 4 (3) (1982) 382–401.
[15] L. Lazos, R. Poovendran, Serloc: Robust localization for wireless sensor networks, ACM Trans. Sen. Netw. 1 (1) (2005) 73–100.
[16] L. Lazos, R. Poovendran, S. Capkun, Rope: robust position estimation in wireless sensor networks, in: IPSN, IEEE, 2005, pp. 324–331.
[17] C.H. Liu, D.J. Fang, Propagation. in antenna handbook: Theory, applications, and design, Van Nostrand Reinhold, 1988, pp. 1–56 (Chapter 29).
[18] Mikhail Nesterenko, Sébastien Tixeuil, Discovering network topology in the presence of byzantine faults, IEEE Trans. Parallel Distrib. Syst. 20 (12) (2009) 1777–1789.
[19] N.B. Priyantha, A. Chakraborty, H. Balakrishnan, The cricket location-support system, in: 6th ACM MOBICOM, ACM, Boston, MA, 2000.
[20] S.S. Shapiro, M.B. Wilks, An analysis of variance test for normality (complete samples), Bometrika 52 (3–4) (1965) 591–611.
[21] R. Szewczyk, A. Mainwaring, J. Polastre, J. Anderson, D. Culler, An analysis of a large scale habitat monitoring application, in: SenSys'04: Proc. Int. Conf. Embedded Networked Sensor Systems, ACM Press, NY, USA, 2004, pp. 214–226.
[22] Adnam Vora, Mikhail Nesterenko, Sébastien Tixeuil, Sylvie Delaët, Universe detectors for sybil defense in ad hoc wireles networks, in: International Conference on Stabilization, Safety, and Security, SSS 2008, in: Lecture Notes in Computer Science, Springer-Verlag, 2008.
[23] R. Want, A. Hopper, ao V. Falc, J. Gibbons, The active badge location system, ACM Trans. Inf. Syst. 10 (1) (1992) 91–102.
[24] A. Ward, A. Jones, A. Hopper, A new location technique for the active office, Personal Communications, IEEE [see also IEEE Wireless Communications], 4(5) (1997) 42–47.

A.3 Recenser les présents

Tolerating Transient and Intermittent Failures. Sylvie Delaët, and Sébastien Tixeuil. *Journal of Parallel and Distributed Computing (JPDC)*, 62(5) :961-981, May 2002.

– *Résumé :* Sur un réseau de processeurs (aussi appelé système réparti), le problème du recensement consiste à obtenir dans la mémoire locale de chaque processeur la liste des identificateurs de tous les autres processeurs du réseau ainsi que leur distance relative. Nous proposons ici un algorithme qui résoud le problème du recensement tout en supportant des défaillances transitoires des mémoires locales de chaque processeur ainsi que des liens de communication (il est auto-stabilisant). De plus, il fonctionne correctement même en présence de fautes intermittentes des liens de communication. Ceux-ci peuvent perdre des messages (de manière équitable), les dupliquer (de manière finie) les déséquencer, voire combiner ces trois défaillances. Une preuve formelle est associée à la présentation de l'algorithme pour assurer qu'il résoud le problème tout en tolérant les défaillances précitées. Notre algorithme peut être utilisé comme canevas pour résoudre n'importe quel problème statique tout en tolérant les défaillances transitoires de corruption de mémoire et les défaillances intermittentes des liens de communication.

Journal of Parallel and Distributed Computing **62**, 961–981 (2002)
doi:10.1006/jpdc.2001.1827

Tolerating Transient and Intermittent Failures[1]

Sylvie Delaët and Sébastien Tixeuil[2]

Laboratoire de Recherche en Informatique, UMR CNRS 8623, Université de Paris Sud, 91405 Orsay Cedex, France
E-mail: delaet@lri.fr, tixeuil@lri.fr

Received January 31, 2000; accepted January 11, 2002

Fault tolerance is a crucial property for recent distributed systems. We propose an algorithm that solves the census problem (list all processor identifiers and their relative distance) on an arbitrary strongly connected network.

This algorithm tolerates transient faults that corrupt the processors and communication links memory (it is self-stabilizing) as well as intermittent faults (fair loss, reorder, finite duplication of messages) on communication media. A formal proof establishes its correctness for the considered problem. Our algorithm leads to the construction of algorithms for any silent problems that are self-stabilizing while supporting the same communication hazards. © 2002 Elsevier Science (USA)

Key Words: self-stabilization; unreliable communication; census.

1. INTRODUCTION

We consider distributed systems that consist of a collection of processors linked to each other by communication media that allow them to exchange messages. As larger systems imply less reliability, several kinds of failures can be classified according to their locality (processor or link) and their nature.

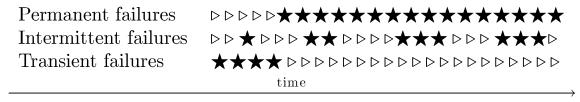
1. *Permanent failures* are failures that make one or several components of the system stop running forever. In the case of a processor, it stops executing its program forever. In the case of a communication link, this can be interpreted as a definitive rupture of the communication service.
2. *Intermittent failures* are failures that make one or several components of the system behave erratically from time to time. A processor can have a Byzantine behavior, and a communication link may loose, duplicate, reorder or modify messages in transit.

[1]An extended abstract of a preliminary version of this paper appeared in [3]. This work was supported in part by the French STAR project.
[2]To whom correspondence should be addressed.

0743-7315/02 $35.00

3. *Transient failures* are failures that place one or several components of the system in some arbitrary state, but stop occurring after some time. For a processor, such failures may occur following a crash and repair operation, or temporary shutdown of its power supply. For a communication link, electromagnetic fields may lead to similar problems.

The time organization of permanent, intermittent and transient failures is presented below, where ▷ denotes a correct behavior of the system, and where ★ denotes a failure of the system.



1.1. Failure-tolerance in Distributed Systems

Robustness is one of the most important requirements of modern distributed systems. Two approaches are possible to achieve fault-tolerance: on the one hand, robust systems use redundancy to mask the effect of faults, on the other hand, self-stabilizing systems may temporarily exhibit an abnormal behavior, but must recover correct behavior within finite time.

Robustness: When a small number of the system components fail frequently (this is the case for permanent and intermittent failures), robust systems should always guarantee a correct behavior. Most often, such approaches make the hypothesis of a limited number of faults, due to impossible results even when communication links are reliable. A fundamental result (see [20]) shows that the Consensus problem (all processors agree on a common initial value) is impossible to solve deterministically in an asynchronous system even if the failure is permanent and limited to a single processor. In the case of intermittent failures on communication links, most works deal with transformation of unreliable links into reliable links (for higher level messages). The case of fair loss was solved in [6, 27], but such constructions are impossible when links may duplicate (finitely) and reorder messages (see [28]).

Self-stabilization: Conversely, when all system components behave correctly most of the time (this is the case with transient failures), one could accept temporary abnormal service when that system suffers from a general failure, as long as recovery to a correct behavior is guaranteed after finite time. This approach, called self-stabilization, consists in always behaving as if all system components were correct. On the contrary to fault-tolerance, self-stabilization does not make any restriction of the subset of the system that is hit by the failure. Since its introduction by Dijkstra (see [14]), a growing number of self-stabilizing algorithms solving different problems have been presented (see [16]). In particular, several recent publications prove that being able to start from any arbitrary configuration is desirable as a property of fault tolerance. For example, Jayaram and Varghese [23] show that processor crashes and

restarts may lead a system to an arbitrary global state, from which a self-stabilizing algorithm is able to recover.

1.2. Related Work

Historically, research in self-stabilization over general networks has mostly covered undirected networks where bidirectional communication is feasible (the Update protocol of Dolev and Herman [18], or the algorithms presented in [3, 19]). For example, the self-stabilizing algorithms that are built upon the paradigm of local checking (see [5]) use bidirectional communication to compare one node state with those of its neighbors and check for consistency. The lack of bidirectional communication was overcome in recent papers using strong connectivity (which is a weaker requirement than bidirectional) to build a virtual well-known topology on which the self-stabilizing algorithm may be run (a tree in [1]). As many self-stabilizing algorithms exist for rings or trees in the literature, these constructions may be used to reuse existing algorithms in general networks.

Several algorithms are self-stabilizing and tolerate a limited amount of processor crash failures (see [4, 21, 9]). However, they are studied in a communication model that is almost reliable (links are only subject to transient failures). Most related to our problem is [25], where Masuzawa presents a self-stabilizing Topology Update algorithm that also supports, to a certain extent, permanent processor failures. In [7], the authors consider the case of systems subject to crash failures for processors and intermittent failures for links (only the loss case is considered). However, in their approach, bidirectional communication link is assumed to provide a lower level communication protocol that is reliable enough for their purpose. To some extent, topology changes can be considered as permanent failures on links. In this context, Super-stabilizing and Snap-stabilizing protocols (introduced in [18] and [10], respectively) are self-stabilizing protocols that also tolerate limited topology changes. Afek and Brown [2], consider self-stabilizing algorithms along with lossy communication links, but they assume bidirectional communications in order to build an underlying self-stabilizing data-link protocol. Finally, Hoepman *et al.* [22] consider the construction of wait-free objects in a self-stabilizing setting.

1.3. The Census Problem

The Census problem is derived from the Topology Update task by removing the location information requirement. Informally, a self-stabilizing distributed algorithm that solves the Census problem must ensure that eventually, the system reaches a global state where each processor knows the identifiers of all processors in the network and their relative distances to itself. Census information is sufficient to solve many fundamental tasks, such as leader election (the processor with minimum identifier is elected), counting all nodes in the system (the number of processors in the Census list), or topological sort of ancestors. Typically, a Topology Update algorithm would require each processor in the system to store each link of the communication graph (inducing a $\Omega(N^2)$ bits storage at each node, where N is the size of the network), while a Census algorithm would require each processor in

the system to store each node of the communication graph along with its relative distance (inducing a $\Omega(N \times \log_2 N)$ bits storage at each node).

A self-stabilizing solution (although not presented as such) to the problem of Topology Update has been proposed in [26]. Subsequent works on self-stabilization and the Topology Update problem include [25, 15, 18], but none of the aforementioned protocols consider intermittent link failures. Those algorithm typically use $O(\delta \times N^2)$ bits per node when links can be enumerated, where δ is the degree of the node, and $O(\delta \times N^2 \times \log_2(k))$ bits otherwise, where k is the number of possible identifiers for nodes. These works share two common points: (i) communication between neighboring nodes is carried out using reliable bidirectional links, and (ii) node are aware of their local topology. In the context of *directed* networks, both points (i) and (ii) are questionable:

1. If a bidirectional network is not available, then self-stabilizing data link protocols (that are acknowledgment based, such as those presented in [2]) cannot be used to transform any of those works so that they perform in unreliable message passing environments.
2. In directed networks, it is generally easy to maintain the set of input neighbors (by checking who has "recently" sent a message), but it is very difficult (if not impossible) to maintain the set of output neighbors (in a satellite network, a transmitter is generally not aware of who is listening to the information it communicates).

Two algorithms [12, 8] were previously presented for the Census problem in unidirectional networks. They are both self-stabilizing and assume the simple topology of a unidirectional ring: [8] assumes reliable communications and supports the efficient cut-through routing scheme, while [12] supports fair loss, finite duplication, and desequencing of messages.

1.4. Our Contribution

We extend the result of Delaët and Tixeuil [12] from unidirectional rings, which are a small subset of actual communication networks, to general strongly connected networks, that are a proper superset of unidirectional rings and bidirectional networks. However, we retain all link failure hypothesis of Delaët and Tixeuil [12]: fair loss, finite duplication, reordering. In more details, we present a self-stabilizing algorithm for the Census problem that tolerates message loss, duplication and reordering both in the stabilizing phase and in the stabilized phase. Our algorithm only assumes that the input neighborhood of each node is known, but not the output neighborhood, so that it can be used in a large class of actual systems. Our algorithm requires $O(N \times (\log_2(k) + \delta_i^-))$ bits per node, where k is the number of possible identifiers for nodes, and where δ_i^- is the input degree of node i. The stabilization time is $O(D)$, where D is the diameter of the network.

Using the scheme presented in [18] on top of our algorithm, we are then able to solve any global computation task (i.e., any task that can be solved by a silent system) in a self-stabilizing way, and still cope with unreliable communication links even when stabilized.

Although we assume the system communication graph is strongly connected, we do not use this information to build a well-known topology (e.g., a ring) and run a well-known algorithm on it. Indeed, this approach could potentially lower the performance of the overall algorithm, due to the fact that the communication possibilities are not used to their full extent. As a matter of fact, when our distributed algorithm is run on a network that is not strongly connected, we ensure that the collected information at each node is a topologically sorted list of its ancestors. In DAG (directed acyclic graph) structured networks, such kind of information is often wished (see [11]), and our approach makes it tolerant to link failures for free.

1.5. Outline

Section 2 presents a model for distributed systems we consider, as well as convenient notations used in the rest of the paper. Section 3 describes our self-stabilizing census algorithm on strongly connected networks, while Section 4 establishes its proof of correctness. Concluding remarks are proposed in Section 5.

2. PRELIMINARIES

2.1. Distributed Systems

In order to present a formal proof of our algorithm, we introduce a few definitions and notations that describe the model used in the rest of the paper.

A *processor* is a sequential deterministic machine that uses a local memory, a local algorithm and input/output capabilities. Intuitively, such a processor executes its local algorithm. This algorithm modifies the state of the processor memory, and send/receive messages using the communication ports.

An *unidirectional communication link* transmits messages from a processor o (for *origin*) to a processor d (for *destination*). The link is interacting with one input port of d and one output port of o. Depending on the way messages are handled by a communication link, several properties can be defined on a link. Lynch [24] proposes a complete formalization of these properties. We only enumerate those that are related to our algorithm. There is a *fair loss* when, infinitely messages are being emitted by o, infinitely messages are received by d. There is *finite duplication* when every message emitted by o may be received by d a finite number of times: however, a bound on the number of time a message was duplicated is *not* known to the processors. There is *reordering* when messages emitted by o may be received by d in a different order than that they were emitted. We also assume that any message that is not lost is eventually received by d. In particular, if the origin node o *continuously* send the same message infinitely, then this message is eventually received by the destination node d.

A *distributed system* is a 2-uple $\mathscr{S} = (P, L)$, where P is a set of processors and L is a set of communication links. A distributed system is represented by a directed graph whose nodes denote processors and whose directed edges (or arcs) denote communication channels (or links). The state of a processor can be reduced to the

state of its local memory, the state of a communication link can be reduced to its contents, then the global system state can be simply defined as

DEFINITION 1. A *configuration* of a distributed system $\mathscr{S} = (P, L)$ is the product of the states of memories of processors of P and of contents of communication links in L. The set of configurations is noted as $\mathscr{C}$.

Our system is not fixed once for all: it passes from a configuration to another when a processor executes an instruction of its local algorithm or when a communication link delivers a message to its destination. This sequence of reached configurations is called a *computation*.

DEFINITION 2. A *computation* of $\mathscr{S} = (P, L)$ is a maximal sequence of configurations of $\mathscr{S}$ noted as $C_1, C_2, \ldots$ and such that for any positive integer i, the transition from C_i to C_{i+1} is done through execution of an atomic action of every element of a non-empty subset of P and/or L. Configuration C_1 is called the *initial configuration* of the computation.

In the most general case, the specification of a problem is by enumerating computations that solve this problem. In the special case of the Census problem, where a global deterministic calculus is done, the specification can be given in terms of a set of system configurations.

DEFINITION 3. A configuration c satisfies the Census specification if and only if for any i in P, i knows the name and distance of all other elements relatively to itself in c.

A computation e satisfies the Census specification if and only if every configuration c in e satisfies the Census specification. A set of configurations $B \subset \mathscr{C}$ is *closed* if for any $b \in B$, any possible computation of System $\mathscr{S}$ whose b is initial configuration only contains configurations in B. A set of configurations $B_2 \subset \mathscr{C}$ is an *attractor* for a set of configurations $B_1 \subset \mathscr{C}$ if for any $b \in B_1$ and any possible computation of $\mathscr{S}$ whose initial configuration is b, the computation contains a configuration of B_2.

DEFINITION 4. A system $\mathscr{S}$ is self-stabilizing for a specification $\mathscr{A}$ if there exists a non-empty set of configurations $\mathscr{L} \subset \mathscr{C}$ such that:

Closure any computation of $\mathscr{S}$ whose initial configuration is in $\mathscr{L}$ satisfies $\mathscr{A}$,
Convergence $\mathscr{L}$ is an attractor for $\mathscr{C}$.

To show that a given system is self-stabilizing, it is sufficient to exhibit a particular non-empty subset of configurations of the system: the *legitimate configurations*. Then it is to be shown that any computation starting from a legitimate configuration satisfies the considered problem (closure property), and that from any possible configuration of the system, any possible computation of the system leads to a legitimate configuration (convergence property).

2.2. System Settings

Constants: Each node knows its unique identifier, which is placed in non-corruptible memory. We denote this identifier as an italics Latin letter. Each node i is aware of its input degree δ_i^- (the number of its incident arcs), which is also placed in non-corruptible memory. A node i arbitrarily numbers its incident arcs using the first δ_i^- natural numbers. When receiving a message, the node i knows the number of the corresponding incoming link (that varies from 1 to δ_i^-).

Local memory: Each node maintains a local memory. The local memory of i is represented by a list denoted by $(i_1; i_2; \ldots; i_k)$. Each i_x is a non-empty list of pairs $\langle identifier, colors \rangle$, where *identifier* is a node identifier, and where *colors* is an array of booleans of size δ_i^-. Each boolean in the *colors* array is either *true* (denoted by $\bullet$) or *false* (denoted by $\circ$). We assume that natural operations on boolean arrays, such as unary *not* (denoted by $\neg$), binary *and* (denoted by $\wedge$) and binary *or* (denoted by $\vee$) are available.

The goal of the Census algorithm is to guarantee that the local memory of each node contains the list of lists of identifiers (whatever be the *colors* value in each pair $\langle identifier, colors \rangle$) that are predecessors of i in the communication graph. For the Census task to be satisfied, we must ensure that the local memory of each node i can contain as many lists of pairs as necessary. We assume that a minimum of

$$(N-1) \times (\log_2(k) + \delta_i^-)$$

bits space is available at each node i, where N is the number of nodes in the system and k is the number of possible identifiers in the system (see Lemma 4).

For example,

$$((j, [\bullet \circ \circ]; q, [\circ \bullet \circ]; t, [\circ \circ \bullet])(z, [\bullet \bullet \bullet]))$$

is a possible local memory for node i, assuming that δ_i^- equals 3. From the local memory of node i, it is possible to deduce the knowledge that node i has about its ancestors. With the previous example, node j is a direct ancestor of i (it is in the first list of the local memory of i) and this information was carried through incoming channel number 1 (only the first position of the *colors* array relative to node j is *true*). Similarly, nodes q and t are direct ancestors of i and this information was obtained through incoming links 2 and 3, respectively. Then, node z is at distance 2 from i, and this information was received through incoming links numbered 1, 2, and 3.

Messages: Each node sends and receives messages. The contents of a message is represented by a list denoted by $(i_1; i_2; \ldots; i_k)$. Each i_x is a non-empty list of *identifiers*.

For example,

$$((i)(j; q; t)(z))$$

is a possible content of a message. It means that i sent the message (since it appears first in the message), that i believes that j, q, and t are the direct ancestors of i, and that z is an ancestor at distance 2 of i.

Notations: The *distance* from i to j is denoted by $d(i,j)$, which is the minimal number of arcs from i to j. We assume that the graph is *strongly connected*, so the distance from i to j is always defined. Yet, since the graph may not be bidirectional, $d(i,j)$ may be different from $d(j,i)$. The *age* of i, denoted by χ_i, is the greatest distance $d(j,i)$ for any j in the graph. The network *diameter* is then equal to

$$\max_i \chi_i = D.$$

3. SELF-STABILIZING CENSUS

3.1. Overview

Our algorithm can be seen as a knowledge collector on the network. The local memory of a node then represents the current knowledge of this node about the whole network. The only certain knowledge a node may have about the network is local: its identifier, its incoming degree, the respective numbers of its incoming channels. This is the only information that is stored in non-corruptible memory.

The algorithm for each node consists in updating in a coherent way (according to its constant information, see Section 2.2) its knowledge upon receipt of other processors' messages, and communicating this knowledge to other processors after adding its constant information about the network. More precisely, each information placed in a local memory is related to the local name of the incoming channel that transmitted this information. For example, node i would only emit messages starting with singleton list (i) and then not containing i since it is trivially an ancestor of i at distance 0. Coherent update consists of three kinds of actions: the first two being trivial coherence checks on messages and local memory, respectively.

Check message coherence: Since all nodes have the same behavior, when a node receives a message that does not start with a singleton list, the message is trivially considered as erroneous and is ignored. For example, messages of the form $((j;q;t)(k)(m;y)(p;z))$ are ignored.

Check local coherence: Regularly and at each message receiving, a node checks for local coherence. We only check here for trivial inconsistencies (see the *problem*( ) helper function): a node is incoherent if there exist at least one pair $\langle$*identifier*, *colors*$\rangle$ such that *colors*$= [\circ \cdots \circ]$ (which means that some information in the local memory was not obtained from any of the input channels). If a problem is detected upon time-out, then the local memory is reinitialized, else if a problem is detected upon a message receipt, the local memory is completely replaced by the information contained in the message.

Trust most recent information: When a node receives a message through an incoming channel, this message should contain an information that was constructed after its own and then more reliable. The node removes all previous information obtained through this channel from its local memory. Then it integrates new information and only keeps old information (from its other incoming channels) that does not clash with new information.

Example: Assume that a message $mess = ((j)(k;\ l)(m)(p;\ q\,;r;\ i))$ is received by node i through its incoming link 1 and that $\delta_i^- = 2$. The following information can be deduced:

1. j is a direct ancestor of i (it appears first in the message),
2. k and l are ancestors at distance 2 of i and may transmit messages through node j,
3. m is an ancestor at distance 3 of i,
4. p, q and r are ancestors at distance 4 of i, j obtained this information through m.

This information is compatible with a local memory of i such as

$$((j,[\bullet\circ];q,[\circ\bullet])(k,[\bullet\circ];\ e,[\circ\bullet];\ w,[\circ\bullet])(m,[\circ\bullet];y,[\bullet\bullet])(p,[\bullet\circ];z,[\circ\bullet];h,[\bullet\circ]))$$

Upon receipt of message *mess* at i, the following operations take place: (i) the local memory of i is cleared from previous information coming from link 1, (ii) the incoming message is "colored" by the number of the link (here each identifier α in the message becomes a pair $\alpha,[\bullet\circ]$ since it is received by link number 1 and *not* by link number 2), and (iii) the local memory is enriched as in the following (where "$\leftarrow$" denotes information that was acquired upon receipt of a message, and where "$\rightarrow$" denotes information that is to be forwarded to the node output links):

$$\begin{array}{cccccc}
 & ((q,[\circ\bullet]) & (e,[\circ\bullet];w,[\circ\bullet]) & (m,[\circ\bullet];y,[\circ\bullet]) & (z,[\circ\bullet]))\\
\leftarrow\Big(& (j,[\bullet\circ]) & (k,[\bullet\circ];l,[\bullet\circ]) & (m,[\bullet\circ]) & \begin{pmatrix} p,[\bullet\circ];q,[\bullet\circ];\\ r,[\bullet\circ];i,[\bullet\circ]\end{pmatrix}\Big)\\
\rightarrow\Big(& (j,[\bullet\circ];q,[\circ\bullet]) & \begin{pmatrix} k,[\bullet\circ];e,[\circ\bullet];\\ w,[\circ\bullet];l,[\bullet\circ]\end{pmatrix} & (m,[\bullet\bullet];y,[\circ\bullet]) & \begin{pmatrix} p,[\bullet\circ];z,[\circ\bullet];\\ q,[\bullet\circ];r,[\bullet\circ]\end{pmatrix}\Big)
\end{array}$$

This message enabled the modification of the local memory of node i in the following way: l is a new ancestor at distance 2. This was acquired through incoming link number 2 (thus through node j). Nodes m and y are confirmed to be ancestors at distance 3, but *mess* sends information via nodes j and q, while y only transmits its information via node q. Moreover, q and r are part of the new knowledge of ancestors at distance 4. Finally, although i had information about h ($h,[\bullet\circ]$) before receiving *mess*, it knows now that the information about h is obsolete.

3.2. Communication Issues

The property of resilience to intermittent link failures of our algorithm is mainly due to the fact that each message is self-contained and independently moves towards

a complete correct knowledge about the network. More specifically

1. The fair loss of messages is compensated by the infinite spontaneous retransmission by each processor of their current knowledge.
2. The finite duplication tolerance is due to the fact that our algorithm is *idempotent* in the following sense: if a processor receives the same message twice from the same incoming link, the second message does not modify the knowledge of the node.
3. The desequencing can be considered as a change in the relative speeds of two messages towards a complete knowledge about the network. Each message independently becomes more accurate and complete, so that their relative order is insignificant. A formal treatment of this last and most important part can be found in Section 4.

3.3. Helper Functions

We now describe helper functions that will enhance readability of our algorithm. Those functions operate on lists, integers and pairs $\langle$*identifier*, *colors*$\rangle$. The specifications of those functions use the following notations: l denotes a list of identifiers, p denotes an integer, lc denotes a list of pair $\langle$*identifier*, *colors*$\rangle$, Ll denotes a list of lists of identifiers, and Llc denotes a list of lists of pairs $\langle$*identifier*, *colors*$\rangle$.

We assume that classical operations on generic lists are available: $\setminus$ denotes the binary operator "minus" (and returns the first list from which the elements of the second have been removed), $\cup$ denotes the binary operator "union" (and returns the list without duplicates of elements of both lists), $+$ denotes the binary operator "concatenate" (and returns the list resulting from concatenation of both lists), $\sharp$ denotes the unary operator that returns the number of elements contained in the list, and [ ] takes an integer parameter p so that $l[p]$ returns a reference to the pth element of the list l if $p \leqslant \sharp l$ (in order that it can be used on the left part of an assignment operator ": ="), or expand l with $p - \sharp l$ empty lists and returns a reference to the pth element of the updated list if $p > \sharp l$.

clean$(lc, p) \rightarrow$ **list of couples** returns the empty list if lc is empty and a list of pairs lc_2 such that for each $\langle identifier_{lc}, colors_{lc}\rangle \in lc$, if $colors_{lc} \wedge \neg colors(p) \neq [\circ \cdots \circ]$, then $\langle identifier_{lc}, colors_{lc} \wedge \neg colors(p)\rangle$ is in lc_2, else $\langle identifier_{lc}, *\rangle$ is not in lc_2.

Example: assuming $\delta_i^- = 3$,

$clean((j, [\circ \bullet \circ]; q, [\circ \circ \bullet]; k, [\circ \bullet \circ]; e, [\circ \circ \bullet]; w, [\circ \circ \bullet]; m, [\circ \circ \bullet]; y, [\circ \bullet \bullet]; p, [\circ \bullet \circ]; z, [\circ \circ \bullet]; h, [\circ \bullet \circ]), 2)$

$= (q, [\circ \circ \bullet]; e, [\circ \circ \bullet]; w, [\circ \circ \bullet]; m, [\circ \circ \bullet]; y, [\circ \circ \bullet]; z, [\circ \circ \bullet])$.

colors$(p) \rightarrow$ **array of booleans** returns the array of booleans that corresponds to the pth incoming link, *i.e.* the array that is such that $[\underbrace{\circ \cdots \circ}_{p-1 \text{ times}} \bullet \underbrace{\circ \cdots \circ}_{\delta_i^- - p \text{ times}}]$.

Example: assuming $\delta_i^- = 3$, $colors(2) = [\circ \bullet \circ]$.

emit(i, Llc) sends the message resulting from $(i)+identifiers(Llc)$ on every outgoing link of i.

identifiers$(Llc) \rightarrow$**list of list of identifiers** returns the empty list if Llc is empty and returns a list Ll of list of identifiers (such that each pair $\langle$*identifier*,*colors*$\rangle$ in Llc becomes *identifier* in Ll) otherwise.

Example: assuming $\delta_i^- = 3$,

$identifiers((j, [\circ\bullet\circ]; q, [\circ\circ\bullet])(k, [\circ\bullet\circ]; e, [\circ\circ\bullet]; w, [\circ\circ\bullet])(q, [\circ\bullet\circ])(k, [\circ\circ\bullet]; p, [\circ\bullet\circ]; j, [\circ\bullet\circ]))$

$= ((j; q)(k; e; w)(q)(k; p; j))$.

merge$(lc, l, p) \rightarrow$ **list of couples** returns the empty list if lc and l are both empty and

$$\bigcup_{\substack{\langle i,c\rangle \in lc \\ i \in l}} (\langle i, c \vee colors(p)\rangle) \cup \bigcup_{\substack{\langle i,*\rangle \notin lc \\ i \in l}} (\langle i, colors(p)\rangle)$$

otherwise.

Example: assuming $\delta_i^- = 3$,

$merge((j, [\circ \bullet \circ]; q, [\circ \circ \bullet]; k, [\circ \bullet \circ]; e, [\circ \circ \bullet]; w, [\circ \circ \bullet]), (q; k; p; j), 1)$

$= (j, [\bullet \bullet \circ]; q, [\bullet \circ \bullet]; k, [\bullet \bullet \circ]; e, [\circ \circ \bullet]; w, [\circ \circ \bullet]; p, [\bullet \circ \circ])$.

new$(lc, l) \rightarrow$ **list of couples** returns the empty list if lc is empty and the list of pairs $\langle$*identifier*, *colors*$\rangle$ whose *identifier* is not in l otherwise.

Example: assuming $\delta_i^- = 3$,

$new((j, [\circ \bullet \circ]; q, [\circ \circ \bullet]; k, [\circ \bullet \circ]; e, [\circ \circ \bullet]; w, [\circ \circ \bullet]), (i; j; e))$

$= (q, [\circ \circ \bullet]; k, [\circ \bullet \circ]; w, [\circ \circ \bullet])$.

problem$(Llc) \rightarrow$**boolean** returns *true* if there exist two integers p and q such that $p \leq \sharp(Llc)$ and $q \leq \sharp(Llc[p])$ and $Llc[p][q]$ is of the form $\langle$*identifier*, *colors*$\rangle$ and all booleans in *colors* are *false* ($\circ$). Otherwise, this function returns *false*.

3.4. The Algorithm

In addition to its local memory, each node makes use of the following local variables when processing messages: α is the current index in the local memory and message main list, *i_pertinent* is a boolean that is *true* if the αth element of the local memory main list contains pertinent information, *m_pertinent* is a boolean that is *true* if the αth element of the message main list contains pertinent information, *known* is the list of all identifiers found in the local memory and message found up to rank α, *temp* is a temporary list that stores the updated αth element of the local memory main list.

We are now ready to present our Census Algorithm (noted $\mathscr{CA}$ in the remaining of the paper). This algorithm is message driven: processors execute their code when they receive an incoming message. In order to perform correctly in configurations where no messages are present, Algorithm $\mathscr{CA}$ also uses a spontaneous action that will emit a message.

Spontaneously, a node i runs the following code:

IF *problem*(*local_memory*) THEN *local_memory* := ( ) ENDIF
emit(*i*,*local_memory*)

Upon receipt of a message named *message* from incoming link number p, a node i whose local memory is denoted by *local_memory* runs the following code:

Check for local memory coherence:
i_pertinent:= NOT *problem*(*local_memory*)

Check for message coherence:
m_pertinent:= (♯(*message*[1]) = 1)

Update local memory:
α := 0;*known*:= (*i*);
WHILE *m_pertinent* OR *i_pertinent* DO
  α := α + 1;*temp*:= ( )
  local_memory[α] := *clean*(*local_memory*[α], *p*)
  IF *i_pertinent* THEN
    temp:= *new*(*local_memory*[α],*known*)
    IF *temp* = ( ) THEN *i_pertinent*:= FALSE ENDIF
  ENDIF
  IF *m_pertinent* THEN
    IF *message*[α]*known*= ( ) THEN
      m_pertinent:= FALSE
    ELSE
      temp:= *merge*(*temp*,*message* [α]*known*, *p*)
    ENDIF
  ENDIF
  IF *temp*≠( ) THEN
    local_memory[α] := *temp*
    known:=*known*∪ *identifiers*(*temp*)
  ENDIF
ENDWHILE

Truncate local memory up to position α:
local_memory := (*local_memory*[1],...,*local_memory*[α])

Emit message along with identifier:
emit(*i*,*local_memory*)

4. PROOF OF CORRECTNESS

In this section, we show that Algorithm $\mathscr{CA}$ is a self-stabilizing Census algorithm. In more details, independent of the initial configuration of network channels (non-infinitely full) and of the initial configuration of local memories of nodes, every node ends up with a local memory that reflects the contents of the network, even if unreliable communication media is used for the underlying communication between nodes.

4.1. Overview

First, we define a formal measure on messages that circulate in the network and on local memories of the nodes. This measure is either the distance between the current form of the message and its canonical form (that denotes optimal knowledge about the network), or between the current value of the local memories and their canonical form (when a node has a perfect knowledge about the network). We use this measure to compute the weight of a configuration.

Then, we show that after a set of emissions and receptions of messages, the weight of a configuration decreases. An induction shows that this phenomenon continues to appear and that the weight of a configuration reaches 0, i.e., a configuration where each message is correct and where each node has an optimal knowledge about the network. We also show that such a configuration (whose weight is 0) is stable when a message is emitted or received. According to the previous definitions, a configuration of weight 0 is a *legitimate configuration* after finite time.

These two parts prove, respectively, the convergence and closure of our algorithm, and establish its self-stabilizing property.

4.2. Legitimate Configurations

The Census problem being static and deterministic, when we only consider node local memories, there is a single legitimate configuration. This legitimate configuration exists when each node has a global correct knowledge about the network. The fact is also that the system would reach the stable configuration had it been started from a zero knowledge configuration (where the local memory of each node is null, and where no messages are in transit in the system).

In this legitimate configuration, all circulating messages are of the same kind, and the only difference between legitimate configurations is the number of messages in each communication link. This induces the following definitions of canonical messages and canonical local memory.

DEFINITION 5. The canonical form of a message circulating on a link between nodes j and i is the list of lists starting with the singleton list (j) followed by the χ_j lists of ancestors of j at distance between 1 and χ_j.

DEFINITION 6. The canonical form of node i's local memory is the list of lists of pairs *Llc* of the χ_i lists of pairs $\langle$*identifier*,*colors*$\rangle$ such that:

- *identifiers*(*Llc*) is the list of the χ_i lists of ancestors of i at distance 1 to χ_i.
- if a shortest path from node j to node i passes through the pth input channel of i, then the boolean array *colors* associated with node j in *Llc* has *colors*[p] = ●.

For the sake of simplicity, we will also call the αth list of a canonical message or a canonical local memory a *canonical list*.

4.3. Closure

PROPOSITION 1. *The canonical form of node i's local memory and that of its incoming and outgoing channels are coherent.*

Proof. If node i's local memory is in canonical form (according to Definition 6), then the *emit* action trivially produces a canonical message (according to Definition 5).

Conversely, upon receipt by node i of a canonical message through incoming link j, the local memory of i is replaced by a new identical canonical memory. Indeed, *clean* first removes from the αth list of i's local memory all pairs $\langle identifier, colors \rangle$ such that $colors = colors(p)$, yet by definition of canonical memory, each such *identifier* is that of a node such that the shortest path from *identifier* to i is of length α and passes through j. Moreover, the list l used by *merge* is the list of nodes at distance $\alpha - 1$ of node i, so for any *identifier* appearing in l, two cases may occur:

1. There exists a path from *identifier* to i that is of length $< \alpha$; then $identifiers \in known$ and it does not appear in the new list of rank α.
2. There exists a shorter path from *identifier* to i through j of length α; then $\langle identifier, colors(p) \rangle$ is one of the elements that was removed by *clean* and this information is put back into node i's local memory. ∎

COROLLARY 1. *The set of legitimate configurations is closed.*

Proof. Starting from a configuration where every message and every local memory is canonical, none of the local memories is modified, and none of the emitted message is non-canonical. ∎

4.4. Configuration Weight

We define a weight on configurations as a function on system configurations that returns a positive integer. As configurations of weight zero are legitimate, the weight of a configuration c denotes the "distance" from c towards a legitimate configuration.

In order to evaluate the weight of configurations, we define a measure on messages and local memory of nodes as an integer written using $D + 2$ digits in base 3 (where D denotes the graph diameter). The weight of a configuration is then the pair of the maximum weight of local memories, and the maximum weight of circulating messages. For the sake of clarity, a single integer will denote the weight of the configuration when both values are equal. Note that since a canonical message is of size $\leqslant D + 1$, we have $m\_canonical[D + 2] = ( )$.

DEFINITION 7. Let *mess* be a circulating message on a communication link whose canonical message is denoted by *m_canonical*. The weight of *mess* is the integer written using $D + 2$ base 3 digits and whose αth digit is

- 0, if $mess[\alpha] = m\_canonical[\alpha]$,
- 1, if $mess[\alpha] \subsetneq m\_canonical[\alpha]$,
- 2, if $mess[\alpha] \not\subseteq m\_canonical[\alpha]$.

Example: Assume that a link from j to i in a network of diameter 5 has a canonical message of the form $(j)(k; e; w)(q)(i; p)$. The following messages circulating on this channel will have the following weights:

(j)	$(k; e; w)$	(q)	$(i; p)$				Overall a weight of 0
0	0	0	0	0	0	0	
(j)	$(k; e; w)$	$(q; d)$	$(i; p)$	(z)			Overall a weight $\leqslant 3^5$
0	0	2	0	2	0	0	
(j)							Overall a weight $\leqslant 2 \times 3^5$
0	1	1	1	1	0	0	
(g)	$(h; t)$	(t)	$(i; d)$	(a)	(a)	(a)	Overall a weight of $3^7 - 1$
2	2	2	2	2	2	2	

Then, $3^{D+2} - 1$ is the biggest weight for a message, and corresponds to a message that is totally erroneous. At the opposite, 0 is the smallest weight for a message, and corresponds to a canonical message, or to a message that begins with a canonical message.

DEFINITION 8. Let *memo* be the local memory of a node i whose canonical local memory is *m_canonical*. The weight of *memo* is the integer written using $D + 1$ digits (in base 3) and whose α^{th} digit is

- 0, if $memo[\alpha] = m\_canonical[\alpha]$
- 1, if $memo[\alpha] \neq m\_canonical[\alpha]$ and $identifiers(memo[\alpha]) \subseteq identifiers \times (m\_canonical[\alpha])$ and for any $\langle identifier, colors_1 \rangle$ of $memo[\alpha]$, the associated $\langle identifier, colors_2 \rangle$ in $m\_canonical[\alpha]$ verifies: $(colors_1 \wedge colors_2) = colors_1$.
- 2, otherwise.

Then $3^{D+1} - 1$ is the biggest weight of a local memory and denotes a totally erroneous local memory. At the opposite, 0 is the smallest weight and denotes a canonical local memory.

Let us notice that in both cases (weight of circulating messages and of nodes local memories), the αth digit 0 associated with the αth list denotes that this particular list is in its final form (the canonical form). The αth digit 1 means that the αth list is coherent with the αth canonical list, but still lacks some information. On the contrary, the αth digit 2 signals that the related αth position contains information that shall not persist and that are thus unreliable. The weight of a message indicates how much of the information it contains is pertinent.

4.5. Convergence

After defining message weight and, by extension, configuration weights, we first prove that starting from an arbitrary initial configuration, only messages of weight lower or equal to $3^{D+1} - 1$ are emitted, which stands for the base case for our induction proof.

LEMMA 1. *In any configuration, only messages of weight lower than* 3^{D+1} *may be emitted.*

Proof. Any message that is emitted from a node i on a link from i to j is by function *emit*. This function ensures that this message starts with the singleton list (i).

This singleton list is also the first element of the canonical message for this channel. Consequently, the biggest number that may be associated with a message emitted by node i starts with a 0 and is followed by $D+1$ digits equal to 2. Its overall weight is at most $3^{D+1}-1$. ∎

LEMMA 2. *Assume $\alpha \geqslant 1$. The set of configurations whose weight is strictly lower than $3^{\alpha-1}$ is an attractor for the set of configurations whose weight is strictly lower than 3^{α}.*

Proof. A local memory of weight strictly lower that 3^{α} contains at most α erroneous lists, and it is granted that it starts with $D+2-\alpha$ canonical lists.

By definition of the *emit* function, each node i that owns a local memory of weight strictly below 3^{α} shall emit the singleton list (i) followed by $D+2-\alpha$ canonical lists. Since canonical messages sent by a node and its canonical local memory are coherent, it must emit messages that contain at least $D+2-\alpha+1$ canonical lists, which means at worst $\alpha-1$ erroneous lists. The weight of any message emitted in such a configuration is then strictly lower than $3^{\alpha-1}$.

It follows that messages of weight exactly 3^{α} which remain are those from the initially considered configuration. Hence they are in finite number. Such messages are either lost or received by some node in a finite time. The first configuration that immediately follows the receiving or loss of those initial messages is of weight (3^{α} (local memory), $3^{\alpha-1}$ (messages)).

The receiving by each node of at least one message from any incoming channel occurs in finite time. By the time each node receives a message, and according to the local memory maintenance algorithm, each node would have been updated. Indeed, the receiving of a message from an input channel implies the cleaning of all previous information obtained from this channel. Consequently, in the considered configuration, all lists in the local memory result from corresponding lists in the latest messages sent through each channel. Yet, all these latest messages have a weight strictly lower than $3^{\alpha-1}$ and by the coherence property on canonical forms, they present information that are compatible with the node canonical local memory, up to index $D+3-\alpha$. By the same property, and since all input channels contribute to this information, it is complete. In the new configuration, each node i maintains a local memory whose first $D+3-\alpha$ lists are canonical, and thus the weight of its local memory is $3^{\alpha-1}$. Such a configuration is reached within finite time and its weight is ($3^{\alpha-1}$ (local memory), $3^{\alpha-1}$ (messages)). ∎

PROPOSITION 2. *The set of configurations whose weight is 0 is an attractor for the set of all possible configurations.*

Proof. By induction on the maximum degree of the weight on configurations. The base case is proved by Lemma 1, and the induction step is proved by Lemma 2. Starting from any initial configuration whose weight is greater that 1, a configuration whose weight is strictly inferior is eventually reached. Since the weight of a configuration is positive or zero, and that the order defined on configurations weights is total, eventually a configuration whose weight is zero is eventually reached. By definition, this configuration is legitimate. ∎

THEOREM 1. *Algorithm $\mathscr{CA}$ is self-stabilizing.*

Proof. Consider a message m of weight 0. Two cases may occur: (i) m is canonical, or (ii) m starts with a canonical message, followed by at least one empty list, (possibly) followed by several erroneous lists. Assume that m is *not* canonical; then it is impossible that m was emitted, since the *truncate* part of Algorithm $\mathscr{CA}$ ensures that no message having an empty list can be emitted; then m is an erroneous message that was present in the initial configuration.

Similarly, the only local memories that may contain an empty list are those initially present (e.g., due to a transient failure).

As a consequence, after the receipt of a message by each node and after the receipt of all initial messages, all configurations of weight 0 are legitimate (they only contain canonical messages and canonical local memories).

By Proposition 2, the set of legitimate configurations is an attractor for the set of all possible configurations, and Corollary 1 proves closure of the set of legitimate configurations. Therefore, Algorithm $\mathscr{CA}$ is self-stabilizing. ∎

4.6. Complexity

In this section, we investigate the memory space and time needed for the system to stabilize into a legitimate configuration.

4.6.1. Space complexity. The space complexity result is immediately given by the assumptions made when writing our algorithm. In the following, N denotes the number of nodes in the system, and k denotes the number of possible identifiers for nodes. In practical systems, $\log_2(k)$ typically corresponds to a system word (32 or 64 bits).

LEMMA 3. *Each message m requires at least $N \times (\log_2(k))$ bits space.*

Proof. We compute the space needed by each message to hold all information in the Census algorithm. We do not take into account the implementation-dependent list coding of a message information. Each identifier (of size bounded by $\log_2(k)$) is present exactly once in each message, and there are N such identifiers. Overall, the required memory (in bits) at message m is bounded by

$$N \times \left(\log_2(k)\right). \quad \blacksquare$$

LEMMA 4. *Each node i requires at least*

$$(N-1) \times \left(\log_2(k) + \delta_i^-\right)$$

bits space, where δ_i^- denotes the input degree of the node i.

Proof. We compute the space needed at node i to hold all information in the Census algorithm. We do not take into account the implementation-dependent list coding of a node local information. Each identifier (that is bounded by $\log_2(k)$) in a pair is associated with an boolean array, that represents the incoming links that

transmitted the identifier. This array requires δ_i^- bits. In a correct configuration, node i has a pair $\langle identifier, colors \rangle$ for any other node in the network (thus $N-1$ pairs). Overall, the required memory (in bits) at node is bounded by

$$(N-1) \times \left(\log_2(k) + \delta_i^-\right)$$

where δ_i^- denotes the input degree of the node i. ∎

4.6.2. Time complexity. In the convergence part of the proof, we only assumed that computations were maximal, and that message loss, duplication and desequencing could occur. In order to provide an upper bound on the stabilization time for our algorithm, we assume strong synchrony between nodes and a reliable communication medium between nodes. Note that these assumptions are used for complexity results only, since our algorithm was proven correct even in the case of asynchronous unfair computations with link intermittent failures. In the following D denotes the network diameter.

LEMMA 5. *Assuming a synchronous reliable system $\mathscr{S}$, the stabilization time of algorithm $\mathscr{C}\mathscr{A}$* is $O(D)$.

Proof. Since the network is synchronous, we consider *system steps* as: (i) each node receives all messages that are located at each of its incoming links and updates its local memory according to the received information, and (ii) each node sends as many messages as received on each of its outgoing links. Intuitively, within one system step, each message is received by one processor and sent back. Within one system step, all messages are received, and messages of weight strictly inferior to that of the previous step are emitted (see the proof of Lemma 2). In the same time, when a processor has received messages from each of its incoming links, its weights is bounded by $3^{D+1-\alpha}$, where D is the network diameter, and α is the number of the system step (see the proof of Lemma 2). Since the maximal initial weight of a message and of a local memory is 3^{D+2}, after $O(D)$ system steps, the weight of each message and of each local memory is 0, and the system has stabilized. ∎

5. CONCLUSION

When a distributed system is subject to various kinds of failures, various ways of ensuring recovery from those failures are to be considered. We presented a global approach that allows to solve the Census problem while tolerating two usually distinguished kinds of failures: transient memory failures, and intermittent link failures. Unlike previous work, we did not specifically address the problems related to intermittent link failures. More precisely, in the proof of correctness, we presented a global weight function, and showed that in any system computation, its value was strictly decreasing up to the point when it reached 0. In this final stage, a key property related to idempotency (the receipt of a correct message by a correct node does not modify its state, a correct node always sends the same correct message) hints at a possible general condition for tolerating both transient and intermittent link failures.

We considered the Census problem in a strongly connected graph. However, the very same algorithm can be used for different purposes. In [18], Dolev and Herman show that starting from an algorithm that simply collects node unique identifiers, communicating other local information as well as the node identifier leads to solutions for any silent task (see [17]) using the same underlying Census algorithm. For example, storing local topology information enables the construction of a Topology Update algorithm. In our context, mixing the approach of Dolev and Herman [18] and ours would lead to a self-stabilizing Topology Update algorithm that also supports link intermittent failures.

Although we assume the system communication graph to be strongly connected, we do not use this information to build a well-known topology (e.g., a ring) and run a well-known algorithm on it. Indeed, this approach could potentially lower down the performance of the overall algorithm, due to the fact that the communication possibilities are not used to their full extent. As a matter of fact, when our distributed algorithm is run on a network that is not strongly connected, we ensure that the collected information at each node is a topologically sorted list of its ancestors. In DAG (directed acyclic graph) structured networks, such kind of information is often wished (see [11]), and our approach makes it tolerant to link failures for free.

ACKNOWLEDGMENT

We are grateful to the anonymous referees for helping us to improve this paper, both in presentation and in technical correctness.

REFERENCES

1. Y. Afek and A. Bremler, Self-stabilizing unidirectional network algorithms by power supply, *Chicago J. Theoret. Comput. Sci.* **4**(3) (1998), 1–48.
2. Y. Afek and G. M. Brown, Self-stabilization over unreliable communication media, *Distrib. Comput.* **7** (1993), 27–34.
3. Y. Afek, S. Kutten, and M. Yung, Memory-efficient self-stabilization on general networks, *in* "WDAG90 Distributed Algorithms 4th International Workshop Proceedings," Lecture Notes in Computer Science, Vol. 486, pp. 15–28, Springer-Verlag, Berlin, 1990.
4. E. Anagnostou and V. Hadzilacos, Tolerating transient and permanent failures, *in* "Proceedings of WDAG'93," Lecture Notes in Computer Science, Vol. 725, pp. 174–188, Springer-Verlag, Berlin, 1993.
5. B. Awerbuch, B. Patt-Shamir, and G. Varghese, Self-stabilization by local checking and correction, *in* "Proceedings of the 32nd Annual Symposium on Foundations of Computer Science," pp. 268–277, San Juan, P. R., IEEE Computer Society Press, New York, October 1991.
6. K. A. Bartlett, R. A. Scantlebury, and P. T. Wilkinson, A note on reliable full-duplex transmission over half-duplex links, *Comm. Assoc. Comput. Mach.* **12**(5) (May 1969), 260–261.
7. A. Basu, B. Charron-Bost, and S. Toueg, Simulating reliable links in the presence of process crashes. *in* "Proceedings of the 10th International Workshop on Distributed Algorithms (WDAG'96)," Lecture Notes in Computer Science, Vol. 1151, Springer-Verlag, Berlin, 1996.
8. J. Beauquier, A. K. Datta, and S. Tixeuil, Self-stabilizing census with cut-through constraint, *in* "Proceedings of the 3rd Workshop on Self-Stabilizing Systems (published in association with

ICDCS99, The 19th IEEE International Conference on Distributed Computing Systems),'' pp. 70–77, IEEE Computer Society, New York, 1999.

9. J. Beauquier and S. Kekkonen-Moneta, Fault tolerance and self-stabilization: Impossibility results and solutions using self-stabilizing failure detectors, *Int. J. Systems Sci.* **28**(11) (November 1997), 1177–1187.
10. A. Bui, A. K. Datta, F. Petit, and V. Villain, State-optimal snap-stabilizing pif in tree networks, *in* ''Proceedings of the 4th Workshop on Self-stabilizing Systems,'' pp. 78–85, 1999.
11. S. K. Das, A. K. Datta, and S. Tixeuil, Self-stabilizing algorithms in dag structured networks, *Parallel Process. Lett.* **9**(4) (December 1999), 563–574.
12. S. Delaët and S. Tixeuil, Un algorithme auto-stabilisant en dépit de communications non fiables, *Tech. Sci. Inform.* **5**(17) (1988), 613–633.
13. S. Delaët and S. Tixeuil, Tolerating transient and intermittent failures, *in* ''Proceedings of OPODIS'2000,'' pp. 17–36, December 2000.
14. E. W. Dijkstra, Self-stabilization in spite of distributed control, *Comm. Assoc. Comput. Mach.* **17**(11) (November 1974), 643–644.
15. S. Dolev, Self-stabilizing routing and related protocols, *J. Parallel Distrib. Comput.* **42**(2) (May 1997), 122–127.
16. S. Dolev, ''Self-Stabilization,'' MIT Press, Cambridge, MA, 2000.
17. S. Dolev, M. G. Gouda, and M. Schneider, Memory requirements for silent stabilization, *in* ''PODC96 Proceedings of the 15th Annual ACM Symposium on Principles of Distributed Computing,'' pp. 27–34, 1996.
18. S. Dolev and T. Herman, Superstabilizing protocols for dynamic distributed systems, *Chicago J. Theoret. Comput. Sci.* **3**(4) (1997).
19. S. Dolev, A. Israeli, and S. Moran, Self-stabilization of dynamic systems assuming only read/write atomicity, *Distrib. Comput.* **7** (1993), 3–16.
20. M. J. Fisher, N. A. Lynch, and M. S. Paterson, Impossibility of distributed consensus with one faulty process, *J. Assoc. Comput. Mach.* **32**(2) (April 1985), 374–382.
21. A. S. Gopal and K. J. Perry, Unifying self-stabilization and fault-tolerance, *in* ''Proceedings of PODC'93,'' pp. 195–206, 1993.
22. J.-H. Hoepman, M. Papatriantafilou, and P. Tsigas, Self-stabilization of wait free shared memory objects, *in* ''Proceedings of the WDAG'95,'' pp. 273–287, 1995.
23. M. Jayaram and G. Varghese. Crash failures can drive protocols to arbitrary states, *in* ''PODC96 Proceedings of the 15th Annual ACM Symposium on Principles of Distributed Computing,'' pp. 247–256, 1996.
24. N. A. Lynch, ''Distributed Algorithms,'' Morgan Kaufmann, Los Altos, CA, 1996.
25. T. Masuzawa, A fault-tolerant and self-stabilizing protocol for the topology problem, *in* ''Proceedings of the Second Workshop on Self-Stabilizing Systems,'' pp. 1.1–1.15, 1995.
26. J. M. Spinelli and R. G. Gallager, Event driven topology broadcast without sequence numbers, *IEEE Trans. Comm.* **37**(5) (May 1989), 468–474.
27. N. V. Stenning, A data transfer protocol, *Computer Networks*, **1**(2) (September 1976), 99–110.
28. D.-W. Wang and L. D. Zuck, Tight bounds for the sequence transmission problem, *in* ''Proceedings of the 8th Annual ACM Symposium on Principles of Distributed Computing (PODC'89),'' pp. 73–83, August 1989.

SYLVIE DELAËT has been a *Docteur en Sciences de l'Université Paris Sud*, Orsay, France since December 1995. Her Ph.D. was on self-stabilizing mutual exclusion. She has been a *Maître de conférences* at the University Paris Sud since September 1996. From 1995 to 2001 she was doing research in the self-stabilizing area. Her research interests include distributed computing, communication networks and fault tolerance.

SÉBASTIEN TIXEUIL received the *Magistère d'Informatique Appliquée* from the University Pierre and Marie Curie (France) in 1995, and his M.Sc and Ph.D. in computer science from the University of Paris Sud (France) in 1995 and 2000, respectively. In 2000, he joined the faculty at the University Paris Sud. His research interests include self-stabilizing and fault-tolerant distributed computing.

A.4 Obtenir l'auto-stabilisation gratuitement

Self-stabilization with r-operators revisited. Sylvie Delaët, Bertrand Ducourthial, and Sébastien Tixeuil. *Journal of Aerospace Computing, Information, and Communication (JACIC)*, 3(10) :498-514, 2006.

– *Résumé :* Cet article décrit un algorithme paramétré applicable à n'importe quelle topologie orientée. Cet algorithme tolère des défaillances transitoires qui corrompent les mémoires des processeurs et des liens de communication (il est auto-stabilisant) ainsi que les défaillances intermittentes (perte équitable, réordonnancement, duplication finie de messages) sur les média de communication. Une preuve formelle établit sa correction pour le problème considéré. La fonction paramètre de notre algorithme peut être instanciée pour produire des algorithmes répartis pour plusieurs applications fondamentales et de haut niveau, comme le calcul des plus courts chemins, et la construction d'un arbre en profondeur. Grâce aux propriétés de recouvrement de fautes de notre algorithme, les protocoles résultants sont auto-stabilisants sans surcoût.

Self-Stabilization with r-operators revisited

Sylvie Delaët[1] Bertrand Ducourthial[2] Sébastien Tixeuil[3]

[1]LRI – CNRS UMR 8623, Université Paris Sud, France
[2]Heudiasyc – UMR CNRS 6599, UTC, Compiègne, France
[3]LRI – CNRS & INRIA Grand Large, Université Paris Sud, France

A preliminary version of this work appears in [6]. The final version of this work can be found in [7].

Abstract

We present a generic distributed algorithm for solving silents tasks such as shortest path calculus, depth-first-search tree construction, best reliable transmitters, in directed networks where communication may be only unidirectional. Our solution is written for the asynchronous message passing communication model, and tolerates multiple kinds of failures (transient and intermittent).

First, our algorithm is self-stabilizing, so that it recovers correct behavior after finite time starting from an arbitrary global state caused by a transient fault. Second, it tolerates fair message loss, finite message duplication, and arbitrary message reordering, during both the stabilizing phase and the stabilized phase. This second property is most interesting since, in the context of unidirectional networks, there exist no self-stabilizing reliable data-link protocol. The correctness proof subsumes previous proofs for solutions in the simpler reliable shared memory communication model.

1 Introduction

Historically, research in self-stabilization over general networks has mostly covered undirected networks where bidirectional communication is feasible and carried out using shared registers (see [8]). This model permits algorithm designers to write elegant algorithms and proofs. To actually implement such self-stabilizing algorithms in real systems, where processors communicate by exchanging messages, transformers that preserve the self-stabilizing property of the original algorithm are needed. Such transformers are presented in [2, 8], and are based on variants of the alternating bit protocol or the sliding window protocol. A common drawback to these transformers is that they require the receiver of a message to be able to send acknowledgments to the emitter periodically, so that the underlying message passing network must be bidirectional for the transformer to be correct.

Hence, in directed networks, acknowledgment-based transformers cannot be used to run self-stabilizing algorithms in message passing networks, since it is possible that there exist two neighbors in the network that are only connected through a unidirectional link. Moreover, in directed message passing networks, it is generally easy to maintain the set of input neighbors (by checking who has "recently" sent a message), but it is very difficult (if not impossible) to maintain the set of output neighbors. For instance, in a satellite or a sensor network, a

hal-00632413, version 1 - 14 Oct 2011

transmitter is generally not aware of who is listening to the information it communicates. Note also that wireless networks can be directed message passing networks, especially when power of emissions are not uniform: a node i can receive a message from j while the converse is not possible.

So, self-stabilizing algorithms that use implicit neighborhood knowledge to compare one node state with those of its neighbors and to check for consistency – a large subset of self-stabilizing algorithms – cannot be used in directed networks.

The particular system hypothesis and the lack of transformers has led authors to design specific self-stabilizing algorithms for directed networks [1, 4, 12, 5, 13, 10].

The solutions [1, 4, 5, 10] are "classical" in the sens that a self-stabilizing layer (or mechanism) is added to a well known (non-stabilizing) protocol to ensure stabilization. This typically induces a potential overhead (extra knowledge, variables, processing are needed). In contrast, [12, 13] are *condition based*: either the algorithm satisfies the condition (and is then stabilizing) or not (and is not stabilizing). So, no overhead is induced by adding the self-stabilizing property to the original algorithm (the original algorithm is not changed). The two solutions of [12, 13] are generic (they can solve multiple problem instances with a single parameterized algorithm), but perform in the unidirectional shared memory model. In [13], the atomicity of communication is composite: in one atomic step, a processor is able to read the actual state of all of its neighbors and update its state, while in [12], the atomicity is read-write: in one atomic step, a processor is able to read the state of one neighbor, or update its state, but not both. Both approaches cannot be transformed to perform in unidirectional message passing networks using known self-stabilizing transformers (see above). The two solutions of [4, 5, 10] are specific (a single problem is addressed, the routing problem in [4], the census problem in [5], and the group communication problem in [10]), but perform in directed message passing networks. While [4, 10] assume reliable communications (links do not lose, duplicate or reorder messages), [5] tolerates message loss, duplication, and reordering. [1] proposes a generic solution in the message passing model, but assumes that communications are reliable (with FIFO links), that nodes have unique identifiers, and that the network is strongly connected, three hypothesis that we do not make.

Our Contribution. In this paper, we concentrate on providing a generic algorithm (that can be instantiated to solve silent tasks, see [9]), that performs on general directed message passing networks. Our solution is not only self-stabilizing (it recovers in finite time from any initial global state), it also tolerates fair message loss, finite duplication, and arbitrary reordering both in the stabilizing and in the stabilized phase. Nice properties of our approach are that the network need not be strongly connected, and nodes need not know whether the network contains cycles, and no upper bound on the network size, diameter, or maximum degree. However, if such information is known, the stabilization time can be significantly reduced.

We provide, in more details, a parameterized algorithm that can be instantiated with a local function. Our parameterized algorithm enables a set of silent tasks to be solved self-stabilizingly, provided that these tasks can be expressed through local calculus operations called r-operators that operate over a set $\mathbb{S}$. The r-operators are general enough to permit applications such as shortest path calculus and depth-first-search tree construction on arbitrary graphs while remaining self-stabilizing.

The main differences between this paper and the most closely related work [12] are twofold.

Reference	Overhead	Atomicity	Reliability	Algorithm
[1]	yes	send/receive atomicity	reliable	generic (total order)
[4]	yes	send/receive atomicity	reliable	specific (routing)
[5]	yes	send/receive atomicity	unreliable	specific (census)
[10]	yes	send/receive atomicity	reliable	specific (group communication)
[13]	no	composite atomicity	reliable	generic (partial order on $\mathbb{S}$)
[12]	no	read/write atomicity	reliable	generic (total order on $\mathbb{S}$)
This paper	no	send/receive atomicity	unreliable	generic (total order on $\mathbb{S}$)

Figure 1: A summary of related self-stabilizing algorithms in directed networks.

First, we consider an unreliable message passing communication network, instead of a reliable shared memory system. As noted above, unidirectional read-write systems cannot be emulated in message passing networks by means of a known self-stabilizing transformer. The key difference is that shared registers may hold only the latest written value, while the communications links we consider may hold an unbounded number of (possibly erroneous) messages that can appear again once the network appears to have stabilized (due to the reordering assumption). Second, the proof technique that we use here is based on a completely different idea than that of [12]. In [12], it is first proved that a terminal configuration is eventually reached starting from any initial configuration, and then (using a complicated induction argument) that this terminal configuration is in fact legitimate. In contrast, in message passing networks, self-stabilizing systems cannot be terminating (otherwise deadlock situations could occur, see [16]), so the proof argument here is to prove the following two invariants: *(i)* the state of each processor is eventually lower than (or equal to) its legitimate state (in the sense of the order defined on $\mathbb{S}$), and *(ii)* the state of each processor is eventually greater than (or equal to) its legitimate state, so that the state of each processor is eventually legitimate. Not only is this new proof simpler and more elegant than that of [12], it also permits algorithm designers to abstract the communication media that is used, so that the same proof applies for shared memory and unreliable message passing systems.

In Figure 1, we capture the key differences between our protocol and the aforementioned related solutions ([1, 4, 12, 5, 13]) in general directed networks regarding the following criteria: communication, overhead, atomicity, reliability, and algorithm nature.

Outline. Section 2 presents a model for distributed systems we consider. Section 3 describes our self-stabilizing parameterized algorithm for general directed networks, along with our system hypothesis. Our main result is presented, and is illustrated by an example. The sketch of the proof of correctness is also given. Section 4 details the proof. An interesting point is that this proof subsumes previous proofs for solutions in the simpler reliabl shared memory model. In Section 5 we show how the very algebraic nature of our scheme makes it suitable for *ad hoc* and sensor wireless networks, considering the unreliable communication mechanisms that are provided in those networks. Concluding remarks are proposed in Section 6.

2 Model

Processors and links. Processors use *unidirectional communication links* to transmit messages from an origin processor o to a destination processor d. The link is interacting with one input port of d and one output port of o. A link may hold an arbitrary number of messages (although our algorithm also works for bounded capacity links). Depending upon the way

hal-00632413, version 1 - 14 Oct 2011

messages are handled by a communication link, several properties can be defined on a link. A complete formalization of these properties is proposed in [18]. We only enumerate those that are related to our algorithm. There is a *fair loss* when, infinitely many messages being emitted by o, infinitely many messages are received by d. There is *finite duplication* when every message emitted by o may be received by d a finite (yet unbounded) number of times. There is *reordering* when messages emitted by o may be received by d in a different order than that they were emitted. There is *eventual delivery* if any message that is not lost is eventually received (*i.e.* no message remains forever in a communication link).

Distributed system. A *distributed system* is a 2-tuple $\mathcal{S} = (\mathcal{P}, \mathcal{L})$ where $\mathcal{P}$ is the set of processors and $\mathcal{L}$ is the set of communication links. Such a system is modeled by a *directed graph* (also called *digraph*) $G = (V, E)$, defined by a set of vertices V and a set E of edges (v_1, v_2), which are ordered pairs of vertices of V ($v_1, v_2 \in V$). Each vertex u in V represents a processor P_u of system $\mathcal{S}$. Each edge (u, v) in E represents a communication link from P_u to P_v in $\mathcal{S}$. In the remainder of the paper, we use interchangeably processors, nodes, and vertices to denote processors, and links and edges to denote communication links. Also, we use the standard notation $A \setminus B$ to denote the set of elements that are in set A but nor in set B.

Graph notations. The *in-degree* of a vertex v of G, denoted by δv is equal to the number of vertices u such that the edge (u, v) is in E. The incoming edges of each vertex v of G are indexed from 1 to δv. A *directed path* P_{v_0,v_k} in a digraph $G(V, E)$ is an ordered list of vertices $v_0, v_1, \ldots, v_k \in V$ such that, for any $i \in \{0, \ldots, k-1\}$, (v_i, v_{i+1}) is an edge of E (*i.e.*, $(v_i, v_{i+1}) \in E$). The *length* of this path is k. If each v_i is unique in the path, the path is *elementary*. The set of all elementary paths from a vertex u to another vertex v is denoted by $\mathcal{X}_{u,v}$. A *cycle* is a directed path P_{v_0,v_k} where $v_0 = v_k$. The *distance* between two vertices u, v of a digraph G (denoted by $\mathrm{d}_G(u, v)$, or by $\mathrm{d}(u, v)$ when G is not ambiguous) is the minimum of the lengths of all directed paths from u to v (assuming there exists at least one such path). The *diameter* of a digraph G is the maximum of the distances between all couples of vertices in G between which a distance is defined. Finally, we denote as Γ_v^- (resp. Γ_v^+) the set of predecessors (resp. successors) of a vertex $v \in V$, that is the set of all vertices $u \in V$ such that there exists a path starting at u (resp. v) and ending at v (resp. u). The predecessors (resp. successors) u of v verifying $\mathrm{d}_G(u, v) = 1$ (resp. $\mathrm{d}_G(v, u) = 1$)) are called *direct-predecessors* (resp. *direct-successors*) and their set is denoted Γ_v^{-1} (resp. Γ_v^{+1}).

Configurations and executions. The global system state, called a *system configuration* (or simply *configuration*) and generally denoted c, is the union of (i) the states of memories of processors of $\mathcal{P}$ and (ii) the contents of communication links of $\mathcal{L}$. The set of configurations is denoted by $\mathcal{C}$. The part of information in a configuration $c \in \mathcal{C}$ related to the processors of $\mathcal{P}$ is denoted by $c_{|\mathcal{P}}$; the part related to a given processor $P \in \mathcal{P}$ is denoted by $c_{|P}$.

Starting from an *initial configuration* c_1, an *execution* $e_{c_1} = c_1, a_1, c_2, a_2, \ldots$ is a maximal alternating sequence of configurations and actions of such that, for any positive integer i, the transition from configuration c_i to configuration c_{i+1} is done through execution of action a_i. Maximal means that either the computation is infinite, or the computation is finite and no action is enabled in the final configuration. The notations $\mathcal{E}_c$, $\mathcal{E}_C$ and $\mathcal{E}$ denote respectively the set of all executions starting (i) from the initial configuration c, (ii) from any configuration

$c \in C \subset \mathcal{C}$, or (iii) from any configuration of $\mathcal{C}$ ($\mathcal{E}_\mathcal{C} = \mathcal{E}$). The ordered list $c_1, c_2, \ldots \in \mathcal{C}$ of the configurations of an execution $e = c_1, a_1, c_2, a_2 \ldots$ is denoted by $e_{|\mathcal{C}}$. In the rest of this paper, we adopt the following convention: if $c_i \in e_{|\mathcal{C}}$ appears before $c_j \in e_{|\mathcal{C}}$, then $i < j$.

Distributed algorithms resolve either static tasks (*e.g.*, distance computation) or dynamic tasks (*e.g.*, token circulation). The aim of static tasks is to compute a global result, which means that after a running time, processors always produce the same output (*e.g.*, the distance from a source). A static task is characterized by a final processor output o_P for any processor $P \in \mathcal{P}$, called *legitimate output*. A *legitimate configuration* c for this task satisfies $c_{|P} = o_P$ for any processor $P \in \mathcal{P}$. A distributed protocol designed for solving a given static task is correct if the distributed system $\mathcal{S}$ running this protocol reaches in finite time a legitimate configuration for this task.

Self-stabilization. A set of configurations $C \subset \mathcal{C}$ is *closed* if, for any $c \in C$, any possible execution $e_c \in \mathcal{E}_c$ of system $\mathcal{S}$ whose c is initial configuration only contains configurations in C. A set of configurations $C_2 \subset \mathcal{C}$ is an *attractor* for a set of configurations $C_1 \subset \mathcal{C}$ if, any execution $e_c \in \mathcal{E}_{C_1}$ contains a configuration of C_2. Let $C \subset \mathcal{C}$ be a non-empty set of configurations. A distributed system $\mathcal{S}$ is *C-stabilizing* if and only if C is a closed attractor for $\mathcal{C}$: any execution e of $\mathcal{E}$ contains a configuration c of C, and any further configurations in e reached after c remains in C. Finally, consider a static task for the distributed system $\mathcal{S}$, and let $L \subset \mathcal{C}$ be the set of the legitimate configurations of $\mathcal{S}$. A distributed protocol designed for solving this static task is *self-stabilizing* if the distributed system $\mathcal{S}$ running this protocol is L-stabilizing.

3 Parametric message passing $\mathcal{PA}$-MP algorithm

In this section, we first describe the distributed system we consider before defining the $\mathcal{PA}$-MP parametrized algorithm. We then introduce the r-operators, that are used as parameters. These operators are derivated from the associative, commutative and idempotent operators (such as the minimum on the integers).

3.1 System

Let $\mathcal{S} = (\mathcal{P}, \mathcal{L})$ be the distributed system we consider in the following. The associated graph composed of processors of $\mathcal{P}$ and communications links of $\mathcal{L}$ is fixed, directed and unknown to the processors of $\mathcal{P}$. Communications between processors are performed by message passing (directed message passing network).

Each processor v of $\mathcal{P}$ is endowed with a local real-time clock mechanism. However, those clocks are use for the sole purpose of being able to perform action based on some timeout mechanism, so our clocks are neither synchronized nor have bounded drift. Each processor v of $\mathcal{P}$ owns an incoming memory denoted as $\texttt{IN}_v$, which is supposed to be unalterable; this can be implemented by a ROM memory (*e.g.*, EPROM), or a memory that is regularly reloaded by any external process (human interface, captor, other independent algorithm, *etc.*). The value of this memory (that will never change) is called *initialization value*. For most provided applications, this initialization value is equal to the identity element of the set $\mathbb{S}$ (except for a limited set of predecessors, see below). Moreover, for each link, starting at processor $u \in \mathcal{P}$ and ending at processor v, there exists a corresponding incoming memory $\texttt{IN}_v^u$ in v, which is

used by v to store incoming messages sent by u. Note that $\texttt{IN}_v^u$ contains only one message. A processor v only stores the latest received message from u. In addition, processor v owns an output memory denoted by $\texttt{OUT}_v$. All these memories are private, and can only be read or written by v (note that v only reads $\texttt{IN}_v$, and only writes $\texttt{OUT}_v$). In the following, we identify the name of a memory with the value it contains. In the same way, a message is considered as equivalent to its value.

Processor v performs a calculation by applying an operator $\triangleleft$ (see § 3.3) on all of its incoming memories, and stores the result in its output memory $\texttt{OUT}_v$.

3.2 Algorithm

In this paper, we design a parameterized distributed protocol for Message Passing systems (denoted as $\mathcal{PA}$-MP). This protocol is composed of one local parameterized algorithm per processor v of $\mathcal{P}$, denoted by $\mathcal{PA}\text{-MP}|_{\triangleleft_v}$, where $\triangleleft_v$ is an operator used as a parameter (parameters could be slightly different on each processor, see Hypothesis 2).

This local algorithm calls three helper functions: $\texttt{Store}_v(m, u)$ stores in the local register $\texttt{IN}_v^u$ the contents of the message m; $\texttt{Evaluate}_v(\triangleleft_v)$ stores in the local register $\texttt{OUT}_v$ the result of the local computation $\triangleleft_v(\texttt{IN}_v, \texttt{IN}_v^{u_1}, \ldots, \texttt{IN}_v^{u_k})$ where $u_1, \ldots, u_k$ are direct predecessors of v ($\in \Gamma_v^{-1}$); $\texttt{Forward}_v$ sends $\texttt{OUT}_v$ to w for each processor $w \in \Gamma_v^{+1}$.

The local algorithm $\mathcal{PA}\text{-MP}|_\triangleleft$ on processor v is composed of two *guarded actions*, which are atomic sets of instructions (actions) executed when a pre-condition (guard) is fulfilled (see Figure 2).

$\mathcal{R}_1$ **Upon receipt of a message m sent by u:**
  if $m \neq IN_v^u$, **then**
    $\texttt{Store}_v(m, u)$
    $\texttt{Evaluate}_v(\triangleleft_v)$
    $\texttt{Forward}_v$
  end if

$\mathcal{R}_2$ **Upon timeout expiration:**
  $\texttt{Evaluate}_v(\triangleleft_v)$
  $\texttt{Forward}_v$
  reset the timeout

Figure 2: Local algorithm $\mathcal{PA}\text{-MP}|_{\triangleleft_v}$ on processor v.

The guard of Rule $\mathcal{R}_1$ is true when a message m from u is received, while the guard of Rule $\mathcal{R}_2$ makes use of a timeout mechanism. So, our algorihtm is both message-driven (an action is executed when a new message is received) and timeout-driven (an action is executed when a timeout expires). In message passing systems, timeouts is required for stabilization purposes since [16] proved that no self-stabilizing algorithm could exist in message passing systems if no kind of timeout mechanism is available. The reason for this impossibility result is that the system may start from an arbitrary global state where no messages are in transit, so if no node has a sending action that is triggered by a spontaneous timeout action, then

the system is deadlocked.

Rule $\mathcal{R}_2$ is also used in case of message loss. In a typical implementation of our algorithm in an actual system, the timeout mechanism should be tuned accordingly to the loss rate of the communication links, in order that not too many spontaneous messages are emitted, and that the stabilization time remains reasonable. Tuning this timeout is clearly beyond the scope of this paper.

3.3 r-operators

An *infimum* (hereby called an *s-operator*) $\oplus$ over a set $\mathbb{S}$ is an associative, commutative and idempotent binary operator. Such an operator defines a partial order relation $\preceq_\oplus$ over the set $\mathbb{S}$ by $x \preceq_\oplus y$ if and only if $x \oplus y = x$ and then a strict order relation $\prec_\oplus$ by $x \prec_\oplus y$ if and only if $x \preceq_\oplus y$ and $x \neq y$.

It is generally assumed that there exists a greatest element on $\mathbb{S}$, denoted by $e_\oplus$, and verifying $x \preceq_\oplus e_\oplus$ for every $x \in \mathbb{S}$. Hence, the $(\mathbb{S}, \oplus)$ structure is an *Abelian idempotent semi-group* with $e_\oplus$ as identity element. The prefix *semi* means that the structure cannot be completed to obtain a group, because the law $\oplus$ is idempotent (see [3]).

When parameterized by such an s-operator $\oplus$, the $\mathcal{PA}$-MP parametric local algorithm converges. However, some counter examples show that it is not self-stabilizing [12]. Consider a loop with a single node initialized with 1 and using the operator min. The output of the node should always be 1. Now suppose that a fault introduces a 0 in the output register of the node (which is sent to itself). Then the node will never produce the correct result.

In [11], a distorted algebra — the r-algebra — is proposed. This algebra generalizes the Abelian idempotent semi-group, and still allows convergence of wave-like algorithms: the three basic properties (associativity, commutativity, idempotency) defining the s-operators are generalized using a mapping (usually denoted r). For instance, the binary operator $\diamond$ defined on the integers by $x \diamond y = x + 2y$ is not associative. However we have $x \diamond (y \diamond z) = (x \diamond y) \diamond 2z = x \diamond y \diamond 2z = x + 2y + 4z$ and $\diamond$ is *r-associative* with the mapping $x \mapsto 2x$.

The following definition summarizes the conditions of existence of the r-operators. The first one (right identity element) is classical. Here, the structure is not necessarily commutative, and only a right identity element is required. The second one (weak left cancellation) is very useful for allowing some simplifications in structures that do not admit inverses (such as idempotent semi-groups). It can be interpreted as follows: if there exists no element x in the definition set that does not agree with the fact that $y = z$, then $y = z$. Almost all useful operators are weak left cancellative, including the laws of groups (eg. addition on the integers) and of semi-groups (eg. minimum on the integers).

Definition 1 *The binary operator $\triangleleft$ on $\mathbb{S}$ is an* r-operator *if there exists a surjective mapping r called r-mapping, such that the following conditions are fulfilled:*

(i) *right identity element:* $\exists e_\triangleleft \in \mathbb{S}, x \triangleleft e_\triangleleft = x$.

(ii) *weak left cancellation:* $\forall y, z \in \mathbb{S}, (\forall x \in \mathbb{S}, x \triangleleft y = x \triangleleft z) \Leftrightarrow (y = z)$

(iii) *r-associativity:* $\forall x, y, z \in \mathbb{S}, x \triangleleft (y \triangleleft z) = (x \triangleleft y) \triangleleft r(z)$;

(iv) *r-commutativity:* $\forall x, y \in \mathbb{S}, r(x) \triangleleft y = r(y) \triangleleft x$;

and ***(v)*** *r-idempotency:* $\forall x \in \mathbb{S}, r(x) \triangleleft x = r(x)$

For example, the operator $\text{minc}(x, y) = \min(x, y + 1)$ (for minimum and increment) is an r-operator on $\mathbb{Z} \cup \{+\infty\}$, with $+\infty$ its right identity element.

hal-00632413, version 1 - 14 Oct 2011

Given an r-operator $\triangleleft$, one can show that the r-mapping r is unique, and is an isomorphism of $(\mathbb{S}, \triangleleft)$. Moreover, the r-operator induces an s-operator on $\mathbb{S}$ by $x \triangleleft y = x \oplus r(y)$ (for instance, the r-operator minc induces the s-operator min). We also have $e_\oplus = e_\triangleleft$ and $r(e_\oplus) = e_\oplus$.

If no fault appears in the distributed system $\mathcal{S}$, our $\mathcal{PA}$-MP algorithm stabilizes when it is parameterized by any idempotent r-operator $\triangleleft$ [11]. Idempotent r-operators verify $x \preceq_\oplus r(x)$ for any $x \in \mathbb{S}$. This last property leads to the definition of *strict idempotency*, verified for instance by the r-operator minc:

Definition 2 *An r-operator $\triangleleft$ is* strictly idempotent *if, for any $x \in \mathbb{S} \setminus \{e_\oplus\}$, we have $x \prec_\oplus r(x)$.*

Note that, among others interesting properties, while it is not necessarily commutative, an r-operator $\triangleleft$ satisfies $\forall x, y, z \in \mathbb{S}, x \triangleleft y \triangleleft z = x \triangleleft z \triangleleft y$, which means that the result of the $\mathcal{PA}$-MP algorithm does not rely on any ordering of the neighborhood.

Finally, binary r-operators can be extended to accept any number of arguments. This is useful for our algorithm because a processor computes a result with one value per direct predecessor plus its own initialization value. An *n-ary r-operator* $\triangleleft$ consists in $n - 1$ binary r-operators based on the same s-operator, and we have, for any $x_0, \ldots, x_{n-1}$ in $\mathbb{S}$, $\triangleleft(x_0, \ldots, x_{n-1}) = x_0 \oplus r_1(x_1) \oplus \cdots \oplus r_{n-1}(x_{n-1})$. If all of these binary r-operators are (strictly) idempotent, the resulting n-ary r-operator is said (strictly) idempotent.

3.4 Hypotheses

In this section, we formalize some hypotheses, introduce some notations, and give basic lemmas that are used throughout the proofs.

Hypothesis 1 *In the distributed system $\mathcal{S}$, links may (fairly) lose, (finitely) duplicate, and (arbitrarily) reorder messages that are sent by neighboring processors. However, any message sent by u on the link (u, v) that is not lost is eventually received by v (*i.e. *no message may remain in a communication link forever).*

This is a weak hypothesis on link's reliability. However, the following lemma is immediate.

Lemma 1 *Let consider a communication link $(u, v) \in \mathcal{L}$. If the origin node u keeps sending the same message infinitely often, then this message is eventually received by the destination node v.*

Hypothesis 2 *In the distributed system $\mathcal{S}$ running the $\mathcal{PA}$-MP algorithm, any processor v runs the local algorithm defined in Figure 2 and parameterized by a strictly idempotent $(\delta v{+}1)$-ary r-operator. Moreover, all these r-operators are defined on the same set $\mathbb{S}$, and are based on the same s-operator $\oplus$, with $e_\oplus$ their common identity element.*

In other words, this hypothesis ensures a form of homogeneity in the distributed system we consider. The following lemma is a direct application of Hypothesis 2, Definition 1, and `Evaluate` function:

Lemma 2 *Let $\triangleleft_v$ be the r-operator used by processor v. Then the computation of the $\texttt{Evaluate}_v(\triangleleft_v)$ function can be rewritten as:*
$\triangleleft_v(\texttt{IN}_v, \texttt{IN}_v^{u_1}, \ldots, \texttt{IN}_v^{u_k}) = \texttt{IN}_v \oplus r_v^{u_1}(\texttt{IN}_v^{u_1}) \oplus \cdots \oplus r_v^{u_k}(\texttt{IN}_v^{u_k})$.

Hence, there is one r-mapping per communication link. We now define the composition of these mappings along a path ($\mathcal{X}_{u,v}$ denotes the set of all elementary paths from u to v).

Definition 3 *Let $P_{u_0,u_k} \in \mathcal{X}_{u_0,u_k}$ be a path from processor u_0 to processor u_k, composed of the edges (u_i, u_{i+1}) $(0 \leq i < k)$. Let r^i_{i+1}, $0 \leq i < k$, be the r-mapping associated to the link (u_i, u_{i+1}). The* r-path-mapping *of P_{u_0,u_k}, denoted by $r_{P_{u_0,u_k}}$, is defined by the composition of the r-mappings r^i_{i+1}, for $0 \leq i < k$: $r_{P_{u_0,u_k}} = r^{k-1}_k \circ \cdots \circ r^0_1$.*

Our proofs of correctness (Lemmas 7 and 12) assume that any result produced on a node with the $\texttt{Evaluate}_v(\lhd_v)$ function (see Lemma 2) is either the initial value of the node ($\texttt{IN}_v$) or one of its incoming value transformed by an r-mapping ($r^{u_i}_v(\texttt{IN}^{u_1}_v)$). For this purpose, we admit that the order $\preceq_\oplus$ defines a total order. Note that with stronger nodes synchronization, such hypothesis is not necessary (see [13], where a proof for composite atomicity in a shared memory model is given).

Hypothesis 3 *The order relation $\preceq_\oplus$ is a total order relation: $\forall x, y \in \mathbb{S}$, either $x \preceq_\oplus y$ or $y \preceq_\oplus x$.*

Since the order $\preceq_\oplus$ is total, when it is clear from the context, in the remaining of the paper, we use "x is smaller than y" (or "y is larger than x") to denote $x \preceq_\oplus y$.

Hypothesis 4 *The set $\mathbb{S}$ is either finite, or any strictly increasing infinite sequence of values of $\mathbb{S}$ is unbounded (except by $e_\oplus$).*

Assuming Hypothesis 3, Hypothesis 4 specifies that the values used in the distributed system $\mathcal{S}$ can be, for instance, integers but not reals. Note that truncated reals (as in any computer implementation) are also convenient. Hypotheses 2 and 4 give the following lemma:

Lemma 3 *The set $\mathbb{S}$ is either finite or any r-mapping r used in $\mathcal{S}$ verifies: $\forall x \in \mathbb{S} \setminus \{e_\oplus\}, r(x) \prec_\oplus e_\oplus$.*

Hypothesis 5 *Each processor v admits at least one predecessor $u \in \Gamma^-_v$ such that $\texttt{IN}_u \neq e_\oplus$, u is called a* non-null *processor.*

In the following, we denote by $\widehat{\texttt{OUT}}_v$ the legitimate output of processor v. Moreover, for any processor v, any predecessor u of v and any configuration c, we denote by $\texttt{OUT}_v(c)$ and $\texttt{IN}^u_v(c)$ the value of the memories $\texttt{OUT}_v$ and $\texttt{IN}^u_v$ in the configuration c.

3.5 Our result

Our protocol is dedicated to static tasks. Such tasks (*e.g.*, the distance computation from a processor u) are defined by one output per processor v (*e.g.*, the distance from u to v), which is the legitimate output of v. With our $\mathcal{PA}$-MP algorithm, this means that, after finite time, each processor $v \in \mathcal{P}$ should contain this output (*e.g.*, $d(u,v)$) in its outgoing memory $\texttt{OUT}_v$. To solve static tasks with the $\mathcal{PA}$-MP distributed algorithm, one must use an operator as parameter (*e.g.*, minc for distance computation) such that the distributed system $\mathcal{S}$ reaches the legitimate configurations and do not leave them thereafter (*i.e.*, any processor reaches and then conserves its legitimate output). In this paper, we prove that if the operator is used

hal-00632413, version 1 - 14 Oct 2011

to parameterize the $\mathcal{PA}$-MP distributed algorithm, then it is self-stabilizing, according to the hypotheses of § 3.4.

Let us define the legitimate outputs of the processor using the r-operators that parameterize the $\mathcal{PA}$-MP algorithm. For instance, to solve the distance computation problem, we state $\mathbb{S} = \mathbb{N} \cup \{+\infty\}$, and each local algorithm is parameterized by the minc r-operator (see § 3.3). All processors v verify $\texttt{IN}_v = +\infty$ except a non null processor u verifying $\texttt{IN}_u = 0$ (0 is absorbing while $+\infty$ is the identity element for minc). Each r-path-mapping adds its length to its argument (*i.e.*, $r_P(x) = x + \text{length}(P)$), and we have:

$$d(u,v) = \min\left(\texttt{IN}_v, \min_{w \in \Gamma^-_v, P_{w,v} \in \mathcal{X}_{w,v}} \left\{r_{P_{w,v}}(\texttt{IN}_w)\right\}\right)$$

We now define the legitimate output of a processor v in the general case.

Definition 4 (Legitimate output) *The* legitimate output *of processor v is:*

$$\widehat{\texttt{OUT}}_v = \texttt{IN}_v \oplus \bigoplus_{u \in \Gamma^-_v, P_{u,v} \in \mathcal{X}_{u,v}} r_{P_{u,v}}(\texttt{IN}_u)$$

The following lemma is given by Lemma 3, Hypothesis 5 and Definition 4; it is used for proving Theorem 1.

Lemma 4 *The set $\mathbb{S}$ is either finite or any processor $v \in \mathcal{P}$ verifies: $\widehat{\texttt{OUT}}_v \prec_\oplus e_\oplus$.*

Now we defined $\widehat{\texttt{OUT}}_v$, we define the set of legitimate configurations $L \subset \mathcal{C}$ of the protocol $\mathcal{PA}$-MP (see Section 3 and Figure 2):

Definition 5 (Legitimate configuration) *For any configuration $c \in L$, for any processor $v \in \mathcal{P}$, $\texttt{OUT}_v(c) = \widehat{\texttt{OUT}}_v$.*

Finally, after defining the distributed system $\mathcal{S}$, the generic algorithm $\mathcal{PA}$-MP, the r-operators used as parameters and some Hypotheses, we can express the main result of this paper as follows, which is proved in the following section:

Theorem 1 *Algorithm $\mathcal{PA}$-MP parameterized by any strictly idempotent r-operator is self-stabilizing in directed message passing networks, despite fair loss, finite duplication and reordering of messages.*

The message passing model that we consider leads to hard difficulties (compared for instance to shared memory model [12]). Indeed, with this model it is possible that an initially wrong message remains in a link for quite a long (finite) time (*e.g.* after several new messages have been exchanged) and then is delivered to cause havoc in the system. Note that to reuse [12] in unreliable message passing systems, a self-stabilizing data link protocol is required, yet no such data link protocol exists in unidirectional networks. So, our approach is the first to date to support multiple metrics in (realistic) unreliable unidirectional networks. We hereby give the main proof arguments. Details are provided in Section 4.

Scketch of proof: Despite weak hypotheses on the communication capabilities of every link (u,v), and possible transient failures that could corrupt data in links or nodes communication buffers $\texttt{OUT}_u$ and $\texttt{IN}^u_v$, we have to prove that eventually any input value read by v in $\texttt{IN}^u_v$ has

effectively been sent by u. Even though this is true, it does not imply that a value sent by u will be received by v. Hence, a legitimate value sent by u could be lost in (u, v), while the inputs of u that were used to produce it disappeared, either because of transient failures, or simply because they were overwritten by other incoming values. This means that legitimate values could completely be removed from $\mathcal{S}$.

We actually have to prove that a value received by v on (u, v) has been sent by u *after* a given configuration. This configuration is chosen such that the value of u fulfills some predicates. One of those predicates is that this value has been built using incoming values of u sent by its predecessors *after* a given configuration. This permits to use recursivity along paths of the network.

By weak fairness, any processor v calls `Evaluate` for updating its output $\texttt{OUT}_v$ using its inputs. By properties of the r-operators, and using the total order Hypothesis (Hyp. 3), this output is either built with $\texttt{IN}_v$ or with a received value, say $\texttt{IN}_v^u$. After the last transient failure, and since duplications are finite on the link (u, v), any value received by v has been sent by u. Since every perturbation on the link is finite, there is a finite number of configurations between the sending of the value by u and its receipt by v. Thus, if we consider a configuration that is far enough in the execution, v must have updated its output using a value received by u after u has itself updated its output too. This way, we can prove that any output is smaller or equal than the legitimate value, which means that every large unlegitimate value eventually disappears from the network.

To complete the proof of correctness, we still have to prove that every processor v may not remain with a smaller value than its legitimate one. Suppose this is the case, then by reusing a recursive reasoning, we obtain an infinite path of processors, such that their outputs are strictly increasing along the path (by the strict idempotency property of the r-operators). Since such a path does not exists in the network (that is finite), it is a cycle. This means that, successive outputs of v increase without ever reaching its legitimate value. That contradicts Hypothesis 4. □

3.6 Example

Some r-operators have been proposed to compute the minimum distance tree and forest, the shortest path tree and forest, the best reliable paths from some transmitters, the depth first search tree... More complex applications [14] have also been proposed by combining several r-operators.

For instance, when the local algorithms are parameterized by the minc r-operator, the system stabilizes to a minimum distance tree when all the node are initialized with $e_{\text{minc}} = +\infty$ except one (the root) initialized with 0 (see Figure 3).

4 Proof of Correctness

This section is divided into six parts. First, we give basic results related to the operators. Second, we prove that eventually the output of each processors is updated using its inputs. Third, we show that eventually each received message was sent in the past. Fourth, we prove that each processors output is upper bounded. Fifth, we prove that each processor eventually reaches its legitimate value. Finally, we present complexity results regarding our distributed protocol.

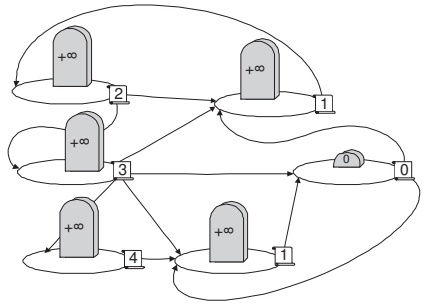


Figure 3: The minc r-operator on each node leads to a minimum distance tree computation in a unreliable unidirectional network.

4.1 Properties of the operators

Any r-operator $\triangleleft$ defined on $\mathbb{S}$ induces an s-operator $\oplus$ on $\mathbb{S}$ by $x \oplus r(y) = x \triangleleft y$. Since the s-operator defines an order relation $\preceq_\oplus$ by $x \preceq_\oplus y \equiv x \oplus y = x$, the Lemma 5 holds. Since r is an homomorphism of $(\mathbb{S}, \oplus)$, the Lemma 6 holds.

Lemma 5 *For all $x, y, z \in \mathbb{S}$, if $x \oplus y = z$ then $z \preceq_\oplus x$ and $z \preceq_\oplus y$.*

Lemma 6 *For all $x, y \in \mathbb{S}$, if $x \preceq_\oplus y$, then $r(x) \preceq_\oplus r(y)$.*

4.2 Outputs eventually result from computations

We begin by defining some predicates on configurations.

Definition 6 *Let P_{0a}, P_{0b} and P_{0c} be predicates on configurations $c \in \mathcal{C}$:*

$$\begin{aligned}
P_{0a}(c) &\equiv \forall v \in \mathcal{P}, \quad \texttt{OUT}_v(c) \preceq_\oplus \texttt{IN}_v \\
P_{0b}(c) &\equiv \forall v \in \mathcal{P}, \forall u \in \Gamma_v^{-1}, \quad \texttt{OUT}_v(c) \preceq_\oplus r_v^u(\texttt{IN}_v^u(c)) \\
P_{0c}(c) &\equiv \forall v \in \mathcal{P}, \forall u \in \Gamma_v^{-1}, \quad \texttt{OUT}_v(c) = \texttt{IN}_v \\
&\quad \vee \texttt{OUT}_v(c) = r_v^u(\texttt{IN}_v^u(c))
\end{aligned}$$

Now, the set $Q_0 \subset \mathcal{E}$ includes executions where processors eventually update their output. Every execution e of Q_0 reaches a configuration c_{i_0} such that any subsequent configuration c_j satisfies Predicates P_{0a}, P_{0b} and P_{0c}.

Definition 7 *Let $Q_0 \subset \mathcal{E}$ be the set of executions such that:*

$$\begin{aligned}
\forall e \in Q_0, \quad \exists c_{i_0} \in e_{|\mathcal{C}}, \forall c_j \in e_{|\mathcal{C}} \text{ with } i_0 \leq j, \\
P_{0a}(c_j) \wedge P_{0b}(c_j) \wedge P_{0c}(c_j)
\end{aligned}$$

We now prove that, thanks to weak fairness hypothesis, any execution of $\mathcal{E}$ is in Q_0.

Lemma 7 *Every execution of the $\mathcal{PA}$-MP algorithm in the distributed system $\mathcal{S}$ is in Q_0.*

Proof: Let $e \in \mathcal{E}$ be an execution. By weak fairness, every processor $v \in \mathcal{P}$ eventually executes a rule. By definition of $\mathcal{PA}$-MP (see Figure 2), any execution of either rule at some node v processes $\texttt{Evaluate}_v(\lhd_v)$. Then, for any processor $v \in \mathcal{P}$, there exists a configuration $c_{i_v} \in e_{|\mathcal{C}}$ where processor v satisfies $\texttt{OUT}_v(c_{i_v}) = \lhd_v(\texttt{IN}_v, \texttt{IN}_v^{u_1}(c_{i_v}), \ldots, \texttt{IN}_v^{u_{\delta_v}}(c_{i_v}))$.

By Lemma 2, we have $\texttt{OUT}_v(c_{i_v}) = \texttt{IN}_v \oplus r_v^{u_1}(\texttt{IN}_v^{u_1}(c_{i_v})) \oplus \cdots \oplus r_v^{u_{\delta_v}}(\texttt{IN}_v^{u_{\delta_v}}(c_{i_v}))$. Then, by Lemma 5, we have $\texttt{OUT}_v(c_{i_v}) \preceq_\oplus \texttt{IN}_v$ and $\texttt{OUT}_v(c_{i_v}) \preceq_\oplus r_v^u(\texttt{IN}_v^u)$ for any direct-predecessor u of v. Hence, both $P_{0a}(c_{i_v})$ and $P_{0b}(c_{i_v})$ hold. Now, since $\preceq_\oplus$ defines a total order relation (Hypothesis 3), either $\texttt{OUT}_v(c_{i_v}) = \texttt{IN}_v$ or $\texttt{OUT}_v(c_{i_v}) = r_v^u(\texttt{IN}_v^u(c_{i_v}))$ for at least one predecessor u of v. This gives $P_{0c}(c_{i_0})$ with $i_0 = \max_{v \in \mathcal{P}} i_v$.

Since any action of v executed upon receipt of a message or upon timeout expiration calls `Evaluate`, any subsequent configuration satisfies Predicates P_{0a} to P_{0c}. □

4.3 Eventually, received messages were previously sent

We define the set Q_1 as the subset of executions $\mathcal{E}$ for which any received value has actually been sent in the past. All executions e of Q_1 reach a configuration c_{i_1} such that, for any subsequent configuration c_j and any communication link (u, v), there exists a configuration $c_{j_{uv}}$ in which v sent the value contained in $\texttt{IN}_v^u$ in configuration c_j.

Definition 8 *Let $Q_1 \subset \mathcal{E}$ be the set of executions that satisfy:*

$$\forall e \in Q_1, \quad \exists c_{i_1} \in e_{|\mathcal{C}}$$
$$\begin{cases} \forall c_j \in e_{|\mathcal{C}} \text{ with } i_1 \leq j, \forall (u,v) \in \mathcal{L}, \\ \exists c_{j_{uv}} \in e_{|\mathcal{C}} \text{ with } j_{uv} \leq j, \quad \texttt{OUT}_u(c_j) = \texttt{IN}_v^u(c_{j_{uv}}) \end{cases}$$

We now prove that, thanks to Hypothesis 1 related to the properties of the communications links, any execution is in Q_1.

Lemma 8 *Every execution of the $\mathcal{PA}$-MP algorithm in the distributed system $\mathcal{S}$ is in Q_1.*

Proof: Let $e \in \mathcal{E}$ be an execution, and consider two processors u and v such that (u, v) is a communication link of $\mathcal{L}$. By definition of $\mathcal{PA}$-MP, processor v sends the value of its $\texttt{OUT}_v$ variable infinitely often to each of its direct successors. By Hypothesis 1, every message that is not lost is eventually delivered. Moreover, every message may be duplicated only a finite number of times. It follows that, after a finite amount of time, only messages that were sent by v are received by all of its direct successors. Hence, there exists a configuration $c_j \in e_{|\mathcal{C}}$ where the incoming value in $\texttt{IN}_v^u$ has actually been sent by u in a previous configuration $c_{j_{uv}}$:

$$\texttt{IN}_v^u(c_j) = \texttt{OUT}_u(c_{j_{uv}}) \text{ with } j_{uv} \leq j \tag{1}$$

After all initial erroneous messages between u and v have been received (including duplicates), and after a configuration where the above property holds, this property remains thereafter on this link. Since all links conform to the same hypotheses, there exists a configuration $c_{i_1} \in e_{|\mathcal{C}}$ where the property holds (and remains so thereafter) for any communication link. We conclude that $e \in Q_1$. □

Note that this lemma does not indicate that any sent value is eventually received. Indeed, it may happen that a message is lost while traversing a link, and the variable it was built with is erased by a new value. Then, any re-sending would not provide the original value, that would not be received again. We now generalize the notation we introduced in the previous proof.

hal-00632413, version 1 - 14 Oct 2011

Definition 9 *Let us consider an incoming value $\texttt{IN}_v^u(c_j)$ on processor v in the configuration c_j. Then we denote by $c_{j_{uv}}$ the configuration in which the value $\texttt{IN}_v^u(c_j)$ has been sent by u, provided that this configuration exists.*

The previous lemma indicates that, for any execution $e \in \mathcal{E}$, there exists a configuration c_{i_1} from which $c_{j_{uv}}$ exists for any subsequent configuration c_j ($i_1 \leq j$), and any communication link (u, v). However, as captured in Figure 4, the definition of Q_1 gives no guarantees about $c_{j_{uv}}$ appearing after configuration c_{i_1} (that is $i_1 \leq j_{uv}$).

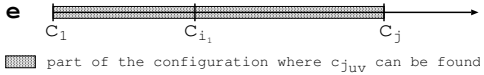


Figure 4: According to Q_1, configuration $c_{j_{uv}}$ exists but could appear before c_{i_1}.

We now introduce additional sets of executions. The following definition, illustrated in Figure 5, indicates that, for any execution in Q_{1b}, from a given configuration $c_{i_{1b}}$, any given configuration c_i admits a configuration $c_{i'}$ such that any configuration $c_{j_{uv}}$ (with $i' \leq j$) appeared after c_i (*i.e.*, $i \leq j_{uv}$).

Definition 10 *Let $Q_{1b} \subset \mathcal{E}$ be the set of executions that satisfy:*

$$\forall e \in Q_{1b}, \quad \exists c_{i_{1b}} \in e_{|\mathcal{C}}$$
$$\begin{cases} \forall c_i \in e_{|\mathcal{C}} \text{ with } i_{1b} \leq i, \exists c_{i'} \in e_{|\mathcal{C}} \text{ with } i \leq i', \\ \forall c_j \in e_{|\mathcal{C}} \text{ with } i' \leq j, \forall (u,v) \in \mathcal{L}, \\ c_{j_{uv}} \in e_{|\mathcal{C}} \wedge \quad i \leq j_{uv} \leq j \end{cases}$$

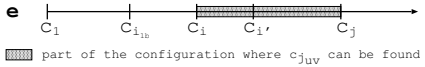


Figure 5: According to Q_{1b}, from a configuration $c_{i_{1b}}$, configurations $c_{j_{uv}}$ can be found later than any given configuration c_i.

We show now that, thanks to weak fairness, every execution is in Q_{1b}.

Lemma 9 *Every execution of the $\mathcal{PA}$-MP algorithm in the distributed system $\mathcal{S}$ is in Q_{1b}.*

Proof: Let $e \in \mathcal{E}$ be an execution that is not in Q_{1b}. From Lemma 8, e is in Q_1 and, from a configuration $c_{i_1} \in e_{|\mathcal{C}}$, for every configuration c_j and every link (u, v), the configuration $c_{j_{uv}}$ exists. Now, let us consider configurations c_i, $c_{i'}$ and c_j in $e_{|\mathcal{C}}$ such that $i_1 \leq i \leq i' \leq j$. If $e \notin Q_{1b}$, then configuration $c_{j_{uv}}$ always appears before c_i, even if $c_{i'}$ (and then c_j) is as far as possible from c_i (see Figure 6). This means that the values produced by processor u after $c_{j_{uv}}$ were never received, that contradicts Lemma 1.

□

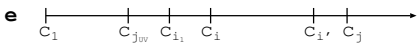


Figure 6: If $e \notin Q_{1b}$, the configuration $c_{j_{uv}}$ always appears before c_i.

4.4 Outputs are eventually smaller than (or equal to) legitimate values

Let us begin by defining two predicates P_2 and P_{2b} on configurations. If $P_2(c)$ holds, then, in configuration c, each processor is smaller than all initial values of its predecessors increased by some r-mappings (more precisely, for any processor v and any of its direct-predecessors u, the output of v is smaller than the initial value of u transformed by the r-path-mapping $r_{P_{u,v}}$ of the path $P_{u,v}$ from u to v). If $P_{2b}(c)$ holds, then, in the configuration c, the output of each processor v is smaller (in the sense of $\oplus$) than its legitimate output.

Definition 11 *Let P_2 and P_{2b} be predicates on configurations $c \in \mathcal{C}$:*

$$\begin{array}{rcl} P_2(c) & \equiv & \forall v \in \mathcal{P}, \forall u \in \Gamma_v^-, \forall P_{u,v} \in \mathcal{X}_{u,v}, \\ & & \mathtt{OUT}_v(c) \preceq_\oplus r_{P_{u,v}}(\mathtt{IN}_u) \\ P_{2b}(c) & \equiv & \forall v \in \mathcal{P}, \quad \mathtt{OUT}_v(c) \preceq_\oplus \widehat{\mathtt{OUT}_v} \end{array}$$

We now define two sets of executions Q_2 and Q_{2b}. If an execution e is in Q_2 (resp Q_{2b}), then there exists a configuration in e from which every configuration satisfies P_2 (resp. P_{2b}).

Definition 12 *Let Q_2 and Q_{2b} be two subsets of $\mathcal{E}$:*

$$\begin{array}{l} \forall e \in Q_2, \quad \exists c_{i_2} \in e_{|\mathcal{C}}, \quad \forall c_j \in e_{|\mathcal{C}} \text{ with } i_2 \leq j, \quad P_2(c_j) \\ \forall e \in Q_{2b}, \quad \exists c_{i_{2b}} \in e_{|\mathcal{C}}, \quad \forall c_j \in e_{|\mathcal{C}} \text{ with } i_{2b} \leq j, \quad P_{2b}(c_j) \end{array}$$

We now prove that, first every execution of $\mathcal{E}$ is in Q_2, and then that every execution of $\mathcal{E}$ is in Q_{2b}. This means that, while the processor's outputs can be larger than the legitimate values in the beginning of an execution, each processor eventually produces some outputs that are smaller than or equal to its legitimate value. In other terms, any erroneous values that are larger than legitimate values eventually disappear from $\mathcal{S}$.

Lemma 10 *Every execution of the $\mathcal{PA}$-MP algorithm in the distributed system $\mathcal{S}$ is in Q_2.*

Proof: Let $e \in \mathcal{E}$ be an execution, and let us consider a processor $v_0 \in \mathcal{P}$, and one of its direct-predecessor $v_1 \in \Gamma_{v_0}^{-1}$. By Lemma 7, e is in Q_0. Then, there exists a configuration $c_{i_0} \in e_{|\mathcal{C}}$ such that, for any subsequent configuration $c_{j_{v_0}} \in e_{|\mathcal{C}}$ ($i_0 \leq j_{v_0}$), Predicate $P_{0b}(c_{j_{v_0}})$ is satisfied: $\mathtt{OUT}_{v_0}(c_{j_{v_0}}) \preceq_\oplus r_{v_0}^{v_1}(\mathtt{IN}_{v_0}^{v_1}(c_{j_{v_0}}))$.

Since $e \in Q_1$, the above configuration $c_{j_{v_0}}$ can be chosen after c_{i_1} in e (*i.e.*, $i_0 \leq j_{v_0}$ and $i_1 \leq j_{v_0}$) so that there exists a configuration $c_{j_{v_1v_0}} \in e_{|\mathcal{C}}$ that appears before $c_{j_{v_0}}$ (*i.e.*, $j_{v_1v_0} \leq j_{v_0}$) satisfying: $\mathtt{OUT}_{v_1}(c_{j_{v_1v_0}}) = \mathtt{IN}_{v_0}^{v_1}(c_{j_{v_0}})$. This gives:

$$\mathtt{OUT}_{v_0}(c_{j_{v_0}}) \preceq_\oplus r_{v_0}^{v_1}\left(\mathtt{OUT}_{v_1}(c_{j_{v_1v_0}})\right) \tag{2}$$

Since $e \in Q_{1b}$, it is possible to choose configuration $c_{j_{v_0}}$ in $e_{|\mathcal{C}}$ in order to ensure that $c_{j_{v_1v_0}}$ appears *after* c_{i_0}. Hence, without loss of generality, we can state $i_0 \leq j_{v_1v_0}$ and thus

$P_{0a}(c_{j_{v_1v_0}})$ holds. This means that $\mathtt{OUT}_{v_1}(c_{j_{v_1v_0}}) \preceq_\oplus \mathtt{IN}_{v_1}$, and, from Lemma 6, we have: $r_{v_0}^{v_1}(\mathtt{OUT}_{v_1}(c_{j_{v_1v_0}})) \preceq_\oplus r_{v_0}^{v_1}(\mathtt{IN}_{v_1})$.

Finally, we obtain the following relation, that remains true for configurations that appear after $c_{j_{v_0}}$:

$$\mathtt{OUT}_{v_0}(c_{j_{v_0}}) \preceq_\oplus r_{v_0}^{v_1}(\mathtt{IN}_{v_1}) \tag{3}$$

and this result remains true hereafter.

To iterate the above reasoning from vertex v_1 (instead of v_0) at configuration $c_{j_{v_1v_0}}$ (instead of $c_{j_{v_0}}$), we must ensure that $c_{j_{v_1v_0}}$ appears after c_{i_0} (to use Q_0) and after c_{i_1} (to use Q_1). Yet using the fact that $e \in Q_{1b}$, the configuration $c_{j_{v_0}}$ can be chosen as far as necessary in e in order to ensure that the related configuration $c_{j_{v_1v_0}}$ happens *after* the configurations c_{i_0} and c_{i_1} (see Figure 5). Hence, for any path $v_k, \ldots, v_0$, there exist some configurations $c_{j_{v_kv_{k-1}}}, \ldots, c_{j_{v_1,v_0}}, c_{j_{v_0}}$ such that the following relations (obtained from Equations 2 and 3) remain true for the rest of the execution:

$$\begin{array}{rccc} & \mathtt{OUT}_{v_0}(c_{j_{v_0}}) & \preceq_\oplus & r_{v_0}^{v_1}(\mathtt{OUT}_{v_1}(c_{j_{v_1v_0}})) \\ \wedge & \mathtt{OUT}_{v_0}(c_{j_{v_0}}) & \preceq_\oplus & r_{v_0}^{v_1}(\mathtt{IN}_{v_1}) \\ & \mathtt{OUT}_{v_1}(c_{j_{v_1v_0}}) & \preceq_\oplus & r_{v_1}^{v_2}(\mathtt{OUT}_{v_1}(c_{j_{v_2v_1}})) \\ \wedge & \mathtt{OUT}_{v_1}(c_{j_{v_1v_0}}) & \preceq_\oplus & r_{v_1}^{v_2}(\mathtt{IN}_{v_2}) \\ & \vdots & \preceq_\oplus & \vdots \\ & \mathtt{OUT}_{v_k}(c_{j_{v_kv_{k-1}}}) & \preceq_\oplus & r_{v_k}^{v_{k-1}}(\mathtt{OUT}_{v_0}(c_{j_{v_1v_0}})) \\ \wedge & \mathtt{OUT}_{v_k}(c_{j_{v_kv_{k-1}}}) & \preceq_\oplus & r_{v_k}^{v_{k-1}}(\mathtt{IN}_{v_{k-1}}) \end{array} \tag{4}$$

Then, for any predecessor v_k of v_0 and any path $P_{v_kv_0} \in \mathcal{X}_{v_k,v_0}$ from v_k to v_0, there exists a configuration $c_{j_{v_0}}$ such that the following remains true in any subsequent configuration: $\mathtt{OUT}_{v_0}(c_{j_{v_0}}) \preceq r_{P_{v_kv_0}}(\mathtt{IN}_{v_k})$. Hence there exists a configuration c_{i_2} reached after all configurations $c_{j_{v_0}}$ (for any processor $v_0 \in \mathcal{P}$) and such that, for any further configuration c_j (*i.e.*, $i_{2b} \leq j$), we have $P_2(c_j)$. This gives the lemma. □

Lemma 11 *Every execution of the $\mathcal{PA}$-MP algorithm in the distributed system $\mathcal{S}$ is in Q_{2b}.*

Proof: Let us consider an execution $e \in \mathcal{E}$. Since $e \in Q_2$, there exists a configuration $c_{i_2} \in e_{|\mathcal{C}}$ such that, for any subsequent configuration $c_j \in e_{|\mathcal{C}}$ (*i.e.*, $i_2 \leq j$), $P_2(c_j)$ holds:

$$\forall v \in \mathcal{P}, \forall u \in \Gamma_v^-, \forall P_{u,v} \in \mathcal{X}_{u,v}, \quad \mathtt{OUT}_v(c_j) \preceq_\oplus r_{P_{u,v}}(\mathtt{IN}_u)$$

Then, we have:

$$\forall v \in \mathcal{P}, \quad \mathtt{OUT}_v(c_j) \preceq_\oplus \bigoplus_{u \in \Gamma_v^-, P_{u,v} \in \mathcal{X}_{u,v}} r_{P_{u,v}}(\mathtt{IN}_u)$$

Since $e \in Q_1$, some of these configurations c_j also satisfy predicate P_{0a}. Without loss of generality, we assume that $P_{0a}(c_j)$ holds: $\mathtt{OUT}_v(c_j) \preceq_\oplus \mathtt{IN}_v$. Hence, we have:

$$\forall v \in \mathcal{P}, \quad \mathtt{OUT}_v(c) \preceq_\oplus \mathtt{IN}_v \oplus \bigoplus_{u \in \Gamma_v^-, P_{u,v} \in \mathcal{X}_{u,v}} r_{P_{u,v}}(\mathtt{IN}_u)$$

This ends the proof, by Definition 4. □

4.5 Legitimate values are eventually reached

Let us begin by defining a predicate on system configurations.

Definition 13 *Let P_3 be a predicate on configurations $c \in \mathcal{C}$:*

$$P_3(c) \quad \equiv \quad \forall v \in \mathcal{P}, \quad \mathtt{OUT}_v(c) = \widehat{\mathtt{OUT}}_v$$

We now define the set of executions Q_3, that corresponds to executions of $\mathcal{E}$ for which every processor eventually reach its legitimate value: all executions of Q_3 reach a configuration c_{i_3} such that, for any subsequent configuration c_j, the outputs of every processor v in c_j are equal to their legitimate values.

Definition 14 *Let $Q_3 \subset \mathcal{E}$ be the set of executions that satisfy:*

$$\forall e \in Q_3, \quad \exists c_{i_3} \in e_{|\mathcal{C}}, \forall c_j \in e_{|\mathcal{C}}, \text{ with } i_3 \leq j, \quad P_3(c)$$

We now prove that any execution is in Q_3.

Lemma 12 *Every execution of the $\mathcal{PA}$-MP algorithm in the distributed system $\mathcal{S}$ is in Q_3.*

Proof: Let $e \in \mathcal{E}$ be an execution, and suppose that $e \notin Q_3$. Since $\preceq_\oplus$ defines a total order (Hypothesis 3), we have:

$$\begin{array}{l} \forall c_{i_3} \in e_{|\mathcal{C}}, \quad \exists c_j \in e_{|\mathcal{C}}, \text{ with } i_3 \leq j, \\ \exists v \in \mathcal{P}, \quad \widehat{\mathtt{OUT}}_v \prec_\oplus \mathtt{OUT}_v(c_j) \quad \vee \quad \mathtt{OUT}_v(c_j) \prec_\oplus \widehat{\mathtt{OUT}}_v \end{array} \tag{5}$$

By Lemma 11, e is in Q_{2b} and there exist some configurations c_j that satisfy both $i_3 \leq j$ and $i_{2b} \leq j$, so that $\mathtt{OUT}_v(c_j) \preceq_\oplus \widehat{\mathtt{OUT}}_v$. Hence, Equation 5 becomes:

$$\begin{array}{l} \forall c_{i_3} \in e_{|\mathcal{C}}, \quad \exists c_j \in e_{|\mathcal{C}}, \text{ with } i_3 \leq j, \\ \exists v \in \mathcal{P}, \quad \mathtt{OUT}_v(c_j) \prec_\oplus \widehat{\mathtt{OUT}}_v \end{array} \tag{6}$$

By Definition 4 and Lemma 5, we have $\widehat{\mathtt{OUT}}_v \preceq_\oplus \mathtt{IN}_v$. This gives $\mathtt{OUT}_v(c_j) \prec_\oplus \mathtt{IN}_v$. Since $e \in Q_0$, there exist some configurations $c_j \in e_{|\mathcal{C}}$ satisfying both Equation 6 and $P_{0c}(c_j)$, that is $i_0 \leq j$. Without loss of generality, we suppose that $P_{0c}(c_j)$ holds: $\exists u \in \Gamma_v^{-1}, \mathtt{OUT}_v(c_j) = r_v^u(\mathtt{IN}_v^u(c_j))$.

As $\mathtt{OUT}_v(c_j) \prec_\oplus \widehat{\mathtt{OUT}}_v$, we have $r_v^u(\mathtt{IN}_v^u(c_j)) \neq e_\oplus$. Since $r_v^u(e_\oplus) = e_\oplus$ (see § 3.3), we have $\mathtt{IN}_v^u(c_j) \neq e_\oplus$. Then, by Definition 2, we have $\mathtt{IN}_v^u(c_j) \prec_\oplus r_v^u(\mathtt{IN}_v^u(c_j))$ and finally $\mathtt{IN}_v^u(c_j) \prec_\oplus \mathtt{OUT}_v(c_j)$. Hence, the following holds: $\exists u \in \Gamma_v^{-1}, \mathtt{IN}_v^u(c_j) \prec_\oplus \mathtt{OUT}_v(c_j)$.

By Lemma 8, $e \in Q_1$, and there exists some configuration c_j that satisfy $i_1 \leq j$ (as well as $i_4 \leq j$, $i_{2b} \leq j$ and $i_0 \leq j$) and for which configuration $c_{j_{uv}}$ exists in e and verifies $\mathtt{OUT}_u(c_{j_{uv}}) = \mathtt{IN}_v^u(c_j)$. Then $\mathtt{OUT}_u(c_{j_{uv}}) \prec_\oplus \mathtt{OUT}_v(c_j) \prec_\oplus \widehat{\mathtt{OUT}}_v$. This means that at least one of the direct-predecessors u of v verifies $\mathtt{OUT}_u(c_{j_{uv}}) \prec_\oplus \widehat{\mathtt{OUT}}_u \vee \widehat{\mathtt{OUT}}_u \prec_\oplus \mathtt{OUT}_u(c_{j_{uv}})$ (indeed, if all predecessors of v reached and hold their legitimate value, then v would reach its legitimate value too). Hence, Equation 6 becomes:

$$\begin{array}{l} \forall c_{i_3} \in e_{|\mathcal{C}}, \quad \exists c_j \in e_{|\mathcal{C}}, \text{ with } i_3 \leq j, \\ \exists u, v \in \mathcal{P} \text{ with } u \in \Gamma_v^{-1}, \quad \exists c_{j_{uv}} \in e_{|\mathcal{C}}, \text{ with } j_{u,v} \leq j \\ \big(\mathtt{OUT}_u(c_{j_{uv}}) \prec_\oplus \widehat{\mathtt{OUT}}_u \quad \vee \quad \widehat{\mathtt{OUT}}_u \prec_\oplus \mathtt{OUT}_u(c_{j_{uv}})\big) \\ \wedge \quad \mathtt{OUT}_u(c_{j_{uv}}) \prec_\oplus \mathtt{OUT}_v(c_j) \prec_\oplus \widehat{\mathtt{OUT}}_v \end{array} \tag{7}$$

hal-00632413, version 1 - 14 Oct 2011

To iterate the above argument from processor u instead of v, and from configuration $c_{j_{uv}}$ instead of c_j, we argue that $i_0 \leq j_{uv}$, $i_1 \leq j_{uv}$, and $i_{2b} \leq j_{uv}$. By Lemma 9, e is in Q_{1b}. This means that configurations c_{i_3} in the above equation can be chosen so that every configurations $c_{j_{uv}}$ appear *after* configurations c_{i_0}, c_{i_1} and $c_{i_{2b}}$ (see Figure 5). This allows to re-use the above reasoning with configuration $c_{j_{uv}}$ instead of c_j.

By iterating the above arguments, and since the network is finite, we exhibit a cycle of nodes and a set of configurations $c_{j_0}, c_{j_1} \ldots$ appearing after c_{i_4} in e such that, for a node w in the cycle, we have:

$$\mathtt{OUT}_w(c_{j_0}) \prec_\oplus \mathtt{OUT}_w(c_{j_1}) \prec_\oplus \ldots \prec_\oplus \widehat{\mathtt{OUT}}_w \tag{8}$$

Using the fact that $e \in Q_{1b}$, this can be found after any configuration c_{i_4} in the execution e. This means that, regardless of configuration c_{i_4}, there exist subsequent configurations $c_{j_0}, \ldots c_{j_1}$, such that $\widehat{\mathtt{OUT}}_w$ increases strictly without reaching its legitimate value. We then exhibit a strictly increasing sequence of values of $\mathbb{S}$ that never reach $\widehat{\mathtt{OUT}}_w$. This is impossible if $\mathbb{S}$ is finite. If $\mathbb{S}$ is infinite, then Lemma 4 gives $\widehat{\mathtt{OUT}}_w \prec_\oplus e_\oplus$. The sequence of values is then upper bounded, that contradicts Hypothesis 4. Hence, $e \in Q_3$. □

4.6 Complexity

In the convergence part of the proof, we only assumed that computations were maximal, and that message loss, duplication and desequencing could occur. In order to provide an upper bound on the stabilization time for our algorithm, we assume strong synchrony between processors and a reliable communication medium between nodes. Note that these assumptions are used for complexity results only, since our algorithm was proved correct even in the case of asynchronous unfair computations with link intermittent failures. In the following, D denotes the network diameter.

In order to give an upper bound on the space and time requirements, we assume that the set $\mathbb{S}$ is finite, and that $|\mathbb{S}|$ denotes its number of elements. This assumption is used for complexity results only, since our algorithm was proved to be correct even in the case when $\mathbb{S}$ is infinite. Note that in any implementation the set of possible values is finite, and if the memories $\mathtt{IN}_v$ and $\mathtt{OUT}_v$ of each node v contains n bits, then $|\mathbb{S}| = 2^n$.

The space complexity result is immediately given by the assumptions made when writing Algorithm $\mathcal{PA}$-MP.

Lemma 13 (Space Complexity) *Each processor $v \in \mathbb{S}$ holds $(\delta v + 1) \times \log_2(|\mathbb{S}|)$ bits.*

Proof: Each processor v has δv local variables that hold the value of the last message sent by the corresponding direct predecessor, and one register used to communicate with its direct descendants. Each of these local variables may hold a value in a finite set $\mathbb{S}$, then need $\log_2(|\mathbb{S}|)$ bits. Note that the constant stored in ROM is not taken into account in this result. □

Lemma 14 (Time Complexity) *Assuming a synchronous system $\mathcal{S}$, the stabilization time is $O(D + |\mathbb{S}|)$.*

Proof: We define ϕ as the function that returns the index of a given element of $\mathbb{S}$. This index always exists since $\mathbb{S}$ is ordered by a total order relation. The signature of ϕ is as follows:

$$\begin{array}{rccc} \phi: & \mathbb{S} & \to & \mathbb{N} \\ & s & \mapsto & \phi(s) \end{array}$$

Also, we have

$$s_1 \prec_{\oplus} s_2 \Rightarrow \phi(s_1) < \phi(s_2)$$

After $O(D)$ steps, every node in the network has received values from all of their predecessors. If those values were badly initialized, then the received values are also possibly badly valued.

For each node u, we consider the difference between the index of its final value (since the algorithm converges to a legitimate configuration where $\mathtt{OUT}_u = \widehat{\mathtt{OUT}}_u$) and the index of the smallest received value which is badly initialized. The biggest possible difference is $M - m$, where M is the maximum index value of $\mathbb{S}$ and m the minimum index value of $\mathbb{S}$. This difference is called d and is $O(|\mathbb{S}|)$.

For each node u, we also consider the smallest and the greatest (in the sense of increasing) r-path mapping from u to u. Let l be the length of the smallest such r-path mapping. It increases a value index by at least l. The greatest such r-path mapping increases a value index by at most d, and is of length at most d.

In the worst case, there exists a node that has an incorrect input value indexed with m, a correct input value indexed with M, so it has to wait until the incorrect value index is increased by $M-m$ before the incorrect value effect is canceled. Each l time units at least, this incorrect value index is increased by l. Again, in the worst case, if $\lfloor \frac{d}{l} \rfloor < \frac{d}{l}$, another incorrect value may still be lower than the correct value, and the greatest cycle may be followed, inducing an extra d time delay. Overall, after the first $O(D)$ time units, $\left(\lfloor \frac{d}{l} \rfloor \times l\right) + d = O(d)$ time units are needed. □

5 Application to *ad hoc* and Sensor Wireless Networks

In this section, we describe how the loose requirements of our scheme make it suitable for wireless networks such as *ad hoc* and sensor networks.

In such networks, communications are typically not bidirectionnal due to the various possible ranges of antennas, and the fact that nodes could be deployed in various geographical settings. Moreover, due to the possible collisions that can occur when neighboring nodes try to communicate at the same time, it is quite possible that messages are lost or duplicated. Also, since for example sensor networks are composed of nodes with low processing power, some desequencing is expected for message delivery when nodes are overloaded.

While some previous works on self-stabilizing sensor networks expect nodes to be aware of their location [17] or the identity of the nodes in their vicinity, the bootstrapping process that is needed to collect this information can be costly. Also, the hypothesis that nodes have unique IDs (that is mandatory to properly construct the set of identifiers in one vicinity, *e.g.* in [15]) could be falsified if such a property can not be guaranteed in practise (and larger scale construction of sensor networks would probably lead to such possibilities). Previous approaches mentioned in the introduction [1, 4, 5, 10, 13, 12] rely heavily on some kind of local knowledge about the topology: number of distinct input links, number of distinct output links, diameter of the network (for some).

In contrast, the correctness of the scheme presented in this paper does not rely necessarily on *e.g.* disctinct *neighbors*, but rather on the number of distinct *input values*. As such, our algorithm for unreliable message passing networks can be derived into a scheme for wireless networks where nodes use a local broadcast primitive to communicate with neighbors. However, only strictly idempotent r-operators that share a unique r-function are solely input value based. One such qualifying r-operator is the minc operator.

hal-00632413, version 1 - 14 Oct 2011

Our algorithm can be modified as follows for input value based r-operators:

1. The `IN` fixed table is replaced by an associative memory of tuples (v, a) where $v \in \mathbb{S}$ and a is time stamp. In this associative memory, v is supposed to have been received by some (possibly anonymous) neighbor node at time a. Each time a value is received through a delivered message, the entry in the associative memory is either inserted (if the value is new) or updated with a new timestamp. To prevent from bad initialization, each time the associative memory is updated, old entries are removed (following *e.g.* the technique provided in [15]).
2. Instead of computing using the `IN` table, the r-operator operates on the associative memory values.
3. Instead of sending a message to each outgoing link, nodes simply perform a local broadcast of their value.

Of course, this scheme assumes that nodes are endowed with a local real time clock (with no assumptions made about clock synchronization or possible drift), and that the timeouts are properly set so that an actual value at some node is regularly sent to the outgoing neighbors (by a local broadcast), so that those nodes in turn do not remove this value from their associative memory. Most schemes envisioned today for wireless communication between neighboring nodes are probabilistic and guarantee that between any two successful sendings, a constant amount of time is expected, provided that the density in each vicinity is upper bounded by a constant, so our hypothesis remains reasonable. Actualy tuning the timeout so that incorrect entries are quickly removed yet correct entries remain in the system is beyond the scope of this paper.

6 Concluding remarks

We presented a generic distributed algorithm for message passing networks applicable to any directed graph topology. This algorithm tolerates transient faults that corrupt the processors and communication links memory as well as intermittent faults (fair loss, reorder, finite duplication of messages) on communication media. Our contribution allows to envisage new applications for wireless networks (such as sensor networks), where nodes are not aware of their neighbors, and communications could be unidirectional (*e.g.*, non uniform power) and unreliable.

We provided evidence that our scheme is also suitable (for a restricted set of operators) to wireless networks, sur as *ad hoc* and sensor networks. Because our approach is essentially value based, computations can be carried out in potentially anonymous networks without the need of a bootstrapping process.

As an illustration, we quickly presented a simple application of the minc r-operator for solving the shortest path tree problem. Thanks to our generic approach, many others applications can be solved in the same way, by simply changing the operator. Moreover only local conditions have to be checked to insure the self-stabilization of our algorithm. Some r-operators have already been proposed for solving both fundamental and high level applications (see [12, 13]) such as: shortest paths spanning tree and related problems, best reliable paths from some transmitters, depth first search tree... More complex applications can be solved with specific r-operators, though the completeness of r-operators is an open problem.

Acknowledgements This work was supported in part by the FRAGILE and SR2I projects of the ACI "Sécurité et Informatique".

References

[1] Y. Afek and A. Bremler. Self-stabilizing unidirectional network algorithms by power supply. *Chicago Journal of Theoretical Computer Science*, 4(3):1–48, 1998.

[2] Y. Afek and G.M. Brown. Self-stabilization over unreliable communication media. *Distributed Computing*, 7:27–34, 1993.

[3] F. Baccelli, G. Cohen, G. Olsder, and J.-P. Quadrat. *Synchronization and Linearity, an algebra for discrete event systems*. Wiley, Chichester, UK, 1992.

[4] J.A. Cobb and M.G. Gouda. Stabilization of routing in directed networks. In *Proceedings of the Fifth Internationa Workshop on Self-stabilizing Systems (WSS'01), Lisbon, Portugal*, pages 51–66, 2001.

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

[6] Sylvie Delaët, Bertrand Ducourthial, and Sébastien Tixeuil. Self-stabilization with r-operators revisited. In Ted Herman and Sébastien Tixeuil, editors, *Self-Stabilizing Systems*, volume 3764 of *Lecture Notes in Computer Science*, pages 68–80. Springer, 2005.

[7] Sylvie Delaët, Bertrand Ducourthial, and Sébastien Tixeuil. Self-stabilization with r-operators revisited. *Journal of Aerospace Computing, Information, and Communication (JACIC)*, 3(10):498–514, 2006.

[8] S. Dolev. *Self-stabilization*. The MIT Press, 2000.

[9] S. Dolev, M.G. Gouda, and M. Schneider. Memory requirements for silent stabilization. *Acta Informatica*, 36(6):447–462, 1999.

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

[11] B. Ducourthial. New operators for computing with associative nets. In *Proceedings of SIROCCO'98, Amalfi, Italia*, 1998.

[12] B. Ducourthial and S. Tixeuil. Self-stabilization with r-operators. *Distributed Computing*, 14(3):147–162, 2001.

[13] B. Ducourthial and S. Tixeuil. Self-stabilization with path algebra. *Theoretical Computer Science*, 293(1):219–236, 2003.

[14] Bertrand Ducourthial and Sébastien Tixeuil. Adaptive multi-sourced multicast. In *Rencontres Francophones sur les aspects Algorithmiques des Télécommunications (AlgoTel'2001)*, pages 135–142, St-Jean de Luz, France, May 2001. in French.

hal-00632413, version 1 - 14 Oct 2011

[15] Ted Herman and Sbastien 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.

[16] S. Katz and K.J. Perry. Message passing extensions for self-stabilizing systems. *Distributed Computing*, 7(1):17–26, 1993.

[17] 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, San Francisco, CA, USA, June 24-25, 2003, Proceedings*, volume 2704 of *Lecture Notes in Computer Science*, pages 17–31. Springer, 2003.

[18] N.A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.

A.5 Répondre immédiatement

Snap-Stabilization in Message-Passing Systems. Sylvie Delaët, Stéphane Devismes, Mikhail Nesterenko, and Sébastien Tixeuil. *Journal of Parallel and Distributed Computing (JPDC)*, 70(12) :1220-1230, December 2010.

– *Résumé :* Dans cet article, nous étudions le problème de la stabilisation instantanée dans les systèmes à passage de messages. La stabilisation instantanée permet de concevoir des protocoles qui tolèrent des défaillances transitoires : toute exécution qui débute après que des fautes soient survenues satisfait immédiatement la spécification attendue. Notre contribution est double : nous montrons que dans les systèmes à passage de messages, la stabilisation instantanée est impossible pour des problèmes non triviaux si la capacité des canaux de communication est inconnue, et nous montrons que la stabilisation instantanée devient possible si cette capacité est connue. Cette dernière contribution est constructive, et nous proposons deux protocoles instantanément stabilisants pour la propagation d'information avec retour et pour l'exclusion mutuelle. Ce travail ouvre de nouvelles perspectives de recherche, car il rend possible l'implantation de protocoles instantanément stabilisants dans des réseaux réels.

J. Parallel Distrib. Comput. 70 (2010) 1220–1230

Contents lists available at ScienceDirect

J. Parallel Distrib. Comput.

journal homepage: www.elsevier.com/locate/jpdc

Snap-stabilization in message-passing systems☆

Sylvie Delaët [a], Stéphane Devismes [b], Mikhail Nesterenko [c], Sébastien Tixeuil [d,*]

[a] LRI UMR 8623, Université de Paris-Sud, France
[b] VERIMAG UMR 5104, Université Joseph Fourier, France
[c] Computer Science Department, Kent State University, United States
[d] LIP6 UMR 7606, Université Pierre et Marie Curie, France

ARTICLE INFO

Article history:
Received 17 August 2009
Received in revised form
14 April 2010
Accepted 24 April 2010
Available online 25 May 2010

Keywords:
Distributed systems
Self-stabilization
Snap-stabilization

ABSTRACT

In this paper, we tackle the problem of *snap-stabilization* in message-passing systems. Snap-stabilization allows designing protocols that withstand transient faults: indeed, any computation that is started after faults cease *immediately* satisfies the expected specification.

Our contribution is twofold: we demonstrate that in message-passing systems (i) snap-stabilization is impossible for nontrivial problems if channels are of finite yet unbounded capacity, and (ii) snap-stabilization becomes possible in the same setting with bounded capacity channels. The latter contribution is constructive, as we propose two snap-stabilizing protocols for propagation of information with feedback and mutual exclusion.

Our work opens exciting new research perspectives, as it enables the snap-stabilizing paradigm to be implemented in actual networks.

1. Introduction

Self-stabilization [18,19,28] is an elegant approach to forward failure recovery: regardless of the global state to which the failure drives the system, after the failures stop, a self-stabilizing system is guaranteed to resume correct operation within *finite time*. This guarantee comes at the expense of temporary safety violation in the sense that a self-stabilizing system may behave incorrectly as it recovers, without a user of the system being notified of this misbehavior.

Bui et al. [9] introduce the related concept of *snap-stabilization* which guarantees that a protocol *immediately* operates correctly, regardless of the arbitrary initial state of the system. Snap-stabilization offers stronger fault-tolerance properties than self-stabilization: regardless of the global state to which the failure drives the system, after the failures stop, a snap-stabilizing system immediately resumes correct behavior.

The notion of safety that is guaranteed by snap-stabilization is orthogonal to the notion of safety that is guaranteed by super-stabilization [20] or safe stabilization [22]. In [20,22], some safety predicates on configurations and executions are preserved at all times while the system is running. Of course, not all safety predicates can be guaranteed when the system is started from an arbitrary global state. In contrast, snap-stabilization's notion of safety is *user-centric*: when the user initiates a request, then the received response is correct. However, between the request and the response, the system can behave arbitrarily (except from giving an erroneous response to the user). In the snap-stabilizing model, all user-safety predicates can be guaranteed while recovering from arbitrary states. Then, if the system user is sensitive to safety violation, snap-stabilization becomes an attractive option.

However, nearly every snap-stabilizing protocol presented so far assumes a high level communication model in which any process is able to read the states of every communication neighbor and update its own state in a single atomic step (this model is often referred to as the *shared memory model with composite atomicity* in the literature). Designing protocols with forward recovery properties (such as self-stabilizing and snap-stabilizing ones) using lower level communication models such as asynchronous message-passing is rather challenging. In such models, a process may either send a message to a single neighbor or receive a message from a single neighbor (but not both) together with some local computations; also messages in transit could be lost or duplicated. It is especially important to consider these low level models since Varghese and Jayaram [30] proved that simple process crashes and restarts and unreliable communication channels can drive protocols to arbitrary states.

☆ A preliminary version of this paper will be presented in Delaët et al. (2009) [16].

* Corresponding author.
E-mail addresses: sylvie.delaet@lri.fr (S. Delaët), stephane.devismes@imag.fr (S. Devismes), mikhail@cs.kent.edu (M. Nesterenko), sebastien.tixeuil@lip6.fr (S. Tixeuil).

0743-7315/$ – see front matter © 2010 Published by Elsevier Inc.
doi:10.1016/j.jpdc.2010.04.002

1.1. Related works

Several papers investigate the possibility of self-stabilization in message-passing systems [23,26,2,1,25,29,5,6,17]. The crucial assumption for communication channels is their *boundedness*. That is, whether or not processes are aware of the maximum number of messages that can be in transit in a particular channel. Gouda and Multari [23] show that for a wide class of problems such as the alternating bit protocol (ABP), deterministic self-stabilization is impossible using bounded memory per process when channel capacities are unbounded. They also present a self-stabilizing version of the ABP with unbounded channels that uses unbounded memory per process. Afek and Brown [2] present a self-stabilizing ABP replacing unbounded process memory by an infinite sequence of random numbers. Katz and Perry [26] derive a self-stabilizing ABP to construct a self-stabilizing snapshot protocol. In turn, the snapshot protocol allows to transform almost all non-stabilizing protocols into self-stabilizing ones. Delaët et al. [17] propose a method to design self-stabilizing protocols with bounded memory per process in message-passing systems with unreliable channels with unbounded capacity for a class of fix-point problems. Awerbuch et al. [7] introduce the property of *local correctability* and demonstrate that protocols that are locally correctable can be self-stabilized using bounded memory per process in spite of unbounded capacity channels. Guaranteeing self-stabilization with bounded memory per process for general (*i.e.* ABP-like) specifications requires considering bounded capacity channels [1,25,29,5,6]. In particular, Varghese [29] presented such self-stabilizing solutions for a wide class of problems, *e.g.* token circulation and propagation of information with feedback (PIF).

A number of snap-stabilizing protocols are presented in the literature. In particular, PIF is the "benchmark" application for snap-stabilization [10,13]. Moreover [14,12] present token circulation protocols. Snap-stabilization has also been investigated for fix-point tasks such as binary search tree construction [8] and cut-set detection [15]. Following the scheme of [26], Cournier et al. [11] propose a method to add snap-stabilization to a large class of protocols. To our knowledge, the only paper that deals with snap-stabilization in message-passing networks is [21]. However the snap-stabilizing snapshot protocol that is presented in [21] for multi-hops networks relies on the assumption that there exists an underlying snap-stabilizing protocol for *one-hop* message transmission, we do not make such an assumption here. To date, the question whether this assumption can be implemented remains open.

1.2. Our contribution

In this paper, we address the problem of *snap-stabilization* in one-hop message-passing systems. Our contribution is twofold:

(1) We show that contrary to the high level shared memory model, snap-stabilization is strictly more difficult to guarantee than self-stabilization in the low level message-passing model. In more detail, for nontrivial distributed problem specifications, there exists no snap-stabilizing (even with unbounded memory per process) solution in message-passing systems with unbounded yet finite capacity channels. This is in contrast to the self-stabilizing setting, where solutions with unbounded memory per process [23], unbounded sequences of random numbers [2], or operating on a restricted set of specifications [17,7] do exist.
(2) We prove that snap-stabilization in the low level message-passing model is feasible when channels have bounded capacity. Our proof is constructive both for dynamic and fix-point distributed specifications, as we present snap-stabilizing protocols for PIF and mutual exclusion.

1.3. Outline

The rest of the paper is organized as follows. We define the message-passing model in Section 2. In the same section, we describe the notion of snap-stabilization and problem specifications. In Section 3, we exhibit a wide class of problems that have no snap-stabilizing solution in message-passing systems with unbounded capacity channels. We present snap-stabilizing algorithms for the message-passing systems with bounded capacity channels in Section 4. We conclude the paper in Section 5.

2. Preliminaries

2.1. Computational model

We consider distributed systems having *n processes* and a *fully-connected topology*: any two distinct processes can communicate by sending messages through a bidirectional link, i.e., two channels in the opposite direction.

A process is an asynchronous sequential deterministic machine that uses a local memory, a local algorithm, and input/output capabilities. The local algorithm modifies the state of the process memory, and sends/receives messages through channels.

We assume that the channels incident to a process are locally distinguished by a *channel number*. For the sake of simplicity, every process numbers its channels from 1 to $n-1$, and in the code of any process we simply denote by the *label* q the number of the channel incoming from the process q.[1] We assume that the channels are FIFO (meaning that messages are received in the order they are sent) but not necessarily *reliable* (messages can be lost). However they all satisfy the following fairness property: if a sender process s sends infinitely many messages to a receiver process r, then infinitely many messages are eventually received by r from s. Any message that is never lost is received in finite but unbounded time.

The messages are of the following form: ⟨*message-type*, *message-value*⟩. The *message-value* field is omitted if the message does not carry any value. A message may also contain more than one *message-value*.

A *protocol* is a collection of n local algorithms, one held by each process. A local algorithm consists of a collection of actions. Any action is of the following form:

$$\langle label\rangle :: \langle guard\rangle \rightarrow \langle statement\rangle.$$

A *guard* is a boolean expression over the variables of a process and/or an *input* message. A *statement* is a sequence of assignments of the process variables and/or message *sends*. An action can be executed only if its guard is true. We assume that the actions are *atomically* executed, meaning that the evaluation of the guard and the execution of the corresponding statement, if executed, are done in one atomic step. An action is *enabled* when its guard is true. Any *continuously* enabled action is executed within finite yet unbounded time.

We reduce the *state* of each process to the state of its local memory, and the state of each link to its content. The global state of the system, referred to as *configuration*, is defined as the product of the states of the memories of processes and the contents of the links.

We describe a distributed system as a *transition system* [27] $\mathcal{S} = (\mathcal{C}, \mapsto, \mathcal{I})$ such that: $\mathcal{C}$ is a set of configurations, $\mapsto$ is a binary transition relation on $\mathcal{C}$, and $\mathcal{I} \subseteq \mathcal{C}$ is the set of initial configurations. Here, we only consider systems $\mathcal{S} = (\mathcal{C}, \mapsto, \mathcal{I})$ such that $\mathcal{I} = \mathcal{C}$, meaning that any possible configuration can be initial.

[1] The label q does not denote the identifier of q. When necessary, we will use the notation ID_q to denote the identifier of q.

An *execution* of $\mathcal{S}$ is a *maximal* sequence of configurations $\gamma_0, \ldots, \gamma_{i-1}, \gamma_i, \ldots$ such that: $\gamma_0 \in \mathcal{I}$ and $\forall i > 0$, $\gamma_{i-1} \mapsto \gamma_i \wedge \gamma_{i-1} \neq \gamma_i$. Any transition $\gamma_{i-1} \mapsto \gamma_i$ is called a *step* and materializes the fact that some processes and/or links change their states. A process changes its state by executing an enabled action of its local algorithm. The state of a link is modified each time a message is sent, received, or lost.

2.2. Snap-stabilization

In the following, we call *specification* a predicate defined over sequences of configurations.

Definition 1 (*Snap-Stabilization [9]*)**.** Let $\mathcal{SP}_{\mathcal{T}}$ be a specification. A protocol $\mathcal{P}$ is *snap-stabilizing* for $\mathcal{SP}_{\mathcal{T}}$ *if and only if* starting from any configuration, any execution of $\mathcal{P}$ satisfies $\mathcal{SP}_{\mathcal{T}}$.

It is important to note that snap-stabilization is suited for *user-centric* specifications. Such specifications are based on a sequence of actions (request, start, etc.) rather than a particular subset of configurations (e.g., the *legitimate configurations*) and are composed of two main properties:

(1) **Start.** Upon an *external* (*w.r.t.* the protocol) *request*, a process (called the *initiator*) starts within finite time a distributed *computation* of specific task $\mathcal{T}$[2] by executing a special type of action called *starting action*.
(2) **User-Safety.** Any computation of $\mathcal{T}$ that has been started is correctly performed.

In snap-stabilization, we consider the system after the last fault occurs: hence we study the behavior of the system from an arbitrary configuration, yet considered as the initial one, and we assume that no more faults occur. Starting from such an arbitrary configuration (*i.e.*, after the end of faults), any snap-stabilizing protocol always guarantee the **start** and the **user-safety** properties. Hence, snap-stabilization is attractive for system users: when a user make a request, it has the guarantee that the requested task will be correctly performed regardless of the initial configuration of the system under the assumption that no further error occurs. We do not have such a guarantee with self-stabilizing protocols. Indeed, while a snap-stabilizing protocol ensures that any request is satisfied despite the arbitrary initial configuration, a self-stabilizing protocol often needs to repeat its computations an unbounded number of times before guaranteeing the proper processing of any request.

3. Message-passing systems with unbounded capacity channels

Alpern and Schneider [3] observe that a specification is an intersection of *safety* and *liveness* properties. They define a *safety* property as a set of "bad things" that must never happen. We now introduce the notion of *safety-distributed* specifications and show that no problem having such a specification admits a snap-stabilizing solution in message-passing systems with finite yet unbounded capacity channels. Intuitively, safety-distributed specification has a safety property that depends on the behavior of more than one process. That is, certain process behaviors may satisfy safety if done sequentially, while violate it if done concurrently. For example, in mutual exclusion, any requesting process must eventually execute the critical section but if several requesting processes execute the critical section concurrently, the safety is violated. Roughly speaking, the class of *safety-distributed* specifications includes all distributed problems where processes need to exchange messages in order to preclude any safety violation.

[2] E.g., a broadcast, a circulation of a single token, ...

The following three definitions are used to formalize the *safety-distributed* specifications.

Definition 2 (*Abstract Configuration*)**.** We call an *abstract configuration* any configuration restricted to the state of the processes (i.e., the state of each link has been removed).

Definition 3 (*State-Projection*)**.** Let γ be a configuration and p be a process. The *state-projection* of γ on p, denoted $\phi_p(\gamma)$, is the local state of p in γ. Similarly, the *state-projection* of γ on all processes, denoted $\phi(\gamma)$, is the product of the local states of all processes in γ (*n.b.* $\phi(\gamma)$ is an abstract configuration).

Definition 4 (*Sequence-Projection*)**.** Let $s = \gamma_0, \gamma_1, \ldots$ be a configuration sequence and p be a process. The *sequence-projection* of s on p, denoted $\Phi_p(s)$, is the state sequence $\phi_p(\gamma_0), \phi_p(\gamma_1), \ldots$. Similarly, the *sequence-projection* of s on all processes, denoted $\Phi(s)$, is the abstract configuration sequence $\phi(\gamma_0), \phi(\gamma_1), \ldots$.

Definition 5 (*Safety-Distributed*)**.** A specification $\mathcal{SP}$ is *safety-distributed* if there exists a sequence of abstract configurations BAD, called *bad-factor*, such that:

(1) For each execution e, if there exist three configuration sequences e_0, e_1, and e_2 such that $e = e_0 e_1 e_2$ and $\Phi(e_1) = \mathtt{BAD}$, then e does not satisfy $\mathcal{SP}$.
(2) For each process p, there exists at least one execution e_p satisfying $\mathcal{SP}$ where there exist three configuration sequences e_p^0, e_p^1, and e_p^2 such that $e_p = e_p^0 e_p^1 e_p^2$ and $\Phi_p(e_p^1) = \Phi_p(\mathtt{BAD})$.

Almost all classical problems of distributed computing have *safety-distributed* specifications including all synchronization and resource allocation problems. For example, in mutual exclusion a *bad-factor* is any sequence of abstract configurations where several requesting processes execute the critical section concurrently. For the PIF, the *bad-factor* consists in the sequence of abstract configurations where the initiator decides of the termination of a PIF it started while some other processes are still broadcasting the message.

We now consider a message-passing system with unbounded capacity channels and show the impossibility of *snap-stabilization* for *safety-distributed* specifications in that case. Since most of classical synchronization and resource allocation problems are safety-distributed, this result prohibits the existence of snap-stabilizing protocols in message-passing systems if no further assumption is made.

We prove the theorem by showing that a "bad" execution can be obtained by filling the communication channels with messages entailing an unsafe state, and no process can detect that those messages occur due to errors because of the safety-distributed nature of the specification.

Theorem 1. *There exists no safety-distributed specification that admits a snap-stabilizing solution in message-passing systems with unbounded capacity channels.*

Proof. Let $\mathcal{SP}$ be a *safety-distributed* specification and $\mathtt{BAD} = \alpha_0, \alpha_1, \ldots$ be a *bad-factor* of $\mathcal{SP}$.

Assume, for the purpose of contradiction, that there exists a protocol $\mathcal{P}$ that is snap-stabilizing for $\mathcal{SP}$. By Definition 5, for each process p, there exists an execution e_p of $\mathcal{P}$ that can be split into three execution factors e_p^0, $e_p^1 = \beta_0^p, \beta_1^p, \ldots$, and e_p^2 such that $e_p = e_p^0 e_p^1 e_p^2$ and $\Phi_p(e_p^1) = \Phi_p(\mathtt{BAD})$. Let us denote by $MesSeq_p^q$ the ordered sequence of messages that p receives from any process q in e_p^1. Consider now the configuration γ_0 such that:

(1) $\phi(\gamma_0) = \alpha_0$.

(2) For all pairs of distinct processes p and q, the link $\{p, q\}$ has the following state in γ_0:
   (a) The messages in the channel from q to p are exactly the sequence $MesSeq_p^q$ (keeping the same order).
   (b) The messages in the channel from p to q are exactly the sequence $MesSeq_q^p$ (keeping the same order).

(It is important to note that we have the guarantee that γ_0 exists because we assume unbounded capacity channels. Assuming channels with a bounded capacity c, no configuration satisfies (2) if there are at least two distinct processes p and q such that $|MesSeq_p^q| > c$.)

As $\mathcal{P}$ is snap-stabilizing, γ_0 is a possible initial configuration of $\mathcal{P}$. To obtain the contradiction, we now show that there is an execution starting from γ_0 that does not satisfy $\mathcal{SP}$. By definition, $\phi(\gamma_0) = \alpha_0$. Consider a process p and the two first configurations of e_p^1: β_0^p and β_1^p. Any message that p receives in $\beta_0^p \mapsto \beta_1^p$ can be received by p in the first step from γ_0: $\gamma_0 \mapsto \gamma_1$. Now, $\phi_p(\gamma_0) = \phi_p(\beta_0^p)$. So, p can behave in $\gamma_0 \mapsto \gamma_1$ as in $\beta_0^p \mapsto \beta_1^p$. In this case, $\phi_p(\gamma_1) = \phi_p(\beta_1^p)$. Hence, if each process q behaves in $\gamma_0 \mapsto \gamma_1$ as in the first step of its execution factor e_q^1, we obtain a configuration γ_1 such that $\phi(\gamma_1) = \alpha_1$. By induction principle, there exists an execution prefix starting from γ_0 denoted *PRED* such that $\Phi(PRED) = \mathtt{BAD}$. As $\mathcal{P}$ is snap-stabilizing, there exists an execution *SUFF* that starts from the last configuration of *PRED*. Now, merging *PRED* and *SUFF* we obtain an execution of $\mathcal{P}$ that does not satisfy $\mathcal{SP}$ — this contradicts the fact that $\mathcal{P}$ is snap-stabilizing. □

The proof of Theorem 1 hinges on the fact that after some transient faults the configuration may contain an unbounded number of arbitrary messages. Note that a safety-distributed specification involves more than one process and thus requires the processes to communicate to ensure that safety is not violated. However, with unbounded channels, each process cannot determine if the incoming message is indeed sent by its neighbor or is the result of faults. Thus, the communication is thwarted and the processes cannot differentiate safe and unsafe behavior.

4. Message-passing systems with bounded capacity channels

We now consider systems with bounded capacity channels. In such systems, we assume that if a process sends a message in a channel that is full, then the message is lost. For the sake of simplicity, we restrict our study to systems with single-message capacity channels. The extension to an arbitrary but known bounded message capacity is straightforward (by applying the principles described in [29,6,7]).

Below, we propose two snap-stabilizing protocols for fully-connected networks (Algorithms 1 and 3) respectively for the PIF (*Propagation of Information with Feedback*) and mutual exclusion problems. The mutual exclusion algorithm is obtained using several PIFs.

4.1. PIF for fully-connected networks

4.1.1. Principle

Informally, the PIF algorithm (also called *echo* algorithm) can be described as follows: upon a request, a process – called *initiator* – starts the first phase of the PIF by broadcasting a data message m into the network (the *broadcast phase*). Then, each non-initiator *acknowledges*[3] to the initiator the receipt of m (the *feedback phase*). The PIF terminates by a *decision event* at the initiator.[4] This decision is taken following the acknowledgments for m, meaning that when the decision event for the message m occurs, the last acknowledgments the initiator delivers from all other processes are acknowledgments for m.

Note that any process may need to initiate a PIF. Thus, any process can be the initiator of a PIF and several PIFs may run concurrently. Hence, any PIF protocol has to cope with concurrent PIFs.

More formally, the PIF problem can be specified as follows:

Specification 1 (*PIF-Exec*)**.** *An execution e satisfies Predicate* PIF-Exec *if and only if e satisfies the following two properties:*

(1) **Start.** *Upon a request to broadcast a message m, a process p starts a PIF of m.*
(2) **User-Safety.** *During any PIF of some message m started by p:*
   - *Every process other than p receives m.*
   - *p receives acknowledgments for m from all other processes.*
   - *p executes the decision event in finite time, this decision is taken following the acknowledgments for m.*

We now outline a snap-stabilizing implementation of the PIF called $\mathcal{PIF}$ (the code is provided in Algorithm 1). In the following (and in the rest of the paper), the *message-values* will be replaced by "—" when they have no impact on the reasoning.

A basic PIF implementation requires the following input/output variables:

- The variable $\mathtt{Req}_p$ is used to manage the requests at the process p. The value of $\mathtt{Req}_p$ belongs to $\{\mathtt{Wait}, \mathtt{In}, \mathtt{Done}\}$. $\mathtt{Req}_p = \mathtt{Wait}$ means that a PIF is requested. $\mathtt{Req}_p = \mathtt{In}$ means that the protocol is currently executing a PIF. $\mathtt{Req}_p = \mathtt{Done}$ means no PIF is under execution, i.e., the protocol is waiting for the next request.
- The buffer variable $\mathtt{BMes}_p$ is used to store the message to broadcast.
- The array $\mathtt{FMes}_p[1 \ldots n-1]$ is used to store acknowledgments, i.e., $\mathtt{FMes}_p[q]$ contains the acknowledgment for the broadcast message coming from q.

Using these variables, the protocol proceeds as follows: assume that a user wants to broadcast a message m from process p. It waits until the current PIF terminates (i.e., until $\mathtt{Req}_p = \mathtt{Done}$) even if the current PIF is due to a fault, and then notifies the request to p by setting $\mathtt{BMes}_p$ to m and $\mathtt{Req}_p$ to $\mathtt{Wait}$. Consequently to this request, a PIF is started (in particular, $\mathtt{Req}_p$ is set to $\mathtt{In}$). The current PIF terminates when $\mathtt{Req}_p$ is set to $\mathtt{Done}$ (this latter assignment corresponds to the decision event). Between the start and the termination, the protocol has to generate two types of events at the application level. First, a "**B-receive** $\langle m \rangle$ **from** p" event at each other process q. When this event occurs, the application at q is assumed to process the broadcast message m and put an acknowledgment Ack_m into $\mathtt{FMes}_q[p]$. The protocol then transmits $\mathtt{FMes}_q[p]$ to p: this generates a "**F-receive** $\langle Ack_m \rangle$ **from** q" event at p so that the application at p can access the acknowledgment.

Note that the protocol has to operate correctly despite arbitrary messages in the channels left after the faults. Note also that messages may be lost. To counter the message loss the protocol repeatedly sends duplicate messages. To deal with the arbitrary initial messages and the duplicates, we mark each message with a flag that ranges from 0 to 4. Two arrays are used to manage the flag values:

- In $\mathtt{State}_p[q]$, process p stores a flag value that it attaches to the messages it sends to its qth neighbor.
- In $\mathtt{NState}_p[q]$, p stores the flag of the last message it has accepted from its qth neighbor.

Using these two arrays, our protocol proceeds as follows: when p starts a PIF, it initializes $\mathtt{State}_p[q]$ to 0, for every index q. The PIF terminates when $\mathtt{State}_p[q] \geq 4$ for every index q.

During the PIF, p repeatedly sends $\langle \mathtt{PIF}, \mathtt{BMes}_p, -, \mathtt{State}_p[q], - \rangle$ to every process q such that $\mathtt{State}_p[q] < 4$. When a process q receives $\langle \mathtt{PIF}, B, -, pState, - \rangle$ from p, q updates $\mathtt{NState}_q[p]$ to $pState$. Then, if $pState < 4$, q sends $\langle \mathtt{PIF}, -, \mathtt{FMes}_q[p], -, \mathtt{NState}_q[p] \rangle$ to p. Finally, p increments $\mathtt{State}_p[q]$ only when it receives a $\langle \mathtt{PIF}, -, F, -, qNState \rangle$ message from q such that $qNState = \mathtt{State}_p[q]$ and $qNState < 4$.

The main idea behind the algorithm is as follows: assume that p starts to broadcast the message m. While $\mathtt{State}_p[q] < 4$, $\mathtt{State}_p[q]$ is incremented only when p received a message $\langle \mathtt{PIF}, -, F, -, qNState \rangle$ from q such that $qNState = \mathtt{State}_p[q]$. So, $\mathtt{State}_p[q]$ will be equal to 4 only after p successively receives $\langle \mathtt{PIF}, -, F, -, qNState \rangle$ messages from q with the flag values 0, 1, 2, and 3. Now, initially there is at most one message in the channel from p to q and at most another one in the channel from q to p. So these messages can only cause at most two increments of $\mathtt{State}_p[q]$. Finally, the arbitrary initial value of $\mathtt{NState}_q[p]$ can cause at most one increment of $\mathtt{State}_p[q]$. Hence, since $\mathtt{State}_p[q] = 3$, we have the guarantee that p will increment $State_p[q]$ to 4 only after it receives a message sent by q after q receives a message sent by p. That is, this message is a correct acknowledgment of m by q.

It remains to describe the generation of the **B-receive** and **F-receive** events:

- Any process q receives at least four copies of the broadcast message from p. But, q generates a **B-receive** event only once for each broadcast message from p: when q switches $\mathtt{NState}_q[p]$ to 3.
- After it starts, p is sure to receive the correct feedback from q since it receives from q a $\langle \mathtt{PIF}, -, F, -, qNState \rangle$ message such that $qNState = \mathtt{State}_p[q] = 3$. As previously, to limit the number of events, p generates a **F-receive** event only when it switches $\mathtt{State}_p[q]$ from 3 to 4. The next copies are ignored.

4.1.2. Correctness

Below, we prove that Algorithm $\mathcal{PIF}$ is a snap-stabilizing PIF algorithm for fully-connected networks. Note that the principle of proof is similar to [21]. However, the proof details are quite different, mainly due to the fact that our communication model is weaker than the one used in [21]. Remember that in [21], authors abstract the activity of communication links by assuming an underline snap-stabilizing ARQ data link algorithm. Here, we just assume that links are fair-lossy and FIFO.

The proof of snap-stabilization of $\mathcal{PIF}$ consists in showing that, despite the arbitrary initial configuration, any execution of $\mathcal{PIF}$ always satisfies the **start** and the **user-safety** properties of Specification 1.

Considering an arbitrary initial configuration, we state the **start** property (Corollary 1) in two steps:

(S1) We first prove that each time a user wants to broadcast a message from some process p, then it can eventually submit its request to the process (i.e. it is eventually enabled to execute $\mathtt{Req}_p \leftarrow \mathtt{Wait}$).
(S2) We then prove that once a request has been submitted to some process p, the process starts (i.e., executes action $\mathtt{A}_1$) the corresponding PIF within finite time.

To prevent the aborting of a previous PIF, a user can initiate a request at some process p only if $\mathtt{Req}_p = \mathtt{Done}$. Hence, to show (S1), we show that from any configuration where $\mathtt{Req}_p \in \{\mathtt{Wait}, \mathtt{In}\}$, the system eventually reaches a configuration where $\mathtt{Req}_p = \mathtt{Done}$. This latter claim is proven in two stages:

- We first show in Lemma 1 that from any configuration where $\mathtt{Req}_p = \mathtt{Wait}$, in finite time the system reaches a configuration where $\mathtt{Req}_p = \mathtt{In}$.
- We then show in Lemma 3 that from any configuration where $\mathtt{Req}_p = \mathtt{In}$, in finite time the system reaches a configuration where $\mathtt{Req}_p = \mathtt{Done}$.

Lemma 1. *Let p be any process. From any configuration where $\mathtt{Req}_p = \mathtt{Wait}$, in finite time the system reaches a configuration where $\mathtt{Req}_p = \mathtt{In}$.*

Proof. When $\mathtt{Req}_p = \mathtt{Wait}$, action $\mathtt{A}_1$ is continuously enabled at p and by executing $\mathtt{A}_1$, p sets $\mathtt{Req}_p$ to $\mathtt{In}$. □

The next technical lemma is used in the proof of Lemma 3.

Lemma 2. *Let p and q be two distinct processes. From any configuration where $(\mathtt{Req}_p = \mathtt{In}) \wedge (\mathtt{State}_p[q] < 4)$, $\mathtt{State}_p[q]$ is incremented in finite time.*

Proof. Assume, for the sake of contradiction, that $\mathtt{Req}_p = \mathtt{In}$ and $\mathtt{State}_p[q] = i$ with $i < 4$ but $\mathtt{State}_p[q]$ is never incremented. Then, $\mathtt{Req}_p = \mathtt{In}$ and $\mathtt{State}_p[q] = i$ hold forever and by checking actions $\mathtt{A}_2$ and $\mathtt{A}_3$, we know that:

- p only sends to q messages of the form $\langle \mathtt{PIF}, -, -, i, - \rangle$.
- p sends such messages infinitely many times.

As a consequence, q eventually only receives from p messages of the form $\langle \mathtt{PIF}, -, -, i, - \rangle$ and q receives such messages infinitely often. By action $\mathtt{A}_3$, $\mathtt{NState}_q[p] = i$ eventually holds forever. From that point, any message that q sends to p is of the form $\langle \mathtt{PIF}, -, -, -, i \rangle$. Also, as $i < 4$ and q receives infinitely many messages from p, q sends infinitely many messages of the form $\langle \mathtt{PIF}, -, -, -, i \rangle$ to p. Hence, p eventually receives $\langle \mathtt{PIF}, -, -, -, i \rangle$ from q and, as a consequence, increments $\mathtt{State}_p[q]$ (see action $\mathtt{A}_3$) — a contradiction. □

Lemma 3. *Let p be any process. From any configuration where $\mathtt{Req}_p = \mathtt{In}$, in finite time the system reaches a configuration where $\mathtt{Req}_p = \mathtt{Done}$.*

Proof. Assume, for the sake of contradiction, that from some configuration $\mathtt{Req}_p \neq \mathtt{Done}$ forever. Then, $\mathtt{Req}_p = \mathtt{In}$ eventually holds forever by Lemma 1. Now, by Lemma 2 and owing the fact that for every index q, $\mathtt{State}_p[q]$ cannot decrease while $\mathtt{Req}_p = \mathtt{In}$, we can deduce that p eventually satisfies "$\forall q \in [1 \ldots n-1]$, $\mathtt{State}_p[q] = 4$" forever. In this case, p eventually sets $\mathtt{Req}_p$ to $\mathtt{Done}$ by action $\mathtt{A}_2$ — a contradiction. □

As explained before, Lemmas 1 and 3 proves (S1). Lemma 1 also implies (S2) because $\mathtt{A}_1$ (the starting action) is the only action where $\mathtt{Req}_p$ is set to $\mathtt{In}$. Hence, we have the following corollary:

Corollary 1 (*Start*)**.** *Starting from any configuration, $\mathcal{PIF}$ always satisfies the **start** property of Specification 1.*

Still considering an arbitrary initial configuration, we now state the **user-safety** property (Corollary 2), that is, during any PIF of some message m started by p:

(U1) Every process other than p receives m.
(U2) p receives acknowledgments for m from all other processes.
(U3) p executes the decision event[5] in finite time, this decision is taken following the acknowledgments for m.

We first show (U1) and (U2) in Lemma 5.
The next technical lemma is used in the proof of Lemma 5.

[3] An acknowledgment is a message sent by the receiving process to inform the sender about data it has correctly received (*cf.* [27]).

[4] That is, an event that causally depends on an action at each process (this definition comes from [27]).

[5] Remember that the decision event corresponds to the statement $\mathtt{Req}_p \leftarrow \mathtt{Done}$.

Algorithm 1 Protocol $\mathcal{PIF}$ for any process p

Constant: n: integer

Variables:

$\texttt{Req}_p \in \{\texttt{Wait}, \texttt{In}, \texttt{Done}\}$: input/output
$\texttt{BMes}_p, \texttt{FMes}_p[1 \dots n-1]$: inputs
$\texttt{State}_p[1 \dots n-1] \in \{0,1,2,3,4\}^{n-1}, \texttt{NState}_p[1 \dots n-1] \in \{0,1,2,3,4\}^{n-1}$: internals

Actions:

$\texttt{A}_1$ :: $(\texttt{Req}_p = \texttt{Wait}) \rightarrow$
  $\texttt{Req}_p \leftarrow \texttt{In}$ /* Start */
  for all $q \in [1 \dots n-1]$ **do** $\texttt{State}_p[q] \leftarrow 0$

$\texttt{A}_2$ :: $(\texttt{Req}_p = \texttt{In}) \rightarrow$
  if $(\forall q \in [1 \dots n-1], \texttt{State}_p[q] = 4)$ **then**
    $\texttt{Req}_p \leftarrow \texttt{Done}$ /* Termination */
  else
    for all $q \in [1 \dots n-1]$ **do**
      if $(\texttt{State}_p[q] < 4)$ **then**
        send$\langle \texttt{PIF}, \texttt{BMes}_p, \texttt{FMes}_p[q], \texttt{State}_p[q], \texttt{NState}_p[q] \rangle$ **to** q

$\texttt{A}_3$ :: **receive**$\langle \texttt{PIF}, B, F, qState, pState \rangle$ **from** channel q $\rightarrow$
  if $(\texttt{NState}_p[q] \neq 3) \wedge (qState = 3)$ **then**
    generate a "**B-receive**$\langle B \rangle$ **from** channel q" event
  $\texttt{NState}_p[q] \leftarrow qState$
  if $(\texttt{State}_p[q] = pState) \wedge (\texttt{State}_p[q] < 4)$ **then**
    $\texttt{State}_p[q] \leftarrow \texttt{State}_p[q] + 1$
    if $(\texttt{State}_p[q] = 4)$ **then**
      generate a "**F-receive**$\langle F \rangle$ **from** channel q" event
  if $(qState < 4)$ **then**
    send$\langle \texttt{PIF}, \texttt{BMes}_p, \texttt{FMes}_p[q], \texttt{State}_p[q], \texttt{NState}_p[q] \rangle$ **to** q

Lemma 4. *Let p and q be two distinct processes. After p starts a PIF (action $\texttt{A}_1$), p switches $\texttt{State}_p[q]$ from 2 to 3 only if the three following conditions hold:*

1. *Any message in the channel from p to q is of the form $\langle \texttt{PIF}, -, -, i, - \rangle$ with $i \neq 3$.*
2. *$\texttt{NState}_q[p] \neq 3$.*
3. *Any message in the channel from q to p is of the form $\langle \texttt{PIF}, -, -, -, j \rangle$ with $j \neq 3$.*

Proof. p starts a PIF with action $\texttt{A}_1$. By executing $\texttt{A}_1$, p sets $\texttt{State}_p[q]$ to 0. From that point, $\texttt{State}_p[q]$ can only be incremented one by one until reaching value 4. Let us study the three first increments of $\texttt{State}_p[q]$:

- **From 0 to 1.** $\texttt{State}_p[q]$ switches from 0 to 1 only after p receives a message $\langle \texttt{PIF}, -, -, -, 0 \rangle$ from q. As the link $\{p, q\}$ always contains at most one message in the channel from q to p, the next message that p will receive from q will be a message sent by q.
- **From 1 to 2.** From the previous case, we know that $\texttt{State}_p[q]$ switches from 1 to 2 only when p receives $\langle \texttt{PIF}, -, -, -, 1 \rangle$ from q and this message was sent by q. From actions $\texttt{A}_2$ and $\texttt{A}_3$, we can then deduce that $\texttt{NState}_q[p] = 1$ held when q sent $\langle \texttt{PIF}, -, -, -, 1 \rangle$ to p. From that point, $\texttt{NState}_q[p] = 1$ holds until q receives from p a message of the form $\langle \texttt{PIF}, -, -, i, - \rangle$ with $i \neq 1$.
- **From 2 to 3.** The switching of $\texttt{State}_p[q]$ from 2 to 3 can occurs only after p receives a message $mes_1 = \langle \texttt{PIF}, -, -, -, 2 \rangle$ from q. Now, from the previous case, we can deduce that p receives mes_1 consequently to the reception by q of a message $mes_0 = \langle \texttt{PIF}, -, -, 2, - \rangle$ from p. Now:
  (a) As the link $\{p, q\}$ always contains at most one message in the channel from p to q, after receiving mes_0 and until $\texttt{State}_p[q]$ switches from 2 to 3, every message in transit from p to q is of the form $\langle \texttt{PIF}, -, -, i, - \rangle$ with $i \neq 3$ (Condition 1 of the lemma) because after p starts to broadcast a message, p sends messages of the form $\langle \texttt{PIF}, -, -, 3, - \rangle$ to q only when $\texttt{State}_p[q] = 3$.
  (b) After receiving mes_0, $\texttt{NState}_q[p] \neq 3$ until q receives $\langle \texttt{PIF}, -, -, 3, - \rangle$. Hence, by (a), after receiving mes_0 and until (at least) $\texttt{State}_p[q]$ switches from 2 to 3, $\texttt{NState}_q[p] \neq 3$ (Condition 2 of the lemma).
  (c) After receiving mes_1, $\texttt{State}_p[q] \neq 3$ until p receives $\langle \texttt{PIF}, -, -, -, 3 \rangle$ from q. As p receives mes_1 after q receives mes_0, by (b) we can deduce that after receiving mes_1 and until (at least) $\texttt{State}_p[q]$ switches from 2 to 3, every message in transit from q to p is of the form $\langle \texttt{PIF}, -, -, -, j \rangle$ with $j \neq 3$ (Condition 3 of the lemma).

Hence, when p switches $\texttt{State}_p[q]$ from 2 to 3, the three conditions (1), (2) and (3) are satisfied, which proves the lemma. □

Lemma 5. *Starting from any configuration, if p starts a PIF of some message m (action $\texttt{A}_1$), then:*

- *All other process eventually receive m.*
- *p eventually receives acknowledgments for m from all other processes.*

Proof. p starts a PIF of m by executing action $\texttt{A}_1$: p switches $\texttt{Req}_p$ from Wait to In and sets $\texttt{State}_p[q]$ to 0, for every index q. Then, $\texttt{Req}_p$ remains equal to In until p decides by setting $\texttt{Req}_p$ to Done. Now, p decides in finite time by Lemma 3 and when p decides, we have $\texttt{State}_p[q] = 4, \forall q \in [1 \dots 0]$ (action $\texttt{A}_2$). From the code of Algorithm 1, this means that for every index q, $\texttt{State}_p[q]$ is incremented one by one from 0 to 4. By Lemma 4, for every index q, $\texttt{State}_p[q]$ is incremented from 3 to 4 only after:

- q receives a message sent by p of the form $\langle \texttt{PIF}, m, -, 3, - \rangle$, and then
- p receives a message sent by q of the form $\langle \texttt{PIF}, -, -, 3, - \rangle$.

When q receives $\langle \texttt{PIF}, m, -, 3, - \rangle$ from p for the first time, it generates the event "**B-receive** $\langle m \rangle$ **from** channel p" and then starts to send $\langle \texttt{PIF}, -, F, -, 3 \rangle$ messages to p.[6] From that point and until p decides, q only receives $\langle \texttt{PIF}, m, -, 3, - \rangle$ message from p. So, from that point and until p decides, any message that q sends to p acknowledges the reception of m. Since, p receives the first $\langle \texttt{PIF}, -, F, -, 3 \rangle$ message from q, p generates a "**F-receive** $\langle F \rangle$ **from** channel q" event and then sets $\texttt{State}_p[q]$ to 4.

[6] q sends a $\langle \texttt{PIF}, -, F, -, 3 \rangle$ message to p (at least) each time it receives a $\langle \texttt{PIF}, m, -, 3, - \rangle$ message from p.

Hence, for every process q, the broadcast of m generates a "**B-receive** $\langle m \rangle$ **from** channel p" event at q and the associated "**F-receive** $\langle F \rangle$ **from** channel q" event at p, which proves the lemma. □

The next lemma proves (U3).

Lemma 6. *Starting from any configuration, during any PIF of some message m started by p, (1) p executes a decision event (i.e., $\texttt{Req}_p \leftarrow \texttt{Done}$) in finite time and (2) this decision is taken following the acknowledgments for m.*

Proof. First, during any PIF of some message m started by p, p decides in finite time by Lemma 3.

Let q be a process such that $q \neq p$. We now show the second part of the lemma by proving that *between the start of the PIF of m and the corresponding decision, p generates exactly one* "**F-receive** $\langle F \rangle$ **from** channel q" *event where F is an acknowledgment sent by q for m.*

First p starts a PIF of m by executing action $\texttt{A}_1$: p switches $\texttt{Req}_p$ from Wait to In and sets $\texttt{State}_p[q]$ to 0. Then, $\texttt{Req}_p$ remains equal to In until p decides by setting $\texttt{Req}_p$ to Done. When p decides, we have $\texttt{State}_p[q] = 4$, for every index q. From the code of Algorithm 1, we know that exactly one "**F-receive** $\langle F \rangle$ **from** channel q" event occurs at p before p decides: when p switches $\texttt{State}_p[q]$ from 3 to 4. Lemma 4 implies that F is an acknowledgment for m sent by q and the lemma is proven. □

By Lemmas 5 and 6, follows:

Corollary 2 (*User-Safety*)**.** *Starting from any configuration, $\mathcal{PIF}$ always satisfies the **User-Safety** property of Specification 1.*

By Corollaries 1 and 2, follows:

Theorem 2. *$\mathcal{PIF}$ is snap-stabilizing to Specification 1.*

Below, we give an additional property of $\mathcal{PIF}$, this property will be used in the snap-stabilization proof of $\mathcal{ME}$ (Section 4.2).

Property 1. *If p starts a PIF in the configuration γ_0 and the PIF terminates at p in the configuration γ_k, then any message that was in a channel from and to p in γ_0 is no longer in the channel in γ_k.*

Proof. Assume that a process p starts a PIF in the configuration γ_0. Then, as $\mathcal{PIF}$ is snap-stabilizing to Specification 1, we have the guarantee that for every p'neighbor q, at least one broadcast message crosses the channel from p to q and at least one acknowledgment message crosses the channel from q to p during the PIF-computation. Now, we assumed that each channel has a single-message capacity. Hence, every message that was in a channel from and to p in the configuration γ_0 has been received or lost when the PIF terminates at p in configuration γ_k. □

4.2. Mutual exclusion for fully-connected networks

4.2.1. Specification

We now consider the problem of *mutual exclusion*. The mutual exclusion specification requires that a special section of code, called the *critical section*, is executed by at most one process at any time. A snap-stabilizing mutual exclusion protocol (only) guarantees its safety property when the process requests the critical section after the faults stop [11]. The safety property is not otherwise guaranteed. Hence, we specify the mutual exclusion protocol as follows:

Specification 2 (*ME-Exec*)**.** *An execution e satisfies Predicate ME-Exec if and only if e satisfies the following two properties:*

1. **Start.** *Upon a request, a process enters the critical section in finite time.*
2. **User-Safety.** *If a requested process enters the critical section, then it executes the critical section alone.*

In order to simplify the design of our mutual exclusion algorithm, we propose below an identifier-discovery algorithm, $\mathcal{IDL}$, that is a straightforward extension of $\mathcal{PIF}$.

4.2.2. Identifier-discovery

$\mathcal{IDL}$ assumes that each process has a unique identifier (ID_p denotes the identifier of the process p) and uses three variables at each process p:

- $\texttt{Req}_p \in \{\texttt{Wait}, \texttt{In}, \texttt{Done}\}$. The purpose of this variable is the same as in $\mathcal{PIF}$.
- $\texttt{minID}_p$. After a complete computation of $\mathcal{IDL}$, $\texttt{minID}_p$ contains the minimal identifier of the system.
- $\texttt{IDTab}_p[1 \dots n]$. After a complete computation of $\mathcal{IDL}$, $\texttt{IDTab}_p[q] = ID_q$.

When requested at p, $\mathcal{IDL}$ evaluates the identifiers of all other processes and the minimal identifier of the system using $\mathcal{PIF}$. The results of the computation are available for p since p decides. Based on the specification of $\mathcal{PIF}$, it is easy to see that $\mathcal{IDL}$ is snap-stabilizing to the following specification:

Specification 3 (*ID-Discovery-Exec*)**.** *An execution e satisfies Predicate ID-Discovery-Exec if and only if e satisfies the following two properties:*

1. **Start.** *When requested, a process p starts in finite time to discover the identifiers of the processes.*
2. **User-Safety.** *Any identifier-discovery started by p terminates in finite time by a decision event at p and when the decision occurs, we have:*
   - $\forall q \in [1 \dots n-1], \texttt{IDTab}_p[q] = ID_q$.
   - $\texttt{minID}_p = \min(\{ID_q, q \in [1 \dots n-1]\} \cup \{ID_p\})$.

Theorem 3. *$\mathcal{IDL}$ is snap-stabilizing for Specification 3.*

4.2.3. Principle

We now describe a snap-stabilizing mutual exclusion protocol called $\mathcal{ME}$ (Algorithm 3). $\mathcal{ME}$ also uses the variable Req with the same meaning as previously: $\mathcal{ME}.\texttt{Req}_p$ is to Wait when the process p is requested to execute the critical section. Process p is then called a *requestor* and we assume that $\mathcal{ME}.\texttt{Req}_p$ cannot be set to Wait again until $\mathcal{ME}.\texttt{Req}_p = \texttt{Done}$, i.e., until its current request is done.

The main idea behind the protocol is the following: we assume identifiers on processes and the process with the smallest identifier – called the *leader* – bounces a single token to every process using a variable called Val, this variable ranges over $\{0 \dots n-1\}$. The Val variable of the leader $\mathcal{L}$ designates which process holds the token: process p holds the token if and only if either $p = \mathcal{L}$ and $\texttt{Val}_{\mathcal{L}} = 0$ or $p \neq \mathcal{L}$ and $\texttt{Val}_{\mathcal{L}}$ is equal to the number of its channel incoming from p. A process can access the critical section only if it holds the token. Thus, the processes continuously ask the leader to know if they hold the token.

When a process learns that it holds the token:

(1) It first ensures that no other process can execute the critical section (due to the arbitrary initial configuration, some other processes may wrongly believe that they also hold the token).
(2) It then executes the critical section if it wishes to (it may refuse if it is not a requestor).
(3) Finally, it notifies to the leader that it has terminated Step (2) so that the leader passes the token to another process.

To apply this scheme, $\mathcal{ME}$ is executed in phases from Phase 0 to 4 in such way that each process goes through Phase 0 infinitely often. After a request, a process p can access the critical section only after executing Phase 0: indeed p can access the critical section only if $\mathcal{ME}.\texttt{Req}_p = \texttt{In}$ and p switches $\mathcal{ME}.\texttt{Req}_p$ from Wait to In only in Phase 0. Hence, our protocol just ensures that after executing Phase

Algorithm 2 Protocol $1\mathcal{DL}$ for any process p

Constants: n, ID_p: integers

Variables:

$\texttt{Req}_p \in \{\texttt{Wait}, \texttt{In}, \texttt{Done}\}$	:	input/output
$\texttt{minID}_p \in \mathbb{N}$, $\texttt{IDTab}_p[1 \ldots n-1] \in \mathbb{N}^{n-1}$	:	outputs

Actions:

A_1	::	$(\texttt{Req}_p = \texttt{Wait})$	$\rightarrow$	$\texttt{Req}_p \leftarrow \texttt{In}$ /* Start */ $\texttt{minID}_p \leftarrow ID_p$; $\mathcal{P}1\mathcal{F}.\texttt{BMes}_p \leftarrow \texttt{IDL}$ $\mathcal{P}1\mathcal{F}.\texttt{Req}_p \leftarrow \texttt{Wait}$
A_2	::	$(\texttt{Req}_p = \texttt{In}) \wedge (\mathcal{P}1\mathcal{F}.\texttt{Req}_p = \texttt{Done})$	$\rightarrow$	$\texttt{Req}_p \leftarrow \texttt{Done}$ /* Termination */
A_3	::	**B-receive**$(\texttt{IDL})$ **from** channel q	$\rightarrow$	$\mathcal{P}1\mathcal{F}.\texttt{FMes}_p[q] \leftarrow ID_p$
A_4	::	**F-receive**(ID) **from** channel q	$\rightarrow$	$\texttt{IDTab}_p[q] \leftarrow ID$; $\texttt{minID}_p \leftarrow \min(\texttt{minID}_p, ID)$

0, a process always executes the critical section alone. Below, we describe the five phases of our protocol:

Phase 0. When a process p is in Phase 0, it requests a computation of $1\mathcal{DL}$ to collect the identifiers of all processes and to evaluate which one is the leader. It also sets $\mathcal{ME}.\texttt{Req}_p$ to In if $\mathcal{ME}.\texttt{Req}_p = \texttt{Wait}$. It then switches to Phase 1.

Phase 1. When a process p is in Phase 1, p waits for the termination of $1\mathcal{DL}$. Then, p requests a PIF of the message ASK to know if it is the token holder and switches to Phase 2. Upon receiving a message ASK from the channel p, any process q answers YES if $\texttt{Val}_q = p$, NO otherwise. Of course, p will only take the answer of the leader into account.

Phase 2. When a process p is in Phase 2, it waits for the termination of the PIF requested in Phase 1. After the PIF terminates, p knows if it is the token holder. If p holds the token, it requests a PIF of the message EXIT and switches to Phase 3. The goal of this message is to force all other processes to restart to Phase 0. This ensures that no other process believes to be the token holder when p accesses the critical section. Indeed, due to the arbitrary initial configuration, some process $q \neq p$ may believe to be the token holder, if q never starts Phase 0. On the contrary, after restarting to 0, q cannot receive positive answer from the leader until p notifies to the leader that it releases the critical section.

Phase 3. When a process p is in Phase 3, it waits for the termination of the current PIF. After the PIF terminates, if p is the token holder, then:

1) p executes the critical section and switches $\mathcal{ME}.\texttt{Req}_p$ from In to Done if $\mathcal{ME}.\texttt{Req}_p = \texttt{In}$, and then
2) (a) Either, p is the leader and switches $\texttt{Val}_p$ from 0 to 1.
   (b) Or, p is not the leader and requests a PIF of the message EXITCS to notify to the leader that it releases the critical section. Upon receiving such a message, the leader increments its variable Val modulus $n+1$ to pass the token to another process.

In any case, p terminates Phase 3 by switching to Phase 4.

Phase 4. When a process p is in Phase 4, it waits for the termination of the last PIF and then switches to Phase 0.

4.2.4. Correctness

We begin the proof of snap-stabilization of $\mathcal{ME}$ by showing that, despite the arbitrary initial configuration, any execution of $\mathcal{ME}$ always satisfies the **user-safety** property of Specification 2.

Assume that a process p is a requestor, i.e., $\mathcal{ME}.\texttt{Req}_p = \texttt{Wait}$. Then, p cannot enters the critical section before executing action A_0. Indeed:

- p enters the critical section only if $\mathcal{ME}.\texttt{Req}_p = \texttt{In}$, and
- action A_0 is the only action of $\mathcal{ME}$ allowing p to set $\mathcal{ME}.\texttt{Req}_p$ to In.

Hence, to show the **user-safety** property of Specification 2 (Corollary 3), we have to prove that, despite the initial configuration, after p executes action A_0, if p enters the critical section, then it executes the critical section alone (Lemma 9).

Lemma 7. *Let p be a process. Starting from any configuration, after p executes A_0, if p enters the critical section, then all other processes have switched to Phase 0 at least once.*

Proof. After p executes A_0, to enter the critical section (in A_3) p must execute the three actions A_1, A_2, and A_3 successively. Also, to execute the critical section in action A_3, p must satisfy the predicate *Winner*(p). The value of the predicate *Winner*(p) depends on (1) the $1\mathcal{DL}$ computation requested in A_0 and (2) the PIF of the message ASK requested in A_1. Now, these two computations are done when p executes A_2. So, the fact that p satisfies *Winner*(p) when executing A_3 implies that p also satisfies *Winner*(p) when executing A_2. As a consequence, p requests a PIF of the message EXIT in A_2. Now, p executes A_3 only after this PIF terminates. Hence, p executes A_3 only after every other process executes A_6 (i.e., the feedback of the message EXIT): by this action, every other process switches to Phase 0. □

Definition 6 (*Leader*)**.** We call *Leader* the process with the smallest identifier. In the following, this process will be denoted by $\mathcal{L}$.

Definition 7 (*Favour*)**.** We say that the process p *favours* the process q if and only if $(p = q \wedge \texttt{Val}_p = 0) \vee (p \neq q \wedge \texttt{Val}_p = q)$.

Lemma 8. *Let p be a process. Starting from any configuration, after p executes A_0, p enters the critical section only if $\mathcal{L}$ favours p until p releases the critical section.*

Proof. By checking all the actions of Algorithm 3, we can observe that after p executes A_0, to enter the critical section p must execute the four actions A_0, A_1, A_2, and A_3 successively. Moreover, p executes a complete $1\mathcal{DL}$-computation between A_0 and A_1. Thus:

(1) $1\mathcal{DL}.\texttt{minID}_p = ID_{\mathcal{L}}$ when p executes A_3.
(2) Also, from the configuration where p executes A_1, all messages in the channels from and to p have been sent after p requests $1\mathcal{DL}$ in action A_0 (Property 1, page 17).

Let us now study the following two cases:

- $p = \mathcal{L}$. In this case, when p executes A_3, to enter the critical section p must satisfy $\texttt{Val}_p = \texttt{Val}_{\mathcal{L}} = 0$ by (1). This means that $\mathcal{L}$ favours p (actually itself) when p enters the critical section. Moreover, as the execution of A_3 is atomic, $\mathcal{L}$ favours p until p releases the critical section and this closes the case.

Algorithm 3 Protocol $\mathcal{ME}$ for any process p

Constants: n, ID_p: integers

Variables:

$\texttt{Req}_p \in \{\texttt{Wait}, \texttt{In}, \texttt{Done}\}$	:	input/output
$\texttt{Ph}_p \in \{0, 1, 2, 3, 4\}$, $\texttt{Val}_p \in \{0 \ldots n-1\}$, $\texttt{Answers}_p[1 \ldots n-1] \in \{true, false\}^{n-1}$	:	internals

Predicate:

$Winner(p) \equiv (1\mathcal{DL}.\texttt{minID}_p = ID_p \wedge \texttt{Val}_p = 0) \vee (\exists q \in [1 \ldots n-1], \texttt{Answers}_p[q] \wedge 1\mathcal{DL}.\texttt{IDTab}_p[q] = 1\mathcal{DL}.\texttt{minID}_p)$

Actions:

A_0	::	$(\texttt{Ph}_p = 0)$	$\rightarrow$	$1\mathcal{DL}.\texttt{Req}_p \leftarrow \texttt{Wait}$ **if** $\texttt{Req}_p = \texttt{Wait}$ **then** $\texttt{Req}_p \leftarrow \texttt{In}$ /* Start */ $\texttt{Ph}_p \leftarrow \texttt{Ph}_p + 1$
A_1	::	$(\texttt{Ph}_p = 1) \wedge (1\mathcal{DL}.\texttt{Req}_p = \texttt{Done})$	$\rightarrow$	$\mathcal{P}1\mathcal{F}.\texttt{BMes}_p \leftarrow \texttt{ASK}$; $\mathcal{P}1\mathcal{F}.\texttt{Req}_p \leftarrow \texttt{Wait}$ $\texttt{Ph}_p \leftarrow \texttt{Ph}_p + 1$
A_2	::	$(\texttt{Ph}_p = 2) \wedge (\mathcal{P}1\mathcal{F}.\texttt{Req}_p = \texttt{Done})$	$\rightarrow$	**if** *Winner*(p) **then** $\mathcal{P}1\mathcal{F}.\texttt{BMes}_p \leftarrow \texttt{EXIT}$; $\mathcal{P}1\mathcal{F}.\texttt{Req}_p \leftarrow \texttt{Wait}$ $\texttt{Ph}_p \leftarrow \texttt{Ph}_p + 1$

A_3	::	$(\texttt{Ph}_p = 3) \wedge (\mathcal{P}1\mathcal{F}.\texttt{Req}_p = \texttt{Done})$	$\rightarrow$	**if** *Winner*(p) **then**
				if $\texttt{Req}_p = \texttt{In}$ **then**
				critical section; $\texttt{Req}_p \leftarrow \texttt{Done}$ /* Termination */
				if $1\mathcal{DL}.\texttt{minID}_p = ID_p$ **then**
				$\texttt{Val}_p \leftarrow 1$
				else
				$\mathcal{P}1\mathcal{F}.\texttt{BMes}_p \leftarrow \texttt{EXITCS}$; $\mathcal{P}1\mathcal{F}.\texttt{Req}_p \leftarrow \texttt{Wait}$
				$\texttt{Ph}_p \leftarrow \texttt{Ph}_p + 1$

A_4	::	$(\texttt{Ph}_p = 4) \wedge (\mathcal{P}1\mathcal{F}.\texttt{Req}_p = \texttt{Done})$	$\rightarrow$	$\texttt{Ph}_p \leftarrow 0$

A_5	::	**B-receive**$(\texttt{ASK})$ **from** channel q	$\rightarrow$	**if** $\texttt{Val}_p = q$ **then**
				$\mathcal{P}1\mathcal{F}.\texttt{FMes}_p[q] \leftarrow \texttt{YES}$
				else
				$\mathcal{P}1\mathcal{F}.\texttt{FMes}_p[q] \leftarrow \texttt{NO}$

A_6	::	**B-receive**$(\texttt{EXIT})$ **from** channel q	$\rightarrow$	$\texttt{Ph}_p \leftarrow 0$; $\mathcal{P}1\mathcal{F}.\texttt{FMes}_p[q] \leftarrow \texttt{OK}$
A_7	::	**B-receive**$(\texttt{EXITCS})$ **from** channel q	$\rightarrow$	**if** $\texttt{Val}_p = q$ **then** $\texttt{Val}_p \leftarrow (\texttt{Val}_p + 1) \bmod (n+1)$ $\mathcal{P}1\mathcal{F}.\texttt{FMes}_p[q] \leftarrow \texttt{OK}$
A_8	::	**F-receive**$(\texttt{YES})$ **from** channel q	$\rightarrow$	$\texttt{Answers}_p[q] \leftarrow true$
A_9	::	**F-receive**$(\texttt{NO})$ **from** channel q	$\rightarrow$	$\texttt{Answers}_p[q] \leftarrow false$
A_{10}	::	**F-receive**$(\texttt{OK})$ **from** channel q	$\rightarrow$	/* do nothing */

- $p \neq \mathcal{L}$. In this case, when p executes A_3, p satisfies $1\mathcal{DL}.\texttt{minID}_p = ID_{\mathcal{L}}$ by (1). So, p executes the critical section only if $\exists q \in [1 \ldots n-1]$ such that $1\mathcal{DL}.\texttt{IDTab}_p[q] = ID_{\mathcal{L}} \wedge \texttt{Answers}_p[q] = true$ (see Predicate *Winner*(p)). For that, p must receive a feedback message YES from $\mathcal{L}$ during the PIF of the message ASK requested in action A_1. Now, $\mathcal{L}$ sends such a feedback to p only if $\texttt{Val}_{\mathcal{L}} = p$ when the "**B-receive**$\langle$ASK$\rangle$ **from** p" event occurs at $\mathcal{L}$ (see action A_5). Also, since $\mathcal{L}$ satisfies $\texttt{Val}_{\mathcal{L}} = p$, $\mathcal{L}$ updates $\texttt{Val}_p$ only after receiving an EXITCS message from p (see action A_7). Now, by (2), after $\mathcal{L}$ feedbacks YES to p, $\mathcal{L}$ receives an EXITCS message from p only if p broadcasts EXITCS to $\mathcal{L}$ after releasing the critical section (see action A_3). Hence, $\mathcal{L}$ favours p until p releases the critical section and this closes the case. □

Lemma 9. *Let p be a process. Starting from any configuration, if p enters the critical section after executing A_0, then it executes the critical section alone.*

Proof. Assume, for the sake of contradiction, that p enters the critical section after executing A_0 but executes the critical section concurrently with another process q. Then, q also executes action A_0 before executing the critical section by Lemma 7. By Lemma 8, we have the following two property:

- $\mathcal{L}$ favours p during the whole period where p executes the critical section.
- $\mathcal{L}$ favours q during the whole period where q executes the critical section.

This contradicts the fact that p and q executes the critical section concurrently because $\mathcal{L}$ always favours exactly one process at a time. □

Corollary 3 (*User-Safety*)**.** *Starting from any configuration, $\mathcal{ME}$ always satisfies the* **user-safety** *property of Specification 2.*

We now show that, despite the arbitrary initial configuration, any execution of $\mathcal{ME}$ always satisfies the **start** property of Specification 2 (Lemma 4). As previously, this proof is made in two steps:

(S1) We first prove that each time a user want to execute the critical section at some process p, then it is eventually able to submit its request to the process (i.e. it is eventually enabled to execute $\mathcal{ME}.\texttt{Req}_p \leftarrow \texttt{Wait}$).
(S2) We then prove that once a request has been submitted to some process p, the process enters the critical section in finite time.

To prevent the aborting of the previous request, a user can submit a request at some process p only if $\mathcal{ME}.\texttt{Req}_p = \texttt{Done}$. Hence, to show (S1), we show that from any configuration where $\mathcal{ME}.\texttt{Req}_p \in \{\texttt{Wait}, \texttt{In}\}$, the system eventually reaches a

configuration where $\mathcal{ME}.\mathtt{Req}_p = \mathtt{Done}$. This latter claim is proven in two stages:

- We first show in Lemma 11 that from any configuration where $\mathcal{ME}.\mathtt{Req}_p = \mathtt{Wait}$, in finite time the system reaches a configuration where $\mathcal{ME}.\mathtt{Req}_p = \mathtt{In}$.
- We then show in Lemma 13 that from any configuration where $\mathcal{ME}.\mathtt{Req}_p = \mathtt{In}$, in finite time the system reaches a configuration where $\mathtt{Req}_p = \mathtt{Done}$.

The next technical lemma is used in the proof of Lemma 11.

Lemma 10. *Starting from any configuration, every process p switches to Phase* 0 *infinitely often.*

Proof. Consider the following two cases:

- "**B-receive**⟨EXIT⟩" events occur at p infinitely often. Then, each time such an event occurs at p, p switches to Phase 0 (see $\mathtt{A}_6$) and this closes the case.
- *Only a finite number of* "**B-receive**⟨EXIT⟩" *events occurs at p.* In this case, p eventually reaches a configuration from which it no longer executes action $\mathtt{A}_6$. From this configuration, $\mathtt{Ph}_p$ can only be incremented modulus 5 and depending of the value of $\mathtt{Ph}_p$, we have the following possibilities:
  - $\mathtt{Ph}_p = 0$. In this case, $\mathtt{A}_0$ is continuously enabled at p. Hence, p eventually sets $\mathtt{Ph}_p$ to 1 (see action $\mathtt{A}_0$).
  - $\mathtt{Ph}_p = i$ with $0 < i \leq 4$. In this case, action $\mathtt{A}_i$ is eventually continuously enabled due to the termination property of $\mathcal{1DL}$ and $\mathcal{PIF}$. By executing $\mathtt{A}_i$, p increments $\mathtt{Ph}_p$ modulus 5.

  Hence, if only a finite number of "**B-receive**⟨EXIT⟩" events occurs at p, then $\mathtt{Ph}_p$ is incremented modulus 5 infinitely often and this closes the case. □

Lemma 11. *Let p be any process. From any configuration where* $\mathcal{ME}.\mathtt{Req}_p = \mathtt{Wait}$, *in finite time the system reaches a configuration where* $\mathcal{ME}.\mathtt{Req}_p = \mathtt{In}$.

Proof. Assume that $\mathcal{ME}.\mathtt{Req}_p = \mathtt{Wait}$. Lemma 10 implies that p eventually executes action $\mathtt{A}_0$. By action $\mathtt{A}_0$, $\mathcal{ME}.\mathtt{Req}_p$ is set to In. □

The next technical lemma is used in the proof of Lemma 13.

Lemma 12. *Starting from any configuration,* $\mathtt{Val}_{\mathcal{L}}$ *is incremented modulus* $n + 1$ *infinitely often.*

Proof. Assume, for the sake of contradiction, that there are a finite number of increments of $\mathtt{Val}_{\mathcal{L}}$ (modulus $n + 1$). We can then deduce that $\mathcal{L}$ eventually favours some process p forever.

In order to prove the contradiction, we first show that (*) *assuming that* $\mathcal{L}$ *favours p forever, only a finite number of* "**B-receive**⟨EXIT⟩" *events occurs at p.* Towards this end, assume, for the sake of contradiction, that an infinite number of "**B-receive**⟨EXIT⟩" events occurs at p. Then, as the number of processes is finite, there is a process $q \neq p$ that broadcasts EXIT messages infinitely often. Now, every PIF terminates in finite time. So, q performs infinitely many PIF of the message EXIT. In order to start another PIF of the message EXIT, q must then successively execute actions $\mathtt{A}_0, \mathtt{A}_1, \mathtt{A}_2$. Now, when q executes $\mathtt{A}_2$ after $\mathtt{A}_0$ and $\mathtt{A}_1$, $\mathcal{1DL}.\mathtt{minID}_q = ID_{\mathcal{L}}$ and either (1) $q = \mathcal{L}$ and, as $q \neq p, \mathtt{Val}_{\mathcal{L}} \neq 0$, or (2) $\mathcal{L}$ has feedback NO to the PIF of the message ASK started by q because $\mathtt{Val}_{\mathcal{L}} = p \neq q$. In both cases, q satisfies $\neg$*Winner*(q) and, as a consequence, does not broadcast EXIT (see action $\mathtt{A}_3$). Hence, q eventually stops to broadcast EXIT — a contradiction.

Using Property (*), we now show the contradiction. By Lemma 10, p switches to Phase 0 infinitely often. By (*), we know that p eventually stops executing action $\mathtt{A}_6$. So, from the code of Algorithm 3, we can deduce that p eventually successively executes actions $\mathtt{A}_0$, $\mathtt{A}_1$, $\mathtt{A}_2$, $\mathtt{A}_3$, and $\mathtt{A}_4$ infinitely often. Consider the first time p successively executes $\mathtt{A}_0$, $\mathtt{A}_1$, $\mathtt{A}_2$, $\mathtt{A}_3$, and $\mathtt{A}_4$ and study the following two cases:

- $p = \mathcal{L}$. Then, $\mathtt{Val}_p = 0$ and $\mathcal{1DL}.\mathtt{minID}_p = ID_p$ when p executes $\mathtt{A}_3$ because p executes a complete $\mathcal{1DL}$-computation between $\mathtt{A}_0$ and $\mathtt{A}_1$ and $\mathcal{1DL}$ is snap-stabilizing to Specification 3. Hence, p updates $\mathtt{Val}_p$ to 1 when executing $\mathtt{A}_3$ — a contradiction.
- $p \neq \mathcal{L}$. Then, $\mathcal{1DL}.\mathtt{minID}_p = ID_{\mathcal{L}}$ when p executes $\mathtt{A}_3$ because p executes a complete $\mathcal{1DL}$-computation between $\mathtt{A}_0$ and $\mathtt{A}_1$ and $\mathcal{1DL}$ is snap-stabilizing to Specification 3. Also, p receives YES from $\mathcal{L}$ because p executes a complete PIF of the message ASK between $\mathtt{A}_1$ and $\mathtt{A}_2$ and $\mathcal{PIF}$ is snap-stabilizing to Specification 1. Hence, p satisfies the predicate *Winner*(p) when executing $\mathtt{A}_3$ and, as a consequence, requests a PIF of the message EXITCS in action $\mathtt{A}_3$. This PIF terminates when p executes $\mathtt{A}_4$: from this point on, we have the guarantee that $\mathcal{L}$ has executed action $\mathtt{A}_7$. Now, by $\mathtt{A}_7$, $\mathcal{L}$ increments $\mathtt{Val}_{\mathcal{L}}$ — a contradiction. □

Lemma 13. *Let p be any process. From any configuration where* $\mathcal{ME}.\mathtt{Req}_p = \mathtt{In}$, *in finite time the system reaches a configuration where* $\mathcal{ME}.\mathtt{Req}_p = \mathtt{Done}$.

Proof. Assume, for the sake of contradiction, that from a configuration where $\mathtt{Req}_p = \mathtt{In}$ the system never reaches a configuration where $\mathtt{Req}_p = \mathtt{Done}$. From the code of Algorithm 3, we can then deduce that $\mathtt{Req}_p = \mathtt{In}$ holds forever. In this case there are two possibilities:

- p no longer executes $\mathtt{A}_3$, or
- p satisfies $\neg$*Winner*(p) each time it executes $\mathtt{A}_3$.

Consider then the following two cases:

- $p = \mathcal{L}$. Then, $\mathtt{Val}_p \neq 0$ eventually holds forever — a contradiction to Lemma 12.
- $p \neq \mathcal{L}$. In this case, p no longer starts any PIF of the message EXITCS. Now, every PIF terminates in finite time. Hence, eventually there is no more "**B-receive**⟨EXITCS⟩ **from** p" event at $\mathcal{L}$. As a consequence, $\mathtt{Val}_{\mathcal{L}}$ eventually no longer switches from value p to $(p + 1) \mod (n + 1)$ — which contradicts Lemma 12. □

As explained before, Lemmas 11 and 13 proves (S1). Lemma 13 also implies (S2) because a process switches $\mathcal{ME}.\mathtt{Req}_p$ from In to Done only after executing the critical section. Hence, we have the following corollary:

Corollary 4 (*Start*)**.** *Starting from any configuration,* $\mathcal{ME}$ *always satisfies the* ***start*** *property of Specification 2.*

By Corollaries 3 and 4, follows:

Theorem 4. $\mathcal{ME}$ *is snap-stabilizing to Specification 2.*

5. Conclusion

We addressed the problem of *snap-stabilization* in one-hop message-passing systems and presented matching negative and positive results. On the negative side, we showed that *snap-stabilization* is impossible for a wide class of specifications – namely, the *safety-distributed* specifications – in message-passing systems where the channel capacity is finite yet unbounded. On the positive side, we showed that *snap-stabilization* is possible (even for *safety-distributed* specifications) in message-passing systems if we assume a bound on the channel capacity. The proof is constructive, as we presented the first three snap-stabilizing protocols for message-passing systems with a bounded channel capacity. These protocols respectively solve the PIF and mutual exclusion problem in a fully-connected network.

On the theoretical side, it is worth observing that the results presented in this paper can be extended to general topologies using the approach presented in [21], and then to general specifications that admit a Katz and Perry transformer [26]. Yet, the possible extension to networks where nodes are subject to permanent, i.e., crash faults, remains open. On the practical side, our results imply the possibility of implementing snap-stabilizing protocols on real networks. Actually implementing them is a future challenge.

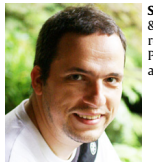
References

[1] Y. Afek, A. Bremler-Barr, Self-stabilizing unidirectional network algorithms by power supply, Chicago J. Theor. Comput. Sci. (1998).
[2] Y. Afek, G.M. Brown, Self-stabilization over unreliable communication media, Distributed Computing 7 (1) (1993) 27–34.
[3] B. Alpern, F.B. Schneider, Recognizing safety and liveness, Distributed Computing 2 (3) (1987) 117–126.
[4] A. Arora (Ed.), 1999 ICDCS Workshop on Self-stabilizing Systems, Austin, Texas, June 5, 1999, Proceedings, IEEE Computer Society, 1999.
[5] A. Arora, M. Nesterenko, Unifying stabilization and termination in message-passing systems, Distributed Computing 17 (3) (2005) 279–290. http://dx.doi.org/10.1007/s00446-004-0111-6.
[6] B. Awerbuch, S. Kutten, Y. Mansour, B. Patt-Shamir, G. Varghese, A time-optimal self-stabilizing synchronizer using a phase clock, IEEE Trans. Dependable Sec. Comput. 4 (3) (2007) 180–190.
[7] B. Awerbuch, B. Patt-Shamir, G. Varghese, Self-stabilization by local checking and correction (extended abstract), in: FOCS, IEEE, 1991, pp. 268–277.
[8] D. Bein, A.K. Datta, V. Villain, Snap-stabilizing optimal binary search tree, in: Herman and Tixeuil [24], pp. 1–17.
[9] A. Bui, A.K. Datta, F. Petit, V. Villain, State-optimal snap-stabilizing pif in tree networks, in: Arora [4], pp. 78–85.
[10] A. Bui, A.K. Datta, F. Petit, V. Villain, Snap-stabilization and pif in tree networks, Distributed Computing 20 (1) (2007) 3–19. http://dx.doi.org/10.1007/s00446-007-0030-4.
[11] A. Cournier, A.K. Datta, F. Petit, V. Villain, Enabling snap-stabilization, in: ICDCS, IEEE Computer Society, 2003, pp. 12–19.
[12] A. Cournier, S. Devismes, F. Petit, V. Villain, Snap-stabilizing depth-first search on arbitrary networks, Comput. J. 49 (3) (2006) 268–280.
[13] A. Cournier, S. Devismes, V. Villain, Snap-stabilizing pif and useless computations, in: ICPADS (1), IEEE Computer Society, 2006, pp. 39–48.
[14] A. Cournier, S. Devismes, V. Villain, A snap-stabilizing dfs with a lower space requirement, in: Herman and Tixeuil [24], pp. 33–47.
[15] A. Cournier, S. Devismes, V. Villain, Snap-stabilizing detection of cutsets, in: D.A. Bader, M. Parashar, S. Varadarajan, V.K. Prasanna (Eds.), HiPC, in: Lecture Notes in Computer Science, vol. 3769, Springer, 2005, pp. 488–497.
[16] S. Delaët, S. Devismes, M. Nesterenko, S. Tixeuil, Snap-stabilization in message-passing systems, in: International Conference on Distributed Systems and Networks, ICDCN 2009, in: LNCS, No. 5404, 2009, pp. 281–286. https://hal.inria.fr/inria-00248465.
[17] S. Delaët, B. Ducourthial, S. Tixeuil, Self-stabilization with r-operators revisited, Journal of Aerospace Computing, Information, and Communication (2006).
[18] E.W. Dijkstra, Self-stabilizing systems in spite of distributed control, Commun. ACM 17 (11) (1974) 643–644.
[19] S. Dolev, Self-stabilization, MIT Press, 2000.
[20] S. Dolev, T. Herman, Superstabilizing protocols for dynamic distributed systems, Chicago J. Theor. Comput. Sci. (1997).
[21] S. Dolev, N. Tzachar, Empire of colonies: self-stabilizing and self-organizing distributed algorithms, in: A.A. Shvartsman (Ed.), OPODIS, in: Lecture Notes in Computer Science, vol. 4305, Springer, 2006, pp. 230–243.
[22] S. Ghosh, A. Bejan, A framework of safe stabilization, in: S.-T. Huang, T. Herman (Eds.), Self-Stabilizing Systems, in: Lecture Notes in Computer Science, vol. 2704, Springer, 2003, pp. 129–140.
[23] M.G. Gouda, N.J. Multari, Stabilizing communication protocols, IEEE Trans. Computers 40 (4) (1991) 448–458.
[24] T. Herman, S. Tixeuil (Eds.), Self-stabilizing systems, in: 7th International Symposium, SSS 2005, Barcelona, Spain, October 26–27, 2005, Proceedings, in: Lecture Notes in Computer Science, vol. 3764, Springer, 2005.
[25] R.R. Howell, M. Nesterenko, M. Mizuno, Finite-state self-stabilizing protocols in message-passing systems, in: Arora [4], pp. 62–69.
[26] S. Katz, K.J. Perry, Self-stabilizing extensions for message-passing systems., Distributed Computing 7 (1) (1993) 17–26.
[27] G. Tel, Introduction to Distributed Algorithms, second ed., Cambridge University Press, Cambridge, UK, 2001.
[28] S. Tixeuil, Self-stabilizing algorithms, in: Algorithms and Theory of Computation Handbook, second ed., in: Chapman & Hall/CRC Applied Algorithms and Data Structures, CRC Press, 2009, pp. 26.1–26.45. Taylor & Francis Group. http://www.crcpress.com/product/isbn/9781584888185.
[29] G. Varghese, Self-stabilization by counter flushing, SIAM J. Comput. 30 (2) (2000) 486–510.
[30] G. Varghese, M. Jayaram, The fault span of crash failures, J. ACM 47 (2) (2000) 244–293.

Sylvie Delaët is an associate professor at the University of Paris Sud-XI (France). She received her Ph.D. in 1995 from the University of Paris Sud-XI for her pioneering work on Self-stabilization. Her research interests include dynamic networks and systems with an emphasis on formal treatment.

Stéphane Devismes is an associate professor at the University Joseph Fourier of Grenoble (France). He is a member of the Synchronous Team of the VERIMAG Laboratory. He received his Ph.D. in 2006 from the University of Picardie Jules Verne (Amiens, France). He has carried out broad research in theoretical issues of distributed fault-tolerant computing, especially relating to self-stabilization.

Mikhail Nesterenko got his Ph.D. in 1998 from Kansas State University. Presently he is an associate professor at Kent State University. He is interested in wireless networking, distributed algorithms and fault-tolerance.



Sébastien Tixeuil is a full professor at the University Pierre & Marie Curie - Paris 6 (France), where he leads the NPA research group. He received his Ph.D. from University of Paris Sud-XI in 2000. His research interests include fault and attack tolerance in dynamic networks and systems.

Annexe B

Biographie de l'auteur

Une carrière réussie est une chose merveilleuse mais on ne peut pas se pelotonner contre elle la nuit quand on a froid l'hiver.

Marilyn Monrœ

Après des études de mathématiques, j'ai fait un doctorat d'Informatique à l'université Paris Sud évalué par les rapporteurs Jean-Michel Hélary (Professeur, Université Rennes I, *rapporteur*), Ted Herman (Professeur, Université d'Iowa, USA, *rapporteur absent à la soutenance*) et que j'ai soutenu en décembre 1995 devant un jury composé de Joffroy Beauquier (Professeur, Université Paris Sud, *directeur*), Marie-Claude Gaudel (Professeur, Université Paris Sud, *présidente*), Christian Lavault (Professeur, Université Paris 13, *examinateur*), Jean-Michel Hélary (Professeur, Université Rennes I, *rapporteur*).

FIGURE B.1 – Sylvie Delaët, avril 2013

Recrutée par la suite Maître de Conférences à l'Institut Universitaire de Technologie d'Orsay où j'enseigne toujours à plein temps, j'ai continué la recherche au sein du Laboratoire de Recherche en Informatique (LRI) en m'investissant dans l'administration de la recherche et de l'enseignement. J'ai eu des positions aussi variées que présidente du secteur enfance du comité d'entraide sociale de la faculté des sciences d'Orsay ou co-présidente d'un comité de programme dans la conférence SSS 2012 ; responsable de filière d'enseignement et responsable du comité d'organisation de la conférences SSS 2007 ; coordinatrice des portes ouvertes pour mon laboratoire ou coordonatrice locale d'un projet de recherche ANR, animatrice d'atelier de vulgarisation de la fête de la sciences ou animatrice d'un groupe d'enseignants pour la programmation objet en java. J'ai enseigné toute l'informatique en premier cycle, de la base de données au réseaux en passant par l'architecture des machines et la programmation impérative, fonctionnelle ou objet. En recherche mon centre principal d'intérêt a toujours été l'autostabilisation, j'ai participé à quelques comités de programmes et ai été experte pour de nombreux journaux et conférences. J'ai encadré trois post doctorants : Stéphane Devismes, Marisuz Rokicki et Partha Sarathi Mandal, j'ai co-encadré ou j'encadre deux doctorants : Peva Blanchard et Sammy Haddad ainsi que quatre étudiants de master recherche : Duy-So N'Guyen, Matthieu Rajœlison, Olga Melekhova et Peva Blanchard.

Ce chapitre liste de manière exhaustive les publications scientifiques que j'ai eu la fierté de co-écrire et de co-signer. Elles sont rangées selon leur type. Au sein de chacune des six sections, elles sont classées par ordre chronologique inverse de leur publication (ce qui ne correspond pas nécessairement à l'ordre dans lequel les travaux ont été réalisés). Les articles dans les revues sont pour la plupart des reprises approfondies et détaillées de communication(s) dans les conférences sous le même titre. Seule « Stabilité des arbres des plus courts chemins en présence de concurrence » est un travail totalement original n'ayant donné lieu à aucune communication préalable à sa publication en revue.

B.1 Articles dans des revues internationales avec comité de lecture

1. Tight complexity analysis of population protocols with cover times - the ZebraNet example. Joffroy Beauquier, Peva Blanchard, Jannah Burman and Sylvie Delaët *Theoretical Computer Science (TCS)*, à paraître, disponible en ligne depuis le 31 octobre 2012.
2. Deterministic Secure Positioning in Wireless Sensor Networks. Sylvie Delaët, Partha Sarathi Mandal, Mariusz Rokicki, and Sébastien Tixeuil. *Theoretical Computer Science (TCS)*, 412(35) : 4471-4481, 2011.
3. Snap-Stabilization in Message-Passing Systems. Sylvie Delaët, Stéphane Devismes, Mikhail Nesterenko, and Sébastien Tixeuil. *Journal of Parallel and Distributed Computing (JPDC)*, 70(12) :1220-1230, December 2010.
4. Transient Fault Detectors. Joffroy Beauquier, Sylvie Delaët, Shlomi Dolev, and Sébastien Tixeuil. *Distributed Computing*, 20(1) :39-51, June 2007.
5. Self-stabilization with r-operators revisited. Sylvie Delaët, Bertrand Ducourthial, and Sébastien Tixeuil. *Journal of Aerospace Computing, Information, and Communication (JACIC)*, 3(10) :498-514, 2006.
6. Tolerating Transient and Intermittent Failures. Sylvie Delaët, and Sébastien Tixeuil. *Journal of Parallel and Distributed Computing (JPDC)*, 62(5) :961-981, May 2002.

B.2 Articles dans la revue francophone avec comité de lecture

1. Stabilité des arbres des plus courts chemins en présence de concurrence. Johanne Cohen, Sylvie Delaët. *Technique et Science Informatiques*, octobre 2011.
2. Un algorithme auto-stabilisant en dépit de communications non fiables. Sylvie Delaët, Sébastien Tixeuil. *Technique et Science Informatiques*, volume 17 numéro 5, Hermès, 1998.

B.3 Edition d'actes

1. Actes de la Journée « Réseaux et Algorithmes Répartis » . Sylvie Delaët, Thomas Hérault, Colette Johnen, Sébastien Tixeuil. Juin 2002, Université Paris Sud, 50 pages.

B.4 Articles dans des conférences internationales avec comité de lecture

1. Computing Time Complexity of Population Protocols with Cover Times - The ZebraNet Example. Joffroy Beauquier, Peva Blanchard, Janna Burman, Sylvie Delaët. In *International Conference on Stabilization, Safety, and Security (SSS 2011)*. pp. 47-61.

2. Safer Than Safe : On the Initial State of Self-stabilizing Systems. Sylvie Delaët, Shlomi Dolev, and Olivier Peres. In *International Conference on Stabilization, Safety, and Security (SSS 2009)*. pp. 775-776.

3. Snap-Stabilization in Message-Passing Systems. Sylvie Delaët, Stéphane Devismes, Mikhail Nesterenko, and Sébastien Tixeuil. In *International Conference on Distributed Systems and Networks (ICDCN 2009)*, pages 281-286, January 2009.

4. Safe and Eventually Safe : Comparing Self-stabilizing and Non-stabilizing Algorithms on a Common Ground. Sylvie Delaët, Shlomi Dolev, Olivier Peres. OPODIS 2009 *International Conference on Principles of Distributed Computing (OPODIS'2009)*, pages 315-329, December 2009

5. Brief Announcement : Snap-Stabilization in Message-Passing Systems. Sylvie Delaët, Stéphane Devismes, Mikhail Nesterenko, and Sébastien Tixeuil . In Proceedings of the *ACM Conference on Principles of Distributed Computing (PODC 2008)*, August 2008

6. Deterministic Secure Positioning in Wireless Sensor Networks. Sylvie Delaët, Partha Sarathi Mandal, Mariusz Rokickil, and Sébastien Tixeuil. In *Proceedings of the ACM/IEEE International Conference on Distributed Computing in Sensor Networks (DCOSS 2008)*, Santorini, Greece, June 2008.

7. Universe Detectors for Sybil Defense in Ad Hoc Wireles Networks. Adnam Vora, Mikhail Nesterenko, Sébastien Tixeuil, and Sylvie Delaët. In *International Conference on Stabilization, Safety, and Security (SSS 2008)*, November 2008.

8. A 1-strong self stabilizing transformer. Joffroy Beauquier, Sylvie Delaët, and Sammy Haddad. in *International Conference on Stabilization, Safety, and Security of distributed Systems (SSS 2006)*, Dallas, Texas, November 2006. LNCS.

9. Necessary and sufficient conditions for 1 adaptativity. Joffroy Beauquier, Sylvie Delaët, and Sammy Haddad. in *IEEE International conference on parallel and distributed Systems IPDPS 2006*, April 2006

10. Self-stabilization with r-operators revisited. Sylvie Delaët, Bertrand Ducourthial, and Sébastien Tixeuil. volume 3764 of *Lecture Notes in Computer Science*, Barcelona, Spain, October 2005. Springer Verlag.

11. Stability and Self-stabilization of BGP. Sylvie Delaët, Duy-So Nguyen, Sébastien Tixeuil. *RIVF 2003*, pp. 139-144, Hanoi, Vietnam, 2003. Les actes sont publiés dans un numéro spécial de la revue internationale *Studia Informatica Universalis*.

12. Tolerating Transient and Intermittent Failures. Sylvie Delaët, Sébastien Tixeuil. *International Conference on Principles of Distributed Computing (OPODIS'2000)*, pp. 17-36, Paris, France, Décembre 2000. Les actes sont publiés dans un numéro spécial de la revue internationale Studia Informatica Universalis.

13. Transient Fault Detectors. Joffroy Beauquier, Sylvie Delaët, Shlomi Dolev and Sébastien Tixeuil. *DISC'98*, LNCS 1499, pages 62-74, Grèce, Septembre 1998.
14. Classes of self stabilizing protocols. Joffroy Beauquier and Sylvie Delaët. In 4th Workshop on future Trends of distributed systems, pages 105-110, Lisbon, September 1995
15. Optimum Probabilistic self-stabilizing on uniform rings Joffroy Beauquier, Stéphane Cordier and Sylvie Delaët. in Workshop on Self Stabilizing Systems, Las Vegas, May 1995.
16. Probabilistic self-stabilizing mutaul exclusion in uniform rings. Joffroy Beauquier and Sylvie Delaët. In Proceedings of the *ACM Conference on Principles of Distributed Computing (PODC 1994)*, August 1994, page 398.

B.5 Articles dans des conférences nationales avec comité de lecture

1. Stabilisation Instantanée dans les Systèmes à Passage de Messages. Sylvie Delaët, Stéphane Devismes, Mikhail Nesterenko, and Sébastien Tixeuil. In *Proceedings of Algotel 2009*, 2009.
2. Stabilité et auto-stabilisation du Routage Inter-Domaine dans Internet. Sylvie Delaët, Duy-So Nguyen, Sébastien Tixeuil. *Rencontres Francophones sur l'Algorithmique des Télécommunications (Algotel 2003)*, Banyuls, France, 2003. Les actes sont publiés par les presses de l'INRIA.
3. Auto-stabilisation en Dépit de Communications non Fiables. Sylvie Delaët, Sébastien Tixeuil. *RenPar'9*, Lausanne, Suisse, Juin 1997.
4. Algorithme d'exclusion mutuelle auto-stabilisant sur un réseau unidirectionnel quelconque. Sylvie Delaët *Journées des jeunes chercheurs en systèmes Informatiques Répartis*, page 6, Rennes 1995.
5. Classes D'algorithmes auto-stabailisables. Sylvie Delaët *Journées des jeunes chercheurs en systèmes Informatiques Répatis*, page 105-110, Grenoble 1993.

B.6 Thèse

1. Auto-stabilisation : Modèle et applications à l'Exclusion mutuelle. Sylvie Delaët *Thèse de doctorat de l'université Paris Sud*, 191 pages, Orsay Décembre 1995.

Annexe C

AAA : Apparence, Appareil et Apparat

Quatre-quart au fromage frais

Dessert - Facile - Bon marché
Temps de préparation : 30 minutes
Temps de cuisson : 60 minutes
Ingrédients (pour 10 personnes) :
– 200 g de fromage frais à température ambiante
– 300 g de beurre ramoli
– 500 g de sucre
– 5 œufs
– 320 g de farine
– 1 cuillère à café d'extrait de vanille

Préparation de la recette : penser à préparer tous les ingrédients à l'avance pour qu'ils soient à température ambiante. Faire chauffer le four à 160 ° C (thermostat 5). Dans un saladier, mélanger le beurre ramolli et le fromage frais. Lorsque le mélange est crémeux ajouter le sucre et bien mélanger. Ajouter les œufs un à un en battant bien. Ajouter la farine en une fois et battre jusqu'à ce que le mélange soit homogène. Ajouter l'extrait de vanille. Verser dans un moule préalablement beurré et chemisé. Faire cuire à 160 ° C pendant une heure ou jusqu'à ce qu'un couteau planté au centre du gâteau ressorte sec. Laisser refroidir avant dégustation. Se conserve jusqu'à 4 jours au réfrigérateur.

Soufflé au fromage blanc

Dessert - Difficile - Bon marché
Temps de préparation : 30 minutes
Temps de cuisson : 30 minutes
Ingrédients (pour 10 personnes) :
– 200 g de fromage frais à température ambiante
– 300 g de beurre ramolli
– 500 g de sucre
– 5 œufs
– 400 g de farine
– Un demi sachet de levure chimique
– 1 cuillère à café d'extrait de vanille

Préparation de la recette : penser à préparer tous les ingrédients à l'avance, qu'ils soient à température ambiante. Faire chauffer le four à 160 ° C (thermostat 5). Dans un saladier mélanger le beurre ramolli et le fromage frais. Lorsque le mélange est crémeux ajouter le sucre et bien mélanger. Ajouter les jaune d'œufs un à un. Ajouter doucement la farine préalablement mélangée à la levure et battre jusqu'à ce que le mélange soit homogène. Ajouter l'extrait de vanille. Battre fermement les blancs en neige. Les incorporer délicatement à l'**appareil**. Verser dans un moule préalablement beurré et chemisé. Faire cuire à 160 ° C pendant une demi heure. Déguster immédiatement.

Bibliographie

[AAD+06] Dana Angluin, James Aspnes, Zoë Diamadi, Michael J. Fischer, and René Peralta. Computation in networks of passively mobile finite-state sensors. *Distributed Computing*, 18(4) :235–253, 2006.

[AAER07] Dana Angluin, James Aspnes, David Eisenstat, and Eric Ruppert. The computational power of population protocols. *Distributed Computing*, 20(4) :279–304, 2007.

[AAFJ08] Dana Angluin, James Aspnes, Michael J. Fischer, and Hong Jiang. Self-stabilizing population protocols. *TAAS*, 3(4), 2008.

[AD02] Yehuda Afek and Shlomi Dolev. Local stabilizer. *J. Parallel Distrib. Comput.*, 62(5) :745–765, 2002.

[AFJ06] Dana Angluin, Michael J. Fischer, and Hong Jiang. Stabilizing consensus in mobile networks. In Phillip B. Gibbons, Tarek F. Abdelzaher, James Aspnes, and Ramesh Rao, editors, *DCOSS*, volume 4026 of *Lecture Notes in Computer Science*, pages 37–50. Springer, 2006.

[AG94] Anish Arora and Mohamed G. Gouda. Distributed reset. *IEEE Trans. Computers*, 43(9) :1026–1038, 1994.

[AH93] Efthymios Anagnostou and Vassos Hadzilacos. Tolerating transient and permanent failures (extended abstract). In André Schiper, editor, *WDAG*, volume 725 of *Lecture Notes in Computer Science*, pages 174–188. Springer, 1993.

[AK93] Sudhanshu Aggarwal and Shay Kutten. Time optimal self-stabilizing spanning tree algorithms. In R. K. Shyamasundar, editor, *FSTTCS*, volume 761 of *Lecture Notes in Computer Science*, pages 400–410. Springer, 1993.

[APSV91] Baruch Awerbuch, Boaz Patt-Shamir, and George Varghese. Self-stabilization by local checking and correction (extended abstract). In *FOCS*, pages 268–277. IEEE, 1991.

[APSVD94] 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.

[Aro99] Anish Arora, editor. *1999 ICDCS Workshop on Self-stabilizing Systems, Austin, Texas, June 5, 1999, Proceedings*. IEEE Computer Society, 1999.

[AW98] H. Attiya and J. Welch. *Distributed Computing : Fundamentals, Simulations, and Advanced Topics*. McGraw-Hill Publishing Company, New York, May 1998. 6.

[BBBD12] J. Beauquier, P. Blanchard, J. Burman, and S. Delaët. Tight complexity analysis of population protocols with cover times — the zebranet example. *Theoretical Computer Science*, (0) :–, 2012.

[BCD95] 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.

[BCPV03] Christian Boulinier, Sébastien Cantarell, Franck Petit, and Vincent Villain. A note on a bounded self-stabilizing local mutual exclusion algorithm. Technical Report 2003-02, LARIA - Université de Picardie Jules Vernes, 2003.

[BCV03] Lélia Blin, Alain Cournier, and Vincent Villain. An improved snap-stabilizing pif algorithm. In Huang and Herman [HH03a], pages 199–214.

[BDBD13] Peva Blanchard, Shlomi Dolev, Joffroy Beauquier, and Sylvie Delaët. Self-stabilizing paxos. *CoRR*, abs/1305.4263, 2013.

[BDDT98] Joffroy Beauquier, Sylvie Delaët, Shlomi Dolev, and Sébastien Tixeuil. Transient fault detectors. In Shay Kutten, editor, *Proceedings of the International Conference on Distributed Computing (DISC 1998)*, pages 62–74, Andros, Greece, September 1998. Springer.

[BDDT07] Joffroy Beauquier, Sylvie Delaët, Shlomi Dolev, and Sébastien Tixeuil. Transient fault detectors. *Distributed Computing*, 20(1) :39–51, June 2007.

[BDKM96] Joffroy Beauquier, Oliver Debas, and Synnöve Kekkonen-Moneta. Fault-tolerant and self-stabilizing ring orientation. In Nicola Santoro and Paul G. Spirakis, editors, *SIROCCO*, pages 59–72. Carleton Scientific, 1996.

[BDPV99] Alain Bui, Ajoy Kumar Datta, Franck Petit, and Vincent Villain. State-optimal snap-stabilizing pif in tree networks. In Arora [Aro99], pages 78–85.

[BDPV07] Alain Bui, Ajoy Kumar Datta, Franck Petit, and Vincent Villain. Snap-stabilization and pif in tree networks. *Distributed Computing*, 20(1) :3–19, 2007.

[BDT99] Joffroy Beauquier, Ajoy Kumar Datta, and Sébastien Tixeuil. Self-stabilizing census with cut-through constraint. In Arora [Aro99], pages 70–77.

[BDV05] Doina Bein, Ajoy Kumar Datta, and Vincent Villain. Snap-stabilizing optimal binary search tree. In Herman and Tixeuil [HT05], pages 1–17.

[BGJ01] Joffroy Beauquier, Maria Gradinariu, and Colette Johnen. Cross-over composition - enforcement of fairness under unfair adversary. In Ajoy Kumar Datta and Ted Herman, editors, *WSS*, volume 2194 of *Lecture Notes in Computer Science*, pages 19–34. Springer, 2001.

[BGJ07] Joffroy Beauquier, Maria Gradinariu, and Colette Johnen. Randomized self-stabilizing and space optimal leader election under arbitrary scheduler on rings. *Distributed Computing*, 20(1) :75–93, January 2007.

[BGKP99] Steven C. Bruell, Sukumar Ghosh, Mehmet Hakan Karaata, and Sriram V. Pemmaraju. Self-stabilizing algorithms for finding centers and medians of trees. *SIAM J. Comput.*, 29(2) :600–614, 1999.

[BGM93] James E. Burns, Mohamed G. Gouda, and Raymond E. Miller. Stabilization and pseudo-stabilization. *Distributed Computing*, 7(1) :35–42, 1993.

[BJM06] Joffroy Beauquier, Colette Johnen, and Stéphane Messika. Computing automatically the stabilization time against the worst and the best schedulers. Technical Report 1448, Université Paris-Sud XI, May 2006.

[BKM97] Joffroy Beauquier and Synnöve Kekkonen-Moneta. On ftss-solvable distributed problems. In Ghosh and Herman [GH97], pages 64–79.

[BP89] James E. Burns and Jan K. Pachl. Uniform self-stabilizing rings. *ACM Trans. Program. Lang. Syst.*, 11(2) :330–344, 1989.

[BPBRT10] Lélia Blin, Maria Potop-Butucaru, Stéphane Rovedakis, and Sébastien Tixeuil. Universal loop-free super-stabilization. In *Proceedings of the International Conference on Stabilization, Safety, and Security in Distributed Systems (SSS 2010)*, Lecture Notes in Computer Science, New York, NY, USA, September 2010. Springer Berlin / Heidelberg.

[BPV04] Christian Boulinier, Franck Petit, and Vincent Villain. About bounded self-stabilizing local mutual exclusion algorithms. Technical Report 2004-02, LARIA - Université de Picardie Jules Vernes, 2004.

[CD11] Johanne Cohen and Sylvie Delaët. Stabilité des arbres des plus courts chemins en présence de concurrence. *Technique et Science Informatiques*, 30(10) :1167–1189, 2011.

[CDGT08] Johanne Cohen, Anurag Dasgupta, Sukumar Ghosh, and Sébastien Tixeuil. An exercise in selfish stabilization. *ACM Transactions of Adaptive Autonomous Systems (TAAS)*, 3(4), November 2008.

[CDL+10] Alain Cournier, Swan Dubois, Anissa Lamani, Franck Petit, and Vincent Villain. Snap-stabilizing linear message forwarding. In *Proceedings of the International Conference on Stabilization, Safety, and Security in Distributed Systems (SSS 2010)*, pages 546–559, 2010.

[CDPT07] Eddy Caron, Frédéric Desprez, Franck Petit, and Cédric Tedeschi. Snap-stabilizing prefix tree for peer-to-peer systems. In Masuzawa and Tixeuil [MT07a], pages 82–96.

[CDPV03] Alain Cournier, Ajoy Kumar Datta, Franck Petit, and Vincent Villain. Enabling snap-stabilization. In *ICDCS*, pages 12–19. IEEE Computer Society, 2003.

[CDPV06] Alain Cournier, Stéphane Devismes, Franck Petit, and Vincent Villain. Snap-stabilizing depth-first search on arbitrary networks. *Comput. J.*, 49(3) :268–280, 2006.

[CDV05a] Alain Cournier, Stéphane Devismes, and Vincent Villain. Snap-stabilizing detection of cutsets. In David A. Bader, Manish Parashar, Sridhar Varadarajan, and Viktor K. Prasanna, editors, *HiPC*, volume 3769 of *Lecture Notes in Computer Science*, pages 488–497. Springer, 2005.

[CDV05b] Alain Cournier, Stéphane Devismes, and Vincent Villain. A snap-stabilizing dfs with a lower space requirement. In Herman and Tixeuil [HT05], pages 33–47.

[CDV06] Alain Cournier, Stéphane Devismes, and Vincent Villain. From self- to snap- stabilization. In Datta and Gradinariu [DG06], pages 199–213.

[CDV09a] Alain Cournier, Swan Dubois, and Vincent Villain. How to improve snap-stabilizing point-to-point communication space complexity ? In *Proceedings of the International Conference on Stabilization, Safety, and Security in Distributed Systems (SSS 2009)*, pages 195–208, 2009.

[CDV09b] Alain Cournier, Swan Dubois, and Vincent Villain. A snap-stabilizing point-to-point communication protocol in message-switched networks. In *IPDPS*, pages 1–11, 2009.

[CFPS12] Mark Cieliebak, Paola Flocchini, Giuseppe Prencipe, and Nicola Santoro. Distributed computing by mobile robots : Gathering. *SIAM J. Comput.*, 41(4) :829–879, 2012.

[CGR11] Christian Cachin, Rachid Guerraoui, and Luís Rodrigues. *Introduction to Reliable and Secure Distributed Programming (2. ed.)*. Springer, 2011.

[CHT96] Tushar Deepak Chandra, Vassos Hadzilacos, and Sam Toueg. The weakest failure detector for solving consensus. *J. ACM*, 43(4) :685–722, 1996.

[CIW12] Shukai Cai, Taisuke Izumi, and Koichi Wada. How to prove impossibility under global fairness : On space complexity of self-stabilizing leader election on a population protocol model. *Theory Comput. Syst.*, 50(3) :433–445, 2012.

[CJ003] Optimized link state routing protocol (olsr), 2003.

[CL85] K. Mani Chandy and Leslie Lamport. Distributed snapshots : Determining global states of distributed systems. *ACM Trans. Comput. Syst.*, 3(1) :63–75, 1985.

[CM84] K.M. Chandy and J. Misra. The drinking philosophers problem. *ACM Transactions on Programming Languages and Systems*, 6(4) :632–646, October 1984.

[Coh09] Johanne Cohen. Théorie des graphes et de l'approximation pour le routeage, la coloration et l'apprentissage d'équilibres. Technical report, UNIVERSITE DE VERSAILLES SAINT-QUENTIN EN-YVELINES, December 2009.

[CT96] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2) :225–267, 1996.

[dC05] M. de Cervantes. *El Ingenioso Hidalgo Don Quichotte de la Mancha*. 1605.

[DDNT10] Sylvie Delaët, Stéphane Devismes, Mikhail Nesterenko, and Sébastien Tixeuil. Snap-stabilization in message-passing systems. *Journal of Parallel and Distributed Computing (JPDC)*, 70(12) :1220–1230, December 2010.

[DDP09] Sylvie Delaët, Shlomi Dolev, and Olivier Peres. Safe and eventually safe : Comparing self-stabilizing and non-stabilizing algorithms on a common ground. In Tarek F. Abdelzaher, Michel Raynal, and Nicola Santoro, editors, *OPODIS*, volume 5923 of *Lecture Notes in Computer Science*, pages 315–329. Springer, 2009.

[DDT99] Sajal Das, Ajoy Kumar Datta, and Sébastien Tixeuil. Self-stabilizing algorithms in dag structured networks. *Parallel Processing Letters (PPL)*, 9(4) :563–574, December 1999.

[DDT06] Sylvie Delaët, Bertrand Ducourthial, and Sébastien Tixeuil. Self-stabilization with r-operators revisited. *Journal of Aerospace Computing, Information, and Communication (JACIC)*, 3(10) :498–514, 2006.

[Del95] Sylvie Delaët. *Auto-stabilisation : Modèle et Applications à l'Exclusion Mutuelle*. PhD thesis, Université Paris Sud, December 1995.

[DFP+12] Shantanu Das, Paola Flocchini, Giuseppe Prencipe, Nicola Santoro, and Masafumi Yamashita. The power of lights : Synchronizing asynchronous robots using visible bits. In *ICDCS*, pages 506–515. IEEE, 2012.

[DFSY10] Shantanu Das, Paola Flocchini, Nicola Santoro, and Masafumi Yamashita. On the computational power of oblivious robots : forming a series of geometric patterns. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC 2010)*, pages 267–276, 2010.

[DG06] Ajoy Kumar Datta and Maria Gradinariu, editors. *Stabilization, Safety, and Security of Distributed Systems, 8th International Symposium, SSS 2006, Dallas, TX, USA, November 17-19, 2006, Proceedings*, volume 4280 of *Lecture Notes in Computer Science*. Springer, 2006.

[DGFG05] Carole Delporte-Gallet, Hugues Fauconnier, and Rachid Guerraoui. What dependability for networks of mobile sensors ? In *First Workshop on Hot Topics in System Dependability (in conjunction with DSN)*, Yokohama, June 2005.

[DGFGR06] Carole Delporte-Gallet, Hugues Fauconnier, Rachid Guerraoui, and Eric Ruppert. When birds die : Making population protocols fault-tolerant. In Phillip B. Gibbons, Tarek F. Abdelzaher, James Aspnes, and Ramesh Rao, editors, *DCOSS*, volume 4026 of *Lecture Notes in Computer Science*, pages 51–66. Springer, 2006.

[DH99] Shlomi Dolev and Ted Herman. Parallel composition of stabilizing algorithms. In Arora [Aro99], pages 25–32.

[DH07] Shlomi Dolev and Ted Herman. Parallel composition for time-to-fault adaptive stabilization. *Distributed Computing*, 20(1) :29–38, 2007.

[Dij68] E. Dijkstra. *Cooperating Sequential Processes*. Academic Press, 1968.

[Dij74] Edsger W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Commun. ACM*, 17(11) :643–644, 1974.

[DK08] Shlomi Dolev and Ronen I. Kat. Hypertree for self-stabilizing peer-to-peer systems. *Distributed Computing*, 20(5) :375–388, 2008.

[DKS10] Shlomi Dolev, Ronen I. Kat, and Elad Michael Schiller. When consensus meets self-stabilization. *J. Comput. Syst. Sci.*, 76(8) :884–900, 2010.

[DMRT11] Sylvie Delaët, Partha Sarathi Mandal, Mariusz Rokicki, and Sébastien Tixeuil. Deterministic secure positioning in wireless sensor networks. *Theoretical Computer Science (TCS)*, 412(35) :4471–4481, August 2011.

[DMT12] Swan Dubois, Toshimitsu Masuzawa, and Sébastien Tixeuil. Bounding the impact of unbounded attacks in stabilization. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 23(3) :460–466, March 2012.

[Dol82] D. Dolev. The Byzantine generals strike again. *Journal of Algorithms*, 3(1) :14–30, 1982.

[Dol00] Shlomi. Dolev. *Self-stabilization*. MIT Press, March 2000.

[Dou02] John R. Douceur. The sybil attack. In Peter Druschel, M. Frans Kaashoek, and Antony I. T. Rowstron, editors, *IPTPS*, volume 2429 of *Lecture Notes in Computer Science*, pages 251–260. Springer, 2002.

[DP09] Yoann Dieudonné and Franck Petit. Scatter of robots. *Parallel Processing Letters*, 19(1) :175–184, 2009.

[DP12] Yoann Dieudonné and Franck Petit. Self-stabilizing gathering with strong multiplicity detection. *Theor. Comput. Sci.*, 428 :47–57, 2012.

[DPBT11] Swan Dubois, Maria Potop-Butucaru, and Sébastien Tixeuil. Dynamic ftss in asynchronous systems : the case of unison. *Theoretical Computer Science (TCS)*, 412(29) :3418–3439, July 2011.

[DPV11a] Stéphane Devismes, Franck Petit, and Vincent Villain. Autour de l'autostabilisation. 1. techniques généralisant l'approche. *Technique et Science Informatiques*, 30(7) :873–894, 2011.

[DPV11b] Stéphane Devismes, Franck Petit, and Vincent Villain. Autour de l'autostabilisation. 2. techniques spécialisant l'approche. *Technique et Science Informatiques*, 30(7) :895–922, 2011.

[DS03] Shlomi Dolev and Frank A. Stomp. Safety assurance via on-line monitoring. *Distributed Computing*, 16(4) :269–277, 2003.

[dSE99] A. de Saint-Exupéry. *Le Petit Prince*. Folio Junior. Editions Gallimard, 1999.

[DT98] Sylvie Delaët and Sébastien Tixeuil. Un algorithme auto-stabilisant en dépit de communications non fiables. *Technique et Science Informatiques (TSI)*, 5(17), May 1998.

[DT01] Bertrand Ducourthial and Sébastien Tixeuil. Self-stabilization with r-operators. *Distributed Computing (DC)*, 14(3) :147–162, July 2001.

[DT02] Sylvie Delaët and Sébastien Tixeuil. Tolerating transient and intermittent failures. *Journal of Parallel and Distributed Computing (JPDC)*, 62(5) :961–981, May 2002.

[DT03] Bertrand Ducourthial and Sébastien Tixeuil. Self-stabilization with path algebra. *Theoretical Computer Science (TCS)*, 293(1) :219–236, February 2003.

[DT11] Swan Dubois and Sébastien Tixeuil. A taxonomy of daemons in self-stabilization. Technical Report 1110.0334, ArXiv eprint, October 2011.

[DTY08] Stéphane Devismes, Sébastien Tixeuil, and Masafumi Yamashita. Weak vs. self vs. probabilistic stabilization. In *Proceedings of the IEEE International Conference on Distributed Computing Systems (ICDCS 2008)*, pages 681–688, Beijin, China, June 2008.

[Duc07] Bertrand Ducourthial. r-semi-groups : A generic approach for designing stabilizing silent tasks. In Masuzawa and Tixeuil [MT07a], pages 281–295.

[DW04] Shlomi Dolev and Jennifer L. Welch. Self-stabilizing clock synchronization in the presence of byzantine faults. *J. ACM*, 51(5) :780–799, 2004.

[FLP85] Michael J. Fischer, Nancy A. Lynch, and Mike Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2) :374–382, 1985.

[Fly72] M. Flynn. Some computer organizations and their effectiveness. *Computers, IEEE Transactions on*, C-21(9) :948–960, 1972.

[FPS12] Paola Flocchini, Giuseppe Prencipe, and Nicola Santoro. *Distributed Computing by Oblivious Mobile Robots*. Synthesis Lectures on Distributed Computing Theory. Morgan & Claypool Publishers, 2012.

[FSVY13] Paola Flocchini, Nicola Santoro, Giovanni Viglietta, and Masafumi Yamashita. Rendezvous of two robots with constant memory. *CoRR*, abs/1306.1956, 2013.

[GÖ3] Felix C. Gärtner. A survey of self-stabilizing spanning-tree construction algorithms. Technical report ic/2003/38, EPFL, 2003.

[GCH06] Mohamed G. Gouda, Jorge Arturo Cobb, and Chin-Tser Huang. Fault masking in tri-redundant systems. In Datta and Gradinariu [DG06], pages 304–313.

[GGH+95] Sukumar Ghosh, Arobinda Gupta, Mehmet Hakan, Karaata Sriram, and V. Pemmaraju. Self-stabilizing dynamic programming algorithms on trees. In *in Proceedings of the Second Workshop on Self-Stabilizing Systems*, pages 11–1, 1995.

[GH97] Sukumar Ghosh and Ted Herman, editors. *3rd Workshop on Self-stabilizing Systems, Santa Barbara, California, August, 1997, Proceedings*. Carleton University Press, 1997.

[Gou98] Mohamed G. Gouda. *Elements of network protocol design*. John Wiley & Sons, Inc., New York, NY, USA, 1998.

[GR09] Rachid Guerraoui and Eric Ruppert. Names trump malice : Tiny mobile agents can tolerate byzantine failures. In Susanne Albers, Alberto Marchetti-Spaccamela, Yossi Matias, Sotiris E. Nikoletseas, and Wolfgang Thomas, editors, *ICALP (2)*, volume 5556 of *Lecture Notes in Computer Science*, pages 484–495. Springer, 2009.

[Gri67] Pierre Gripari. *Contes de la rue Broca*. La Table Ronde, 1967.

[GSW02] Timothy Griffin, F. Bruce Shepherd, and Gordon T. Wilfong. The stable paths problem and interdomain routing. *IEEE/ACM Trans. Netw.*, 10(2) :232–243, 2002.

[Her90] Ted Herman. Probabilistic self-stabilization. *Information Processing Letters*, 35(2) :63–67, 1990.

[Her03] 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.

[HG05] 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.

[HH03a] Shing-Tsaan Huang and Ted Herman, editors. *Self-Stabilizing Systems, 6th International Symposium, SSS 2003, San Francisco, CA, USA, June 24-25, 2003, Proceedings*, volume 2704 of *Lecture Notes in Computer Science*. Springer, 2003.

[HH03b] Shing-Tsaan Huang and Su-Shen Hung. Self-stabilizing token circulation on uniform trees by using edge-tokens. In Huang and Herman [HH03a], pages 92–101.

[HM90] Joseph Y. Halpern and Yoram Moses. Knowledge and common knowledge in a distributed environment. *J. ACM*, 37(3) :549–587, 1990.

[HT04] 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.

[HT05] Ted Herman and Sébastien Tixeuil, editors. *Self-Stabilizing Systems, 7th International Symposium, SSS 2005, Barcelona, Spain, October 26-27, 2005, Proceedings*, volume 3764 of *Lecture Notes in Computer Science*. Springer, 2005.

[IR81] Alon Itai and Michael Rodeh. Symmetry breaking in distributive networks. In *FOCS*, pages 150–158. IEEE Computer Society, 1981.

[Kla02] C. Klapisch. *L'auberge espagnole*. 20th Century Fox, 2002.

[Kle98] N.H. Kleinbaum. *Le cercle des poètes disparus*. Succès du livre éditions, 1998.

[KP93] Shmuel Katz and Kenneth J. Perry. Self-stabilizing extensions for message-passing systems. *Distributed Computing*, 7(1) :17–26, 1993.

[KP99] Shay Kutten and David Peleg. Fault-local distributed mending. *J. Algorithms*, 30(1) :144–165, 1999.

[KY97] H. Kakugawa and M. Yamashita. Uniform and self-stabilizing token rings allowing unfair daemon. *ieeetpds*, 8(2) :154–162, 1997.

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7) :558–565, 1978.

[Lam84] Leslie Lamport. 1983 invited address solved problems, unsolved problems and non-problems in concurrency. In *Proceedings of the third annual ACM symposium on Principles of distributed computing*, PODC '84, pages 1–11, New York, NY, USA, 1984. ACM.

[Lam98] Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2) :133–169, 1998.

[Lan77] Gérard Le Lann. Distributed systems - towards a formal approach. In *IFIP Congress*, pages 155–160, 1977.

[LSP82] Leslie Lamport, Robert E. Shostak, and Marshall C. Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3) :382–401, 1982.

[Lyn80] N.A. Lynch. Fast allocation of nearby resources in a distributed system. In *Proceedings of the 12th Annual ACM Symposium on Theory of Computing, (STOC)*, pages 70–81. ACM Press, April 1980.

[Lyn96] Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1996.

[MH13] Hammurabi Mendes and Maurice Herlihy. Multidimensional approximate agreement in byzantine asynchronous systems. In Dan Boneh, Tim Roughgarden, and Joan Feigenbaum, editors, *STOC*, pages 391–400. ACM, 2013.

[MLG06] H.G. Myung, Junsung Lim, and D. Goodman. Single carrier fdma for uplink wireless transmission. *Vehicular Technology Magazine, IEEE*, 1(3) :30–38, 2006.

[MOKY12] Ryu Mizoguchi, Hirotaka Ono, Shuji Kijima, and Masafumi Yamashita. On space complexity of self-stabilizing leader election in mediated population protocol. *Distributed Computing*, 25(6) :451–460, 2012.

[Moy98] J. Moy. Ospf version 2, 1998.

[MT07a] Toshimitsu Masuzawa and Sébastien Tixeuil, editors. *Reliability, Availibility, and Security, 1st International Workshop, WRAS 2007, Paris, France, November 16, 2007, Proceedings*. Univ. Paris 6, 2007.

[MT07b] Toshimitsu Masuzawa and Sébastien Tixeuil. Stabilizing link-coloration of arbitrary networks with unbounded byzantine faults. *International Journal of Principles and Applications of Information Science and Technology (PAIST)*, 1(1) :1–13, December 2007.

[NA02] Mikhail Nesterenko and Anish Arora. Tolerance to unbounded byzantine faults. In *21st Symposium on Reliable Distributed Systems (SRDS 2002)*, pages 22–29. IEEE Computer Society, 2002.

[NT09] Mikhail Nesterenko and Sébastien Tixeuil. Discovering network topology in the presence of byzantine nodes. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 20(12) :1777–1789, December 2009.

[NT11] Mikhail Nesterenko and Sébastien Tixeuil. Ideal Stabilization. In *Proceedings of IEEE AINA 2011*, pages 224–231, Biopolis, Singapore, March 2011. IEEE Press.

[OT12] Fukuhito Ooshita and Sébastien Tixeuil. On the self-stabilization of mobile oblivious robots in uniform rings. In *Proceedings of the International Conference on Stabilization, Safety, and Security in Distributed Systems (SSS 2012)*, Lecture Notes in Computer Science (LNCS), Toronto, Canada, October 2012. Springer Berlin / Heidelberg.

[PBRD03] C. Perkins, E. Belding-Royer, and S. Das. Ad hoc on-demand distance vector (aodv) routing, 2003.

[Pel00] David Peleg. *Distributed computing : a locality-sensitive approach*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2000.

[Per00] R. Perlman. *Interconnexion Networks*. Addison-Wesley, 2000.

[Rab46] F. Rabelais. *Gargantua et Pantagruel*. Number vol. 2 in Gargantua et Pantagruel. Bibliothèque Larousse, 1546.

[Ray10a] Michel Raynal. *Communication and Agreement Abstractions for Fault-Tolerant Asynchronous Distributed Systems*. Synthesis Lectures on Distributed Computing Theory. Morgan & Claypool Publishers, 2010.

[Ray10b] Michel Raynal. *Fault-tolerant Agreement in Synchronous Message-passing Systems*. Synthesis Lectures on Distributed Computing Theory. Morgan & Claypool Publishers, 2010.

[Ray13a] M. Raynal. *Distributed Algorithms for Message-Passing Systems*. Springer-Verlag New York Incorporated, 2013.

[Ray13b] Michel Raynal. *Concurrent Programming - Algorithms, Principles, and Foundations*. Springer, 2013.

[SNY+09] Yuichi Sudo, Junya Nakamura, Yukiko Yamauchi, Fukuhito Ooshita, Hirotsugu Kakugawa, and Toshimitsu Masuzawa. Loosely-stabilizing leader election in population protocol model. In Shay

Kutten and Janez Zerovnik, editors, *SIROCCO*, volume 5869 of *Lecture Notes in Computer Science*, pages 295–308. Springer, 2009.

[SY99] Ichiro Suzuki and Masafumi Yamashita. Distributed anonymous mobile robots : Formation of geometric patterns. *SIAM J. Comput.*, 28(4) :1347–1363, 1999.

[Tel94] Gerard Tel. Maximal matching stabilizes in quadratic time. *Inf. Process. Lett.*, 49(6) :271–272, 1994.

[Tel00] G. Tel. *Introduction to distributed algorithms*. Cambridge University Press, 2000.

[Tel01] G Tel. *Introduction to distributed algorithms*. Cambridge University Press, Cambridge, UK, Second edition 2001.

[TG99] Oliver E. Theel and Felix C. Gärtner. An exercise in proving convergence through transfer functions. In Arora [Aro99], pages 41–47.

[Tix06] Sébastien Tixeuil. Vers l'auto-stabilisation des systèmes à grande échelle. Habilitation à Diriger les Recherches 1452, LRI Université Paris Sud, mai 2006.

[Tix09] Sébastien Tixeuil. *Algorithms and Theory of Computation Handbook, Second Edition*, chapter Self-stabilizing Algorithms, pages 26.1–26.45. Chapman & Hall/CRC Applied Algorithms and Data Structures. CRC Press, Taylor & Francis Group, November 2009.

[Var97] George Varghese. Compositional proofs of self-stabilizing protocols. In Ghosh and Herman [GH97], pages 80–94.

[VJ00] George Varghese and Mahesh Jayaram. The fault span of crash failures. *J. ACM*, 47(2) :244–293, 2000.

[VNTD08] Adnam Vora, Mikhail Nesterenko, Sébastien Tixeuil, and Sylvie Delaët. Universe detectors for sybil defense in ad hoc wireles networks. In *Proceedings of the International Conference on Stabilization, Safety, and Security in Distributed Systems (SSS 2008)*, Lecture Notes in Computer Science. Springer-Verlag, November 2008.

[YK96a] Masafumi Yamashita and Tsunehiko Kameda. Computing on anonymous networks : Part i-characterizing the solvable cases. *IEEE Trans. Parallel Distrib. Syst.*, 7(1) :69–89, 1996.

[YK96b] Masafumi Yamashita and Tsunehiko Kameda. Computing on anonymous networks : Part ii-decision and membership problems. *IEEE Trans. Parallel Distrib. Syst.*, 7(1) :90–96, 1996.

[YKO+06] Yukiko Yamauchi, Sayaka Kamei, Fukuhito Ooshita, Yoshiaki Katayama, Hirotsugu Kakugawa, and Toshimitsu Masuzawa. Composition of fault-containing protocols based on recovery waiting fault-containing composition framework. In Datta and Gradinariu [DG06], pages 516–532.

Index verborum

Proche de l'index rerum, l'index verborum n'indexe lui que les mots relevant du document sans les associer nécessairement à un thème précis.

Table des figures

Index nominum

Cet index reprend l'ensemble des noms cités dans la partie centrale de ce manuscrit (hors remerciement) .

Index rerum

L'index rerum ou index thématique renvoie au moment où chaque thème est abordé dans le manuscrit.

Table des matières

Quand tu arrives en haut de la montagne continue de grimper

Proverbe tibétain