République Algérienne Démocratique Et Populaire
Ministère De L'Enseignement Supérieur Et De La Recherche Scientifique
Université Abderahmane Mira De Béjaïa
Faculté Des Sciences Et Des Sciences De L'ingénieur
Département D'informatique
École Doctorale Réseaux Et Systèmes Distribués

Mémoire de Magistère
en Informatique

Option

Réseaux et Systèmes Distribués

Thème

Transposition Automatique des Algorithmes sur des Systèmes Distribués et Dynamiques

---

Présenté par

Salah GUESMIA

Soutenu le 12/07/2007

Devant le jury composé de :

Président	M^r Ahmed AÏT-SAIDI, Maître de conférence, Université A/Mira, Béjaïa, Algérie
Rapporteur	M^r Hamamache KHEDDOUCI, Professeur, Université CBL, Lyon, France
Examinateurs	M^{me} Saâdia TAS, Maître de conférence, Université A/Mira, Béjaïa, Algérie
	M^r Ali MELIT, Maître de conférence, Université de Jijel, Jijel, Algérie

Promotion 2005 – 2006

Dédicaces

À mes parents
À mes frères et soeurs
À toute ma famille
À tous mes chères collègues et amis

Remerciements

Je remercie le Dieu tout puissant qui m'a donné la force et la volonté pour réaliser ce modeste travail.

Mes vifs remerciements accompagnés de toute ma gratitude vont tout d'abord à mon encadreur M^r Hamamache KHEDDOUCI, professeur à l'université Claude Bernard (Lyon 1), pour m'avoir proposé ce sujet intéressant et pour ses conseils et orientations. Je le remercie surtout pour m'avoir fait confiance durant toute cette année.

Mes plus sincères remerciements s'adressent aux membres de jury constitué de : M^r Ahmed AÏT-SAIDI, Maître de conférence à l'université de Béjaia, M^{me} Saâdia TAS, Maître de conférence à l'université de Béjaia et M^r Ali MELIT, Maître de conférence à l'université de Jijel d'avoir accepté de juger ce travail.

Je tiens à remercier M^r Kamel TARI, le Chef de département d'informatique et le responsable de l'école doctorale, qui nous a toujours encouragé et soutenu. Je remercie également tous nos enseignants et tous ceux qui ont contribué à la création et la réussite de l'école doctorale ReSyD de Béjaia.

Je tiens aussi à remercier mes collègues Nabil Guellati, Nabil Nouri, Hamza, Fouzi et Rafik pour leur collaboration durant toute cette année. Je remercie mes amis et collègues de l'école doctorale ReSyD pour leur soutien et encouragements durant ces années d'études à Béjaia.

Enfin, je remercie tous ceux qui, de près ou de loin, ont contribué par leurs conseils, leurs encouragements ou leurs amitiés à l'aboutissement de ce travail.

Résumé

Le concept d'auto-stabilisation a été introduit en 1974 par E.G. Dijkstra. Un système réparti est défini comme auto-stabilisant s'il se comporte selon sa spécification au bout d'un temps fini quelque soit sa configuration initiale. Par cette propriété, les algorithmes auto-stabilisants tolèrent tout genre et tout nombre fini de défaillances transitoires. Puisque la conception et la preuve d'algorithmes auto-stabilisants sont généralement des tâches complexes, ainsi que l'existence d'un nombre important d'algorithmes répartis mais qui ne possèdent pas la propriété d'auto-stabilisation, certains modèles d'auto-stabilisation automatique (appelés aussi transformateurs) ont été proposés dans la littérature afin d'ajouter la propriété d'auto-stabilisation aux algorithmes répartis. Dans ce travail, nous avons donné une classification de ces modèles et étudié les modèles les plus intéressants. Nous avons proposé aussi un algorithme réparti auto-stabilisant pour le maintien de la topologie dans les systèmes dynamiques. Cet algorithme est utilisé pour concevoir un transformateur qui transpose une classe d'algorithmes séquentiels en algorithmes répartis auto-stabilisants et dynamiques.

Mots clés **:** algorithmes distribués, communication, systèmes distribués, systèmes dynamiques, réseaux.

Abstract

The concept of self-stabilization was introduced in 1974 by E.G. Dijkstra. A distributed system is defined to be self-stabilizing if regardless of the initial state, it behaves according to its specification after a finit time. By this property, the self-stabilizing algorithms tolerate any kind and any finished number of transitory failures. Since the design and the proof of self-stabilizing algorithms are generally a complex tasks, as well as the existence of an important number of distributed algorithms but which are not self-stabilizing, some models of automatic self-stabilization (called also transformers) were proposed in the literature in order to add the property of self-stabilization to the distributed algorithms. In this work, we gave a classification of these models and studied the most interesting models. We also proposed a distributed self-stabilizing algorithm for topology maintenance in dynamic systems. This algorithm is used to design a transformer which transposes a class of sequential algorithms in distributed self-stabilizing and dynamic algorithms.

Keywords **:** distributed algorithms, communication, distributed systems , dynamic systems, networks.

Table des matières

Liste des figures

Introduction générale

I regard this [Self-stabilization] as Dijkstra's most brilliant work at least, his most brilliant published paper. [...] I regard it to be a milestone in work on fault-tolerance.
Leslie Lamport.

LES systèmes répartis sont les systèmes qui gèrent plusieurs machines reliées entre eux par des moyens de communication. Le développement de ces systèmes a été principalement lié à plusieurs besoins, comme la communication entre entités géographiquement distantes, l'accélération des calculs et la fiabilisation des systèmes due à la redondance des moyens de calcul.

Lorsqu'on augmente le nombre de composants d'un système réparti, la possibilité qu'un (ou plusieurs) de ces composants tombe en panne augmente également. Lorsqu'on réduit le coût de fabrication des composants pour des raisons économiques, on accroît également le taux de défauts potentiels. En fin, lorsqu'on déploie les composants du système dans un environnement que l'on ne contrôle pas nécessairement, les risques de pannes deviennent impossibles à négliger. Ainsi, les algorithmes spécifiques à ces systèmes, les algorithmes répartis, se sont orientés depuis plusieurs années vers la prise en compte des défaillances potentielles des composants du système. Deux approches sont classiquement utilisées pour traiter ces défaillances :

- Les algorithmes robustes qui garantissent qu'un système continue à se comporter correctement en dépit des fautes.
- Les algorithmes auto-stabilisants qui assurent qu'après une perturbation temporaire, le système retrouve de lui-même (i.e. sans intervention extérieure) et en un temps fini un comportement correct.

La grande force des systèmes auto-stabilisants est le fait qu'ils puissent résister à tout type de défaillances transitoires. Alors que la seule restriction considérée dans cette ap-

proche est le fait que les défaillances ne corrompent pas les codes des algorithmes (cette restriction peut être contournée en imposant un rafraîchissement périodique des codes ou en stockant les algorithmes en mémoire morte).

La conception et la preuve d'algorithmes auto-stabilisants sont généralement des tâches complexes, ainsi que l'existence d'un nombre important d'algorithmes répartis mais qui ne possèdent pas la propriété d'auto-stabilisation ont fait que certains modèles d'auto-stabilisation automatique, appelés aussi transformateurs, ont été proposés dans la littérature, afin d'ajouter la propriété d'auto-stabilisation aux algorithmes répartis. Dans ce mémoire, nous étudions ces modèles d'auto-stabilisation et nous proposons un modèle pour la transposition d'une classe d'algorithmes séquentiels de graphe en algorithmes répartis auto-stabilisants et dynamiques.

Organisation du mémoire

Ce mémoire est organisé en quatre chapitres.

Le premier chapitre introduit les systèmes répartis, leurs principales caractéristiques ainsi que les problèmes classiques qui leur sont associés. Les concepts présentés dans ce chapitre sont utilisés tout au long du mémoire.

Dans le deuxième chapitre, nous nous intéressons aux différentes techniques de tolérance aux défaillances, en mettant un accent particulier sur l'auto-stabilisation et ses différentes variantes.

Dans le troisième chapitre, nous présentons les différents modèles d'auto-stabilisation automatique.

Le quatrième chapitre est consacré à notre proposition. Dans ce chapitre, nous présentons notre algorithme auto-stabilisant pour le maintien de la topologie, la preuve d'auto-stabilisation et comment utiliser cet algorithme pour transposer une classe d'algorithmes séquentiels de graphe en algorithmes répartis auto-stabilisants et dynamiques.

Enfin, le mémoire se termine par une conclusion et des perspectives.

CHAPITRE

1 Systèmes répartis

Le domaine des systèmes répartis (distribués) est vaste et assez récent. Ces systèmes répondent à la demande croissante de puissance de calcul et de communication. Dans ce chapitre, nous introduisons les systèmes répartis, leurs propriétés et les problèmes qui leur sont associés. Suite à cette introduction, nous présentons les hypothèses de modulation en décrivant plus précisément le fonctionnent de tels systèmes.

1.1 Généralités

Un système réparti est, selon [4], un ensemble d'entités autonomes (ordinateurs, processeurs, processus) reliées entre elles par un réseau de communication. Ces entités sont autonomes car elles sont capables d'effectuer indépendamment les unes des autres des calculs sur des mémoires locales. Elles peuvent également communiquer les unes avec les autres via le réseau. Les liens de communication sont des mémoires partagées ou des canaux dans lesquels les entités transitent des messages.

Les systèmes répartis existent sous plusieurs formes à priori dissemblables mais qui possèdent plusieurs caractéristiques en commun. Par exemple, un ordinateur exécutant un système d'exploitation multi-processus (comme Unix) peut être considéré comme un système réparti. De la même manière une machine parallèle constituée de plusieurs processeurs partageant une mémoire commune est également un système réparti [49]. Un troisième exemple est celui d'un réseau d'ordinateurs qui regroupe un grand nombre d'ordinateurs individuels, de caractéristiques différentes et exécutant des systèmes d'exploitation différents. Ces ordinateurs peuvent être liés entre eux par divers supports, et communiquant au moyen d'un langage commun : le protocole réseau.

De tels systèmes sont devenus très courants. Ils sont utilisés par un nombre croissant de personnes, et leur utilisation est parfois incontournable. Leur popularité s'explique par les nombreux avantages offerts :

- **Un accès distant facilité** : la possibilité d'accéder à une ressource sans forcément connaître sa localisation physique.
- **Une meilleure fiabilité et une disponibilité augmentée** : d'une part la nature du système permet d'assurer une certaine sûreté de fonctionnement (tolérance aux pannes) puisque les entités matérielles sont autonomes, et d'autre part, le principe de la redondance permet d'accroître la disponibilité des services.
- **Des performances accrues** : notamment grâce à la distribution des calculs et des données, mais aussi grâce à la redondance qui offre la possibilité de choisir le service le mieux adapté parmi des services équivalents.
- **Un passage à l'échelle possible** : Internet semble être la meilleure illustration de la grande capacité d'évolution des systèmes répartis puisque le nombre de machines le composant connaît une augmentation impressionnante (d'une centaine d'ordinateurs avant 1980 à plus de 56 millions en 1999).
- **Un coût financier modéré** : en effet, les petits ordinateurs ont un bien meilleur rapport prix/performances que les gros. De plus le coût technique de mise en place est aussi limité.

Cependant, bien que ces systèmes distribués soient pourvus de nombreux atouts, ils n'en reste pas moins que leur construction et leur utilisation sont sources de nombreuses difficultés.

1.2 Spécificités des systèmes répartis

Pour réaliser les différentes fonctions demandées à un système réparti, on a développé des algorithmes spécifiques adaptés au contexte réparti : les algorithmes répartis. Les différences entre les systèmes répartis et les systèmes centralisés tiennent en trois points essentiels [49] :

1. **Localité des Informations** : Dans un système centralisé, l'état global du système peut être connu à tout instant par le programme en cours d'exécution : cet état global est en général déterminé seulement par l'état des variables et par le compteur de programme. Dans un système réparti, chaque programme en cours d'exécution

(ou processus) ne possède qu'une connaissance locale donc partielle de l'état du système. D'autre part, l'état du système de communication ne peut jamais être observé directement par les processus.

2. **Localité du temps** : Dans un système centralisé, l'unique processeur exécute les actions de son programme séquentiellement à une vitesse donnée. Il lui est facile de déterminer combien d'actions ont été nécessaires à l'établissement d'une tâche, combien de temps il a fallu pour que le système donne un résultat correct. Dans un système réparti, les événements ne sont plus totalement ordonnés mais organisés selon une relation d'ordre partiel.

3. **Non déterminisme** : Du fait de l'hétérogénéité possible des processeurs et des liens de communication dans un système réparti, il est tout à fait possible que ces différents composants agissent à des vitesses différentes, et par conséquent induisent des comportements non déterministes.

1.3 Eléments de base

Un système réparti est composé de deux éléments de base : les liens et les processus. Dans cette section nous présentons une description détaillée des liens de communication en expliquant les différents modèles de communication. Nous donnons aussi une description précise des processus et nous introduisons la notion d'algorithme.

1.3.1 Liens

Les liens représentent les éléments de communication du système, ils sont utilisés par les processus pour échanger des informations. On distingue deux types de liens :

1. Une zone mémoire qui correspond à des variables partagées par des processus. Elle est utilisée dans le modèle de communication par mémoire partagée.

2. Un canal qui correspond à un lien entre deux processus pouvant contenir des messages échangés par ces processus. Il est utilisé dans le modèle de communication par échange de messages.

1.3.1.1 Communication par mémoire partagée

Dans le modèle de mémoire partagée, les processus disposent d'une zone mémoire commune dans laquelle ils peuvent lire pour récupérer des informations et écrire pour passer des informations. La figure 1.1 présente deux variantes de ce modèle de communication :

1. **Le modèle à états** : Dans lequel chaque processus met à jour son propre état et accède directement aux états de ses voisins par des opérations de lecture.

2. **Le modèle à registres** : Dans lequel la communication entre deux processus voisins se fait via deux registres (un registre pour chacun) partagés exclusivement entre ces deux processus. Chaque processus peut lire et écrire dans ses propres registres, mais il ne peut que lire les registres que ses voisins partagent avec lui.

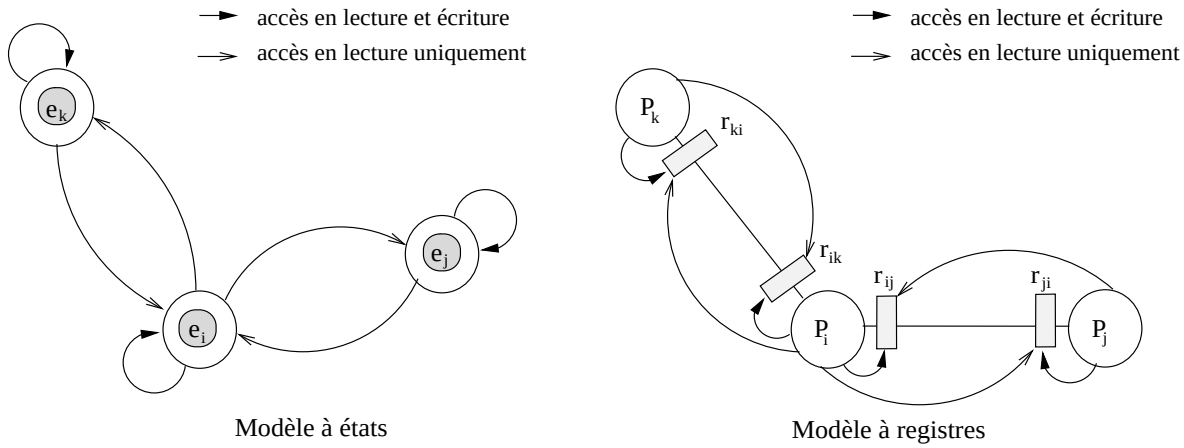


FIG. 1.1 – Modèles de communication par mémoires partagées.

1.3.1.2 Communication par échange de messages

Dans ce modèle de communication, les processus communiquent grâce à des canaux. Lorsqu'un processus veut transmettre une information à un autre processus, il envoie un message via le canal les reliant.

La transformation du modèle de mémoire partagée en un modèle d'échange de messages est possible. A ce titre, de nombreuses études proposent des méthodes pour adapter à un autre modèle des algorithmes conçus pour un modèle particulier [20].

1.3.2 Processus

Les processus représentent les éléments de calcul du système. Ils sont décrits par des algorithmes. Lorsqu'on écrit un algorithme, on utilise fréquemment des variables nommées qui décrivent l'état d'un processus. Ce dernier peut effectuer des actions qui peuvent être divisées en trois ensembles disjoints :

1. I : L'ensemble des actions internes du processus. Ces actions correspondent à la lecture ou à l'écriture d'une variable locale au processus.

2. L : L'ensemble des actions de lecture du processus. Ces actions correspondent à la lecture d'un registre dans le cas d'un système à mémoire partagée, ou à la réception d'un message dans le cas d'un système à échange de messages.

3. E : L'ensemble des actions d'écriture du processus. Ces actions correspondent à l'écriture dans un registre dans le cas d'un système à mémoire partagée, ou à l'émission d'un message dans le cas d'un système à échange de messages.

De façon générale, un algorithme est décrit sous forme d'un ensemble de règles gardées. Une règle gardée d'un processus p est une règle de la forme garde $\rightarrow$ action où la partie gauche est appelée la garde et la partie droite l'action. La garde est une expression booléenne « prédicat »sur l'état du processus p et sur ceux de ses voisins, l'action est une affectation des variables de p. Si la garde est vraie alors on peut effectuer l'action correspondante. Chaque règle est identifiée par une étiquette. Un algorithme est *probabiliste* si certaines de ses règles gardées s'appuient sur un tirage aléatoire.

Dans le modèle à état, pour évaluer la garde d'une règle gardée, un processus p doit lire le contenu de toutes les variables apparaissant dans la garde. Une règle dont la garde est vide est dite *spontanée* [28]. Une règle *activable* est une règle dont la garde est valide, un *processus activable* est un processus dont l'une des ses règles est activable.

1.4 Classification des systèmes répartis

Du fait de la grande diversité des systèmes répartis, des algorithmes répartis spécifiques ont été développés en suivant différents critères [43].

1.4.1 Caractéristiques des processus

Selon les caractéristiques des processus qui composent un système réparti, on peut le répertorier dans l'une des catégories suivantes [43] :

- Un système réparti est *anonyme* si tous les processus sont strictement identiques : c'est à dire qu'ils exécutent tous le même code et qu'ils ont les mêmes propriétés. A l'opposé, un système réparti peut être *non anonyme* quand chaque processus dispose d'un unique identifiant.
- Un système réparti est *uniforme* si tous les processus exécutent le même code mais qu'ils sont capables de se reconnaître. L'uniformité est une propriété plus faible que l'anonymat. Un système est *non uniforme* si au moins un processus exécute un code différent des autres.
- Un système réparti peut être *déterministe* ou *non déterministe*. Dans le premier cas, tous les processus du système exécutent un algorithme local déterministe, tandis que dans le deuxième cas, il existe au moins un processus dont le fonctionnement est non déterministe (probabiliste).

La plupart des systèmes répartis actuels sont non anonymes et non uniformes, plus précisément chaque processus du système dispose d'un unique identifiant, et dans le système il existe au moins un processus dont l'algorithme local est différent de tous les autres. Pour communiquer, les processus utilisent leurs identifiants dans les entêtes des messages échangés.

Quelques auteurs ont essayé de trouver des systèmes de transformation pour passer automatiquement d'un système non uniforme à un système uniforme. Malheureusement, il existe de nombreux résultats d'impossibilité [2].

1.4.2 Caractéristiques des liens de communication

Les processus communiquent via un réseau de communication. Celui-ci est constitué de liens qui sont des registres partagés ou des canaux de communication permettant de faire transiter des messages. Selon la manière dont la communication est effectuée on distingue trois types de systèmes [43] :

1. Système réparti avec une communication *globale* : Dans un tel système, les processus sont connectés via un médium qui permet à tout processus de communiquer avec tous

les autres. C'est l'équivalent du bus des machines parallèles.

2. Système réparti avec une communication *multipoint* : Un lien de communication permet à un sous-ensemble des processus de communiquer entre eux.

3. Système réparti avec une communication *point à point* : Le lien de communication dans un tel système relie exactement deux processus. Le réseau peut être plus ou moins connecté.

Dans le cadre d'une communication point à point, on peut distinguer deux types de communication :

1. Communication *bidirectionnelle* quand le lien permet l'échange d'informations dans les deux sens.

2. Communication *unidirectionnelle* lorsque l'échange d'informations a un sens unique.

La communication entre les processus d'un système réparti peut être représentée par un graphe appelé *graphe de communication*. Ce graphe est orienté si les liens sont unidirectionnels, non orienté sinon.

1.4.3 Caractéristiques liées aux vitesses relatives

Dans un système réparti, les processus doivent communiquer entre eux pour collaborer à résoudre la tâche attribuée au système. Les hypothèses sur les vitesses relatives des processus et des liens de communications induisent une troisième classification des systèmes répartis :

1. Un système réparti est dit *synchrone* quand les processus exécutent leur algorithme local à la même vitesse et que les liens véhiculent l'information à la même vitesse. C'est-à-dire qu'on peut borner les temps d'exécution et de transmission.

2. Un système réparti est dit *asynchrone* lorsque les vitesses relatives des processus ne peuvent être bornées, ou que le délai d'acheminement des messages est fini mais non borné.

Cette classification peut être raffinée en étudiant la borne sur les vitesses relatives des composants [44]. On parle de système partiellement synchrone lorsque le synchronisme n'est pas parfait, mais que certaines bornes existent, qu'elles soient connues ou non.

1.5 Topologie

La notion de voisinage est essentielle dans un système réparti, elle permet d'établir avec quelles autres entités un processus peut communiquer. Pour l'obtenir, on introduit la notion de topologie qui indique la manière d'interconnexion entre les processus et les liens. Classiquement, on représente la topologie d'un système réparti par un graphe (le graphe de communication) où les processus sont les nœuds du graphe et les liens de communication entre les processus sont les arêtes (ou les arcs si les liens sont unidirectionnels).

Le choix d'une topologie dépend du problème traité et de l'avantage que représentent ses caractéristiques. La figure 1.2 montre quelques topologies non-orientées courantes [47] :

- *Topologie en anneau* : un anneau à n sommets est un graphe où il existe une numérotation des sommets de s_0 à s_{n-1} telle que les arêtes sont du type (s_i, s_{i+1}) (les indices sont pris modulo n).
- *Topologie en étoile* : un réseau est en étoile s'il possède un nœud central, et $n-1$ arêtes connectant les $n-1$ autres nœuds au centre.
- *Topologie en chaîne* : une topologie dont les nœuds et les arêtes forment un chemin simple.
- *Topologie en arbre* : un arbre à n nœuds est un graphe connexe à $n-1$ arêtes. Cette définition implique qu'un arbre ne contient pas de cycle.
- *Topologie maillée* : une topologie dont le graphe est quelconque.

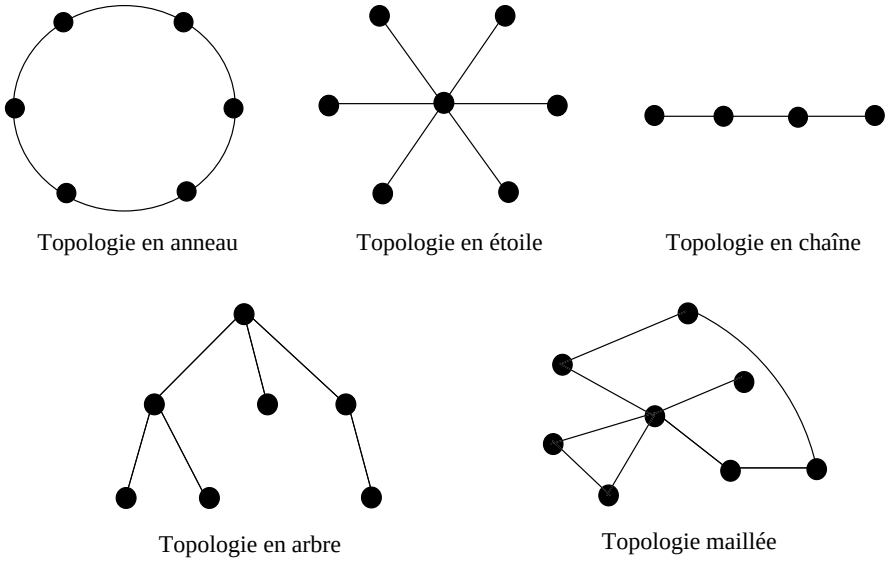


FIG. 1.2 – Exemples de topologies non-orientées.

De manière similaire aux topologies usuelles bidirectionnelles, il existe des topologies de référence lorsque les liens de communication sont unidirectionnels [49]. Parmi celle-ci, nous pouvons citer (la figure 1.3) :

- *Anneau unidirectionnel* : chaque processus possède un unique ascendant et un unique descendant.
- *Le graphe eulérien* : chaque processus possède autant d'ascendants que de descendants.
- *Le graphe fortement connexe* : chaque processus peut transmettre au moins indirectement des informations à chaque processus du réseau.
- *Le graphe avec source* : au moins un processus peut transmettre directement ou indirectement des informations à chaque processus du réseau.

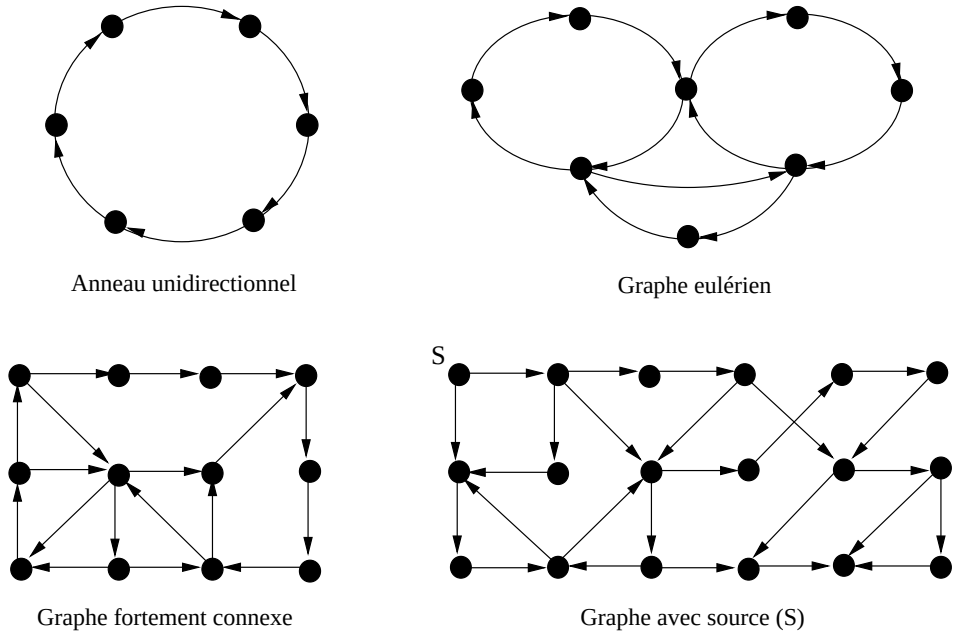


FIG. 1.3 – Exemples de topologies orientées.

De nombreux algorithmes répartis sont conçus pour fonctionner avec des topologies bien particulières, c'est principalement parce que ces topologies facilitent le contrôle de l'exécution du programme. Un problème se pose alors lorsque les systèmes réels sur lesquels sont implantés ces algorithmes répartis ne possèdent pas la topologie requise. Il est alors nécessaire de faire appel à un autre algorithme réparti qui construit virtuellement la topologie attendue par l'algorithme de plus haut niveau. Ainsi, de nombreux algorithmes répartis existent pour la construction de topologie particulière sur des graphes quelconques.

1.6 Systèmes répartis dynamiques

Dans le contexte des systèmes répartis, les transferts de données ne sont pas garantis à 100%. Certains liens de communication peuvent être, temporairement ou définitivement, perdus ou surchargés. On parle alors de systèmes répartis dynamiques. Afin de modéliser ce type de systèmes, différents modèles en théorie des graphes ont été proposés, parmi lesquels on trouve les graphes dynamiques. Intuitivement, un graphe dynamique est un graphe évoluant au cours du temps : des nœuds et des arêtes peuvent être supprimés ou rajoutés [53]. Habituellement, cette évolution est *continue* (seul un certain nombre de sommets, mais pas tous les sommets sont impliqués en même temps).

1.7 Problèmes classiques en algorithmique répartie

A travers la littérature, parmi les problèmes traités, on trouve un petit nombre de problèmes qui reviennent régulièrement. Dans cette section, nous dressons une liste non exhaustive de ces problèmes.

1.7.1 Problèmes statiques

Les problèmes statiques (*non-réactifs*) sont ceux dont la spécification ne dépend que du graphe de communication du système [28]. C'est généralement le cas de l'élection d'un leader, le consensus ou encore la diffusion d'information. La particularité de ces problèmes est qu'à partir d'un certain point de l'exécution, l'algorithme n'évolue plus.

- **Election** : dans certaines applications réparties, il arrive qu'on ait besoin de distinguer un processus des autres, appelé *leader*. Ainsi, le problème de l'élection consiste à élire, parmi tous les processus du système, un et un seul processus dont tous les autres ont connaissance.
- **Consensus** : le problème de consensus est un problème de base auquel se ramènent la plupart des problèmes d'accord. L'objectif du consensus est de permettre à un ensemble de processus, chacun possédant sa valeur initiale, de décider de manière irrévocable sur l'une des valeurs initiales. Plus formellement, le consensus est spécifié par les propriétés suivantes :
  - *Agrément* : deux processus ne peuvent pas décider différemment ;

- *Terminaison* : tout processus doit finalement décider ;
- *Validité* : la valeur décidée est une des valeurs proposées.

– **Diffusion d'information** : quand un système réparti utilise des communications point à point, il est toutefois nécessaire de disposer de primitives de communications globales : la *diffusion* (un processus transmet des informations à tous les autres), l'*échange total* (tous les processus transmettent des informations à tous les autres). Ces primitives rendent possible l'exploitation d'algorithmes prévus pour les systèmes à communications globales dans les systèmes à communication point à point ou multipoints.

1.7.2 Problèmes dynamiques

A l'inverse des problèmes statiques, les problèmes dynamiques (*réactifs*) ne dépendent pas uniquement de l'état du réseau mais aussi de son évolution dans le temps [28]. En effet, les spécifications de ces problèmes portent sur l'exécution du système. C'est notamment le cas de l'exclusion mutuelle ou encore la synchronisation. La particularité de ces problèmes est que toutes les exécutions du système sont infinies.

– **Exclusion mutuelle** : les processus du système ont accès à une ressource partagée mais cette dernière ne peut être accédée que par un processus à la fois. Un algorithme d'exclusion mutuelle garantit que l'accès à cette ressource est sérialisé et équitable :

- *Sûreté* : à chaque instant, un processus au plus accède à la ressource ;
- *Vivacité* : le temps d'attente d'un processus pour accéder à la ressource est fini.

– **Synchronisation** : un algorithme réparti fonctionnant dans un système synchrone est plus facile à concevoir, puisque le non déterminisme du comportement peut être partiellement supprimé, mais pour qu'il fonctionne dans un système asynchrone, il faut utiliser en parallèle un algorithme réparti spécial appelé *synchroniseur*, qui simule des processus synchrones dans un système asynchrone. Généralement, cette transformation s'accompagne d'une perte de performances significative.

1.8 Configurations et transitions

Un système réparti peut être modélisé par un système de transition qui fait passer le système d'une configuration à une autre via une fonction de transition.

L'état global d'un système réparti à un instant précis est entièrement décrit par une configuration. Cette dernière peut être caractérisé par un vecteur d'états de chaque processus et de chaque lien du système. L'ensemble des configurations est noté $\mathcal{C}$.

Une transition d'un système réparti est l'exécution d'un ou plusieurs pas atomiques par un sous-ensemble non vide des processus du système [30]. Elle ne dépend que d'une partie de la configuration, et n'influence qu'une partie de cette configuration (l'état local d'un processus est déduit de son état local et/ou de l'état de ses voisins). C'est ce qui induit la notion de distribution ou de parallélisme du système. L'ensemble des transitions est noté $\mathcal{T}$.

Définition 1 *Un* ***système réparti*** *S est une paire $(\mathcal{C}, \mathcal{T})$ où $\mathcal{C}$ est un ensemble de configurations et $\mathcal{T}$ un ensemble de transitons.*

Afin de dissocier le système de transition d'un processus de celui du système réparti, nous utiliserons les termes de configuration et de transition pour désigner le système réparti et les termes d'état et de pas atomique pour décrire le comportement d'un processus.

1.9 Propriétés des exécutions

Définition 2 *Une* ***exécution*** *d'un système réparti, notée $e = c_0, \tau_0, c_1, ...$, est une suite alternée de configurations et de transitions où c_i est obtenue en appliquant la transition τ_{i-1} à la configuration c_{i-1}. La configuration c_0 est appelée configuration initiale de e.*

Une exécution est soit infinie, soit finie mais aucune transition du système n'est possible à partir de la dernière configuration appelée *configuration terminale* [49]. L'ensemble des exécutions d'un système S est noté $\mathcal{E}$.

Il est fréquent que plusieurs actions soient exécutables à partir d'une même configuration d'un système réparti. Par conséquent, l'exécution d'une action résulte du choix de cette action parmi toutes celles qui sont exécutables en cette configuration et du choix de leur niveau d'atomicité. On va donc pouvoir générer plusieurs exécutions (souvent une infinité) à partir d'une seule configuration initiale du système. Ces exécutions résultent des différents choix faits en chaque configuration rencontrée. Dans la suite de cette section, nous étudierons certaines propriétés remarquables d'une exécution.

1.9.1 Atomicité

Chaque exécution d'un système réparti est vue comme une suite de *pas atomiques* (*action atomique*). Un pas atomique peut être considéré comme la plus petite action que le système peut exécuter sans être interrompu [19]. On peut considérer différents niveaux d'actions comme étant atomiques. Par exemple, on peut considérer que toutes les actions d'un processus sont atomiques, qu'elles soient internes ou de communication. C'est le niveau d'atomicité le plus fin, mais ce niveau d'atomicité rend les preuves très difficiles. Toutefois, on peut également choisir des pas atomiques plus gros en regroupant certaines actions entre elles. Par exemple, on peut considérer que la partie droite des règles (la garde) est atomique, quel que soit le nombre d'actions qu'elle comporte.

Dans la littérature, deux niveaux d'atomicités ont été définis : l'atomicité *composite* et l'atomicité *lecture/écriture*. Les définitions données ici sont celles de [19] :

- **L'atomicité composite :** on considère qu'un pas atomique est composé de plusieurs actions d'entrée (lecture de registre ou réception de message), de plusieurs actions internes et d'une action de sortie (écriture de registre ou émission de message).
- **L'atomicité lecture/écriture :** on considère qu'un pas atomique est composé de plusieurs actions internes et une seule action d'entrée ou de sortie.

1.9.2 Équité

Nous avons vu qu'à chaque configuration, un choix doit être fait pour déterminer quelle action va être exécutée parmi toutes celles qui sont exécutables. Considérons par exemple deux processus qui s'exécutent en parallèle tels qu'ils sont activables en chaque configuration. Il existe différentes exécutions relatives à ce système. Une exécution possible consiste à choisir en chaque état du système d'exécuter l'action du premier processus. Dans cette exécution, le deuxième processus n'exécute jamais une action. Cette exécution est typique d'un comportement du système avec famine. La propriété d'équité sur les exécutions infinies garantit qu'une action ou un processus ne soit pas défavorisé par rapport aux autres.

La propriété d'équité ne porte pas seulement sur l'exécution des actions des processus, mais aussi sur les communications par échange de messages. En effet, si les canaux ne sont pas ordonnés, il se peut qu'un message envoyé ne soit jamais reçu lors de certaines exécutions. On dit que les communications d'une exécution e sont équitables si et seulement si pour toute action d'émission d'un message dans e, il existe l'action de réception

correspondante dans e [44].

1.9.3 Démon

Un démon modélise une propriété d'exécutions des systèmes répartis. Cette propriété concerne l'ordre des actions apparaissant dans ces exécutions. Il sélectionne, pour chaque étape d'une exécution, un sous-ensemble non vide de processus activables qui vont effectuer une action. Le choix de ce sous-ensemble peut se faire de manière déterministe ou probabiliste selon le démon considéré. Dans la littérature, nous trouvons un grand nombre de démons [49, 44, 48]. Nous ne citerons que les démons utilisés dans les systèmes à mémoire partagée :

- Le démon ***lecture/écriture*** n'impose pas d'ordre particulier sur les actions. Ainsi, l'ensemble des exécutions d'un système réparti S sous le démon lecture/écriture est l'ensemble de toutes les exécutions de S.
- Le démon ***totalement réparti*** impose que l'exécution d'une garde d'une règle gardée ne soit jamais interrompue et de même, que l'exécution d'une action d'une règle gardée ne soit jamais interrompue. C'est-à-dire, ce démon considère que les gardes sont évaluées atomiquement, et que les actions associées sont exécutées atomiquement.
- Le démon ***réparti*** est un démon totalement réparti tel que toute exécution peut être organisée en une séquence de phase d'évaluation des gardes puis de phase d'exécution des actions associées à ces gardes. Formellement, le démon choisit un sous-ensemble de processus parmi ceux qui peuvent agir. Tous les processus de cet ensemble doivent simultanément et en seule action atomique lire les variables de leurs voisins et ensuite écrire dans leurs propres variables.
- Le démon ***synchrone*** est un démon réparti tel que toute séquence de phases implique tous les processus du système. Formellement, les processus qui peuvent agir doivent simultanément et en seule action atomique lire les variables de leurs voisins et ensuite écrire dans leurs propres variables.
- Le démon ***centralisé*** est un démon réparti tel que toute séquence de phases implique un unique processus. Autrement dit, le démon centralisé impose que l'exécution d'une règle gardée ne soit jamais interrompue.

Il est possible d'ordonner les démons selon la contrainte qu'ils imposent sur les exécutions. En effet, un démon contraint de manière plus ou moins forte les exécutions d'un système. Si $\mathcal{E}$ est l'ensemble des exécutions possibles d'un système réparti, alors l'ensemble

des exécutions possibles du système sous un démon est un sous-ensemble de $\mathcal{E}$. On peut classifier ces sous-ensembles par simple inclusion. Si, pour tout système réparti S, l'ensemble des exécutions de S sous un démon d_1 est strictement inclus dans l'ensemble des exécutions de S sous un démon d_2, alors on dit que d_2 est plus puissant que d_1. Il est démontré dans [49] que la hiérarchie présentée dans la figure 1.4 est valide.

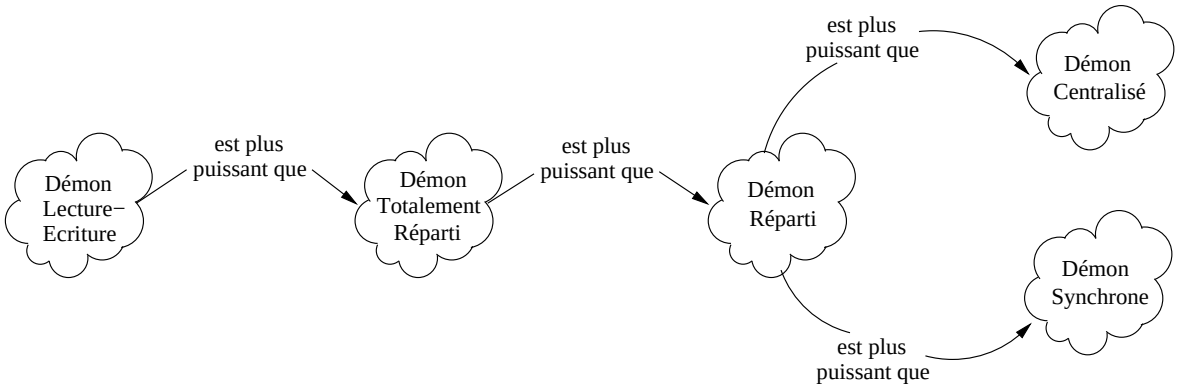


FIG. 1.4 – Hiérarchie des démons [49].

Il en résulte que si toutes les exécutions d'un système réparti S sous un démon d_1 vérifient une propriété *Prop*, et si d_1 est plus puissant que d_2, alors toutes les exécutions de S sous le démon d_2 vérifient *Prop*.

1.9.4 Spécification

Le plus souvent, une spécification pour une tâche donnée est décrite au moyen de prédicat sur les exécutions des systèmes répartis qui peuvent résoudre cette tâche [27]. Comme une spécification pour une tâche donnée est indépendante du système qui résout cette tâche, plusieurs systèmes distincts peuvent résoudre une même tâche T qui est spécifiée par une seule spécification $\mathcal{P}$. On dit aussi que ces systèmes satisfont la spécification $\mathcal{P}$ de la tache T.

DÉFINITION 3 *Une **spécification** $\mathcal{P}$ pour une tâche donnée T est un prédicat portant sur les exécutions des système répartis qui peuvent résoudre T.*

Notons qu'une spécification décrit des exécutions qui vérifient à tout instant une contrainte. Ainsi, un système vérifie une spécification si : après un temps fini correspondant au temps

de calcul nécessaire au système pour résoudre la tâche, les exécutions du système vérifient la spécification.

1.10 Conclusion

Dans ce chapitre, nous avons présenté des notions générales sur les systèmes répartis ainsi que les principales caractéristiques selon lesquelles les systèmes répartis peuvent être classifiés. Ces caractéristiques concernent les propriétés physiques du système, les liens de communication et les hypothèses sur la synchronisation des processus. Par la suite, plusieurs propriétés sur les exécutions des systèmes répartis ont été explicitées.

CHAPITRE

2 Tolérance aux défaillances

L'augmentation du nombre des éléments constitutifs d'un système réparti contribue à l'augmentation de la probabilité que l'un de ces éléments tombe en panne. L'intervention d'un opérateur pour corriger les défaillances qui deviennent alors fréquentes est donc impossible. Dans le cadre de l'algorithmique répartie, le problème de tolérer des défaillances tout en maintenant un comportement globalement cohérent a mené à l'étude d'algorithmes tolérants aux défaillances.

2.1 Notions de base : fautes, erreurs et défaillances

Une défaillance du système survient lorsque le service délivré diverge de l'accomplissement de la fonction du système, c'est-à-dire de ce à quoi le système est destiné. Une erreur est la partie de la configuration du système qui est susceptible d'entraîner une défaillance, c'est-à-dire qu'une défaillance se produit lorsque l'erreur atteint l'interface du service fourni et le modifie. Une faute ou panne est la cause adjugée ou supposée d'une erreur. On voit donc qu'il existe une chaîne causale entre faute, erreur et défaillance [45].

Il est à noter que dans les systèmes formés par l'interaction de plusieurs sous-systèmes, les notions de faute, d'erreur et de défaillance sont récursives : la défaillance d'un sous-système devient une faute pour le système le plus global (la figure 2.1).

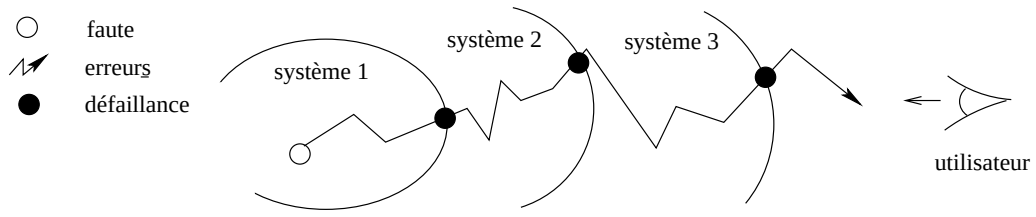


FIG. 2.1 – Faute, erreur et défaillance [45].

2.2 Taxonomie des défaillances dans les systèmes répartis

Plusieurs types de défaillances peuvent être répertoriés suivant la localisation des défaillances dans le temps et leurs nature [50].

2.2.1 Localisation des défaillances dans le temps

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

1. Les défaillances ***transitoires*** : des défaillances de nature arbitraire peuvent venir frapper le système, mais elles sont supposées se produire très rarement. Pour un processus, de telles défaillances peuvent se produire suite à une panne immédiatement suivie d'une réparation, ou encore lors de coupures temporaires de l'alimentation électrique. Pour un lien de communication, des débranchements et branchements successifs, ainsi que des perturbations électromagnétiques ponctuelles, peuvent conduire à des problèmes similaires.

2. Les défaillances ***définitives*** : des défaillances de nature arbitraire peuvent venir frapper le système, mais elles se prolongent indéfiniment dans le temps. Dans le cas d'un processus, cela signifie que celui-ci stoppe définitivement son exécution. Dans le cas d'un lien de communication, ça peut être traduit par une coupure définitive de ce lien.

3. Les défaillances ***intermittentes*** : des défaillances de nature arbitraire peuvent venir frapper le système à tout moment de l'exécution. Elles sont plus difficiles à pallier.

Bien sûr, les défaillances transitoires et les défaillances définitives sont deux cas particuliers des défaillances intermittentes. Cependant, dans un système où les défaillances intermittentes apparaissent rarement, un système qui tolère des défaillances transitoires peut être utile, car le taux de la vie utile du système (dans laquelle le système se comporte correctement) peut rester suffisamment élevé.

2.2.2 Nature des défaillances

Un deuxième critère est la nature des défaillances. Pour chaque élément d'un système réparti, ses états représentent les valeurs possibles des variables de cet élément, et les transitions représentent le code exécuté par l'élément. On peut alors distinguer les défaillances suivant qui surviennent sur l'état ou sur le code d'un élément du système :

2.2.2.1 Défaillances d'état

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

2.2.2.2 Défaillances de code

Le changement arbitraire du code d'un élément résulte la plupart du temps d'une attaque (par exemple le remplacement d'un élément par un adversaire malicieux), mais certains types moins graves peuvent être, par exemple, liés à des erreurs de programmation ou à des défaillances matérielles. On distingue donc plusieurs sous-catégories de défaillances de code :

1. Les défaillances ***crash*** : à un point donné de l'exécution, un élément arrête définitivement son exécution.

2. Les ***omissions*** : à divers instants de l'exécution, un élément peut omettre de communiquer avec les autres éléments du système, soit en émission, soit en réception.

3. Les ***duplications*** : à divers instants de l'exécution, un élément peut effectuer une action plusieurs fois, quand bien même son code précise que cette action doit être exécutée une fois.

4. Les ***déséquencements*** : à divers instants de l'exécution, un élément peut effectuer des actions correctes, mais dans le désordre.

5. Les défaillances ***byzantines*** : elles correspondent simplement à un type arbitraire de défaillances. Elles sont donc les défaillances les plus malicieuses possibles.

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

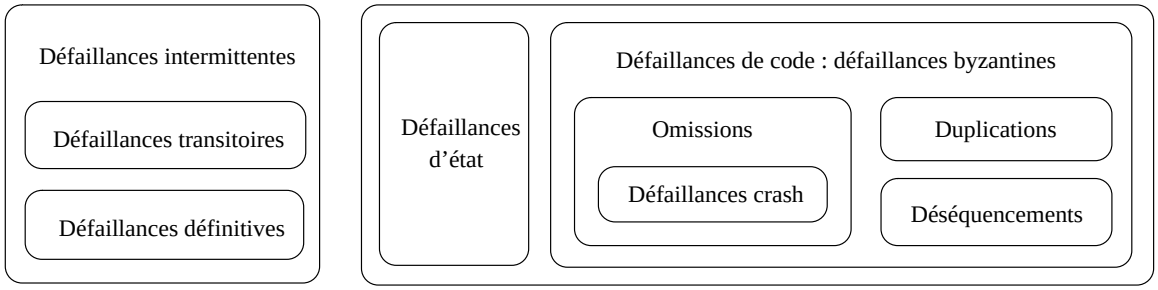


FIG. 2.2 – Taxonomie des défaillances dans les systèmes répartis.

La figure 2.2 résume les relations d'inclusions pouvant être induites entre les différents types de défaillances.

2.3 Classes d'algorithmes tolérants aux défaillances

En fonction du type de défaillances que l'on considère, du type de tolérance que nous voulons mettre en application et du nombre de défaillances que nous voulons tolérer, les systèmes ont ou non la possibilité de tolérer ces défaillances. Les techniques mises en oeuvre pour les tolérer sont alors différentes. Ces solutions peuvent être classifiées suivant la

visibilité de l'effet des défaillances à un observateur du système (par exemple un utilisateur). Une solution *masquante* cache à l'observateur l'occurrence des fautes (si celles-ci restent dans la limite tolérée par le système), alors qu'une solution *non masquante* ne présente pas cette propriété : l'effet des défaillances est visible pendant un temps plus ou moins long, puis le système recommence à se comporter correctement.

Une approche masquante apparaît a priori préférable, puisqu'elle s'applique à un plus grand nombre d'applications (et en particulier les applications critiques de sécurité). Cependant, une solution masquante est en général plus coûteuse (en ressources, en temps) qu'une solution non masquante, et ne peut tolérer des défaillances que dans la mesure où celles-ci ont été prévues.

Deux classes principales d'algorithmes tolérants aux défaillances peuvent être distinguées : *les algorithmes robustes* et *les algorithmes auto-stabilisants.*

2.3.1 Les algorithmes robustes

Ces algorithmes sont basés sur une vision pessimiste : on considère qu'il est nécessaire de vérifier constamment le comportement du système pour détecter la présence des défaillances et y remédier. De tels algorithmes sont typiquement masquants. Ils utilisent fréquemment la redondance à plusieurs niveaux des informations, des communications, ou des nœuds du système. Ils font en général l'hypothèse qu'un nombre limité de défaillances peut frapper le système, de manière à conserver au moins une majorité des éléments corrects (parfois plus, si les défaillances sont plus graves) : face à des processus byzantins, il a été démontré que même avec des systèmes de communication très fiables, il faut considérer que moins d'un tiers des processus subit des défaillances pour pouvoir atteindre un agrément [41]. En outre, Ficher, Lynch et Paterson [21] ont montré qu'en cas de défaillance définitive touchant un seul processus, il est impossible de résoudre le problème du consensus de manière déterministe dans un système asynchrone.

Les algorithmes robustes sont donc privilégiés lorsque le système est confronté à des défaillances périodiques ou irrégulières. Cependant, afin de masquer la présence de défaillances, il est nécessaire de rajouter lors de chaque opération de communication un contrôle pour vérifier l'état du système et empêcher les défaillances de se propager ou d'influencer le système. Ceci est relativement coûteux et les solutions robustes sont très souvent pénalisées par la perte en efficacité qu'elles impliquent.

2.3.2 Les algorithmes auto-stabilisants

La conception des algorithmes auto-stabilisants se base sur une vision optimiste où on considère que les défaillances sont suffisamment rares pour ne pas être significatives (c'est-à-dire limitées dans le temps), mais elles peuvent concerner tous les éléments du système. En revanche, comme ces défaillances peuvent corrompre complètement le système, il est nécessaire de considérer l'état initial comme étant quelconque, contrairement à l'algorithmique répartie classique où on choisit son état initial. Ainsi, après un ensemble de défaillances, le système est dans un état arbitraire. La propriété d'auto-stabilisation garantit qu'il atteindra au bout d'un temps fini une configuration dite légitime, à partir de laquelle il se comporte suivant sa spécification.

Les algorithmes auto-stabilisants sont typiquement non-masquants, car entre le moment où les défaillances cessent et le moment où le système est stabilisé, ce dernier ne se comporte pas correctement.

2.4 L'auto-stabilisation

Le concept d'auto-stabilisation a été introduit en 1974 par Dijkstra dans [15]. Un système réparti y est défini comme auto-stabilisant s'il se comporte selon sa spécification au bout d'un temps fini quelque soit sa configuration initiale. Par conséquent, un système réparti auto-stabilisant tolère des défaillances transitoires de processus et de liens de communication [40].

Le comportement d'un système auto-stabilisant peut être décomposé en deux phases. Pendant la première phase, nommée phase de stabilisation, le système ne se comporte pas correctement : l'exécution ne vérifie pas la spécification du système (on parle alors de configurations illégitimes). Après un temps fini (temps de stabilisation) et en l'absence de nouvelles défaillances, le système est dans une phase stabilisée dans laquelle le système se comporte correctement : l'exécution vérifie toujours la spécification (on parle alors de configurations légitimes). Il faut préciser que le retour à un comportement normal se fait sans aucune aide extérieur, que ce soit pour détecter la défaillance ou pour y remédier. Ces différents aspects sont résumés sur la figure 2.3.

Les systèmes répartis auto-stabilisants tolèrent des défaillances transitoires sous couvert de deux hypothèses :

1. Un code fiable : les défaillances supportées sont supposées n'affecter que la partie volatile des composants du système. Le code de l'algorithme ne peut être modifié par de telles défaillances (algorithme stocké en mémoire morte).

2. Une faible fréquence des défaillances : les défaillances ne doivent pas se produire trop fréquemment pour que le système puisse retrouver un comportement correct, sinon il restera continuellement en phase de stabilisation.

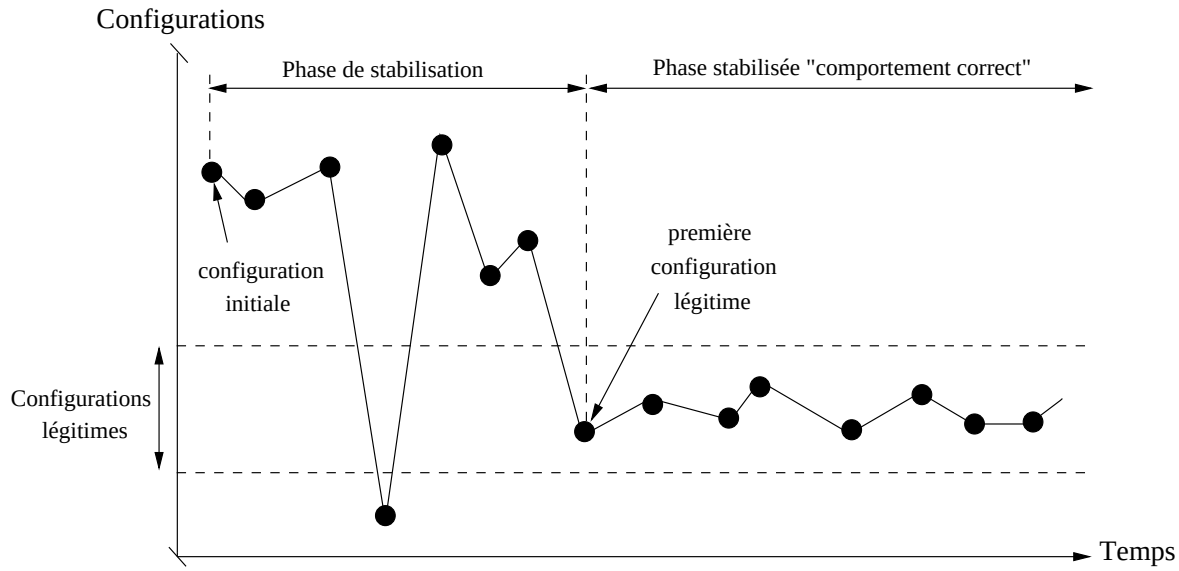


FIG. 2.3 – L'auto-stabilisation.

La propriété de dynamicité des algorithmes répartis est en rapport avec la propriété d'auto-stabilisation. Un algorithme est dit dynamique si son code n'a pas besoin d'être modifié en cas d'ajout ou de retrait de processus ou de liens de communication dans le système [34]. Alors le système tolère des changements de topologie pendant son fonctionnement. Il convient de remarquer que tous les algorithmes auto-stabilisants dont la complexité en nombre d'états par processus ne dépend pas d'un paramètre global sont dynamiques [19].

2.4.1 Avantages et limites de l'auto-stabilisation

En plus de la tolérance aux défaillances transitoires, les algorithmes auto-stabilisants présentent d'autres avantages :

- Aucune initialisation particulière n'est requise pour assurer le bon fonctionnement de l'algorithme. En effet, même si chacun des processus démarre dans un état quelconque, le système va finir par se comporter correctement.
- Aucune restriction n'est faite sur le nombre de processus sujets aux défaillances. Tous les processus peuvent être simultanément sujets aux défaillances, le système est capable de retrouver un comportement correct.
- Les algorithmes auto-stabilisants fonctionnent de la même manière pour des topologies de réseaux dynamiques à condition que les hypothèses d'exécution restent respectées. Un algorithme auto-stabilisant qui calcule une fonction dépendante de la topologie converge vers une nouvelle solution après que la topologie du réseau ait été modifiée.

Par contre, lorsqu'on utilise des algorithmes auto-stabilisants, plusieurs problèmes surviennent :

- A l'origine, l'exécution de l'algorithme ne correspond pas nécessairement à une exécution correcte. En effet, le système doit être capable de supporter la non-conformité à sa spécification pendant un certain temps.
- L'incapacité de détecter la terminaison de l'algorithme. Comme chaque processus n'a qu'une vision locale de l'état du système, les processus ne sont pas capables de s'apercevoir qu'une configuration légitime a été atteinte. Cela les oblige à exécuter leur code en permanence.
- En général, les performances des algorithmes auto-stabilisants sont inférieures à celles de leurs équivalents non auto-stabilisants lorsqu'il n'y a pas de défaillances transitoires.
- L'auto-stabilisation n'impose aucune contrainte sur le temps pendant lequel le système peut ne pas se comporter correctement (temps de stabilisation). La seule garantie est que ce temps est fini.
- Les algorithmes auto-stabilisants utilisent les mêmes mécanismes de correction quelle que soit la gravité de la défaillance : elle peut affecter un seul composant ou la totalité des composants du système.

2.4.2 Auto-stabilisation améliorée

Pour palier aux certaines limites de l'auto-stabilisation, des améliorations ont été proposées. Nous présentons maintenant les approches qui se sont développées dans cette optique.

2.4.2.1 k-stabilisation

La k-stabilisation est une généralisation de l'auto-stabilisation. Elle a été définie dans [9]. L'idée de base est d'obtenir un temps de stabilisation plus court, en faisant l'hypothèse qu'une borne supérieur $k \leq n$ sur le nombre de défaillances est connue, où n est le nombre total de processus dans le système. En effet, un système k-stabilisant frappé par des défaillances affectant plus de k processus n'est pas sûr de se stabiliser. Dans le cas où k est la taille du réseau ($k = n$), les concepts de k-stabilisation et d'auto-stabilisation sont identiques.

2.4.2.2 Adaptabilité en temps

L'adaptabilité en temps (en anglais *time adaptive stabilization*, *scalable stabilization* ou encore *fault local stabilization*) a été introduite dans [38]. L'idée est de relier le temps de stabilisation au nombre de défaillances. Cela signifie que lorsque moins de défaillances frappent le système, la stabilisation soit plus rapide.

2.4.2.3 Confinement de fautes

La notion de confinement de fautes [23] consiste à prendre des précautions particulières pour empêcher les défaillances de se propager dans le système. Ainsi, un algorithme confinant les fautes est un algorithme qui a la capacité de limiter les effets des défaillances transitoires à un petit nombre de composants du système.

2.4.2.4 Super-stabilisation

Dans un système à grande échelle, les aspects de dynamicité et de changement imprévu de l'environnement sont beaucoup plus susceptibles de se produire que des corruptions

arbitraires de la mémoire des nœuds du système. En effet, La super-stabilisation (définie dans [18]) stipule qu'un algorithme super-stabilisant est auto-stabilisant d'une part, et préserve un prédicat (typiquement un prédicat de sûreté) quand des changements de topologie surviennent dans une configuration légitime. Ainsi, si ces changements interviennent seulement après qu'une configuration légitime a été atteinte, le système reste stable.

2.4.2.5 Stabilisation instantanée

Le concept de stabilisation instantanée a été introduit pour la première fois dans [14]. C'est la propriété pour un système auto-stabilisant d'avoir un temps de stabilisation nul, cela signifie que toutes les configurations sont légitimes. Autrement dit, un système instantanément stabilisant vérifie toujours sa spécification quelle que soit la configuration initiale.

2.4.3 Définitions

Cette section propose une formalisation de la définition des systèmes auto-stabilisants, en utilisant les définitions d'un système réparti, d'une exécution et d'une spécification présentées précédemment dans le chapitre 1 (Définitions 1, 2, 3).

L'exécution d'un système auto-stabilisant ne se termine pas et les processus doivent continuellement communiquer avec leurs voisins. La définition de l'auto-stabilisation admise par la quasi-totalité de la communauté des systèmes auto-stabilisants est celle donnée dans [48] :

Définition 4 *Un système $S = (\mathcal{C}, \mathcal{T})$ est **auto-stabilisant** pour une spécification $\mathcal{P}$ si et seulement si il existe un sous-ensemble $\mathcal{C}_\mathcal{L}$ de configurations de S ($\mathcal{C}_\mathcal{L} \subseteq \mathcal{C}$), dit ensemble des configurations légitimes, tel que :*

1. *(correction) toute exécution de S dont la première configuration est dans $\mathcal{C}_\mathcal{L}$ vérifie $\mathcal{P}$;*
2. *(convergence) toute exécution de S contient au moins une configuration dans $\mathcal{C}_\mathcal{L}$.*

Il existe une forme un peu plus faible de la définition d'un système auto-stabilisant qui peut être utilisée dans le cadre de problèmes statiques. Dans laquelle, la propriété de correction s'exprime sous la forme d'une clôture aux configurations légitimes.

(clôture) l'ensemble des configurations légitimes $\mathcal{C}_\mathcal{L}$ est clos : toute exécution de S dont la première configuration est dans $\mathcal{C}_\mathcal{L}$ n'atteint aucune configuration illégitime.

Dans la suite nous détaillons ces trois propriétés fondamentales des systèmes répartis auto-stabilisants : la correction, la convergence et la clôture.

2.4.3.1 Correction

Le concept de correction est couramment utilisé dans l'algorithmique répartie. Il exprime simplement le fait que l'algorithme se comporte de la manière attendue (produit le bon résultat, ou le bon comportement). Autrement dit, la correction est la garantie que toute exécution ayant une configuration initiale légitime satisfait la spécification du problème.

2.4.3.2 Convergence

On peut distinguer deux types de convergence [25] : la convergence forte et la convergence faible. La convergence faible est une variante affaiblie de la définition de la convergence forte.

Définition 5 *Considérons un système réparti $S = (\mathcal{C}, \mathcal{T})$. On dit que S* ***converge fortement*** *vers un sous-ensemble $\mathcal{C}_\mathcal{L}$ de $\mathcal{C}$ si toute exécution de S atteint une configuration de $\mathcal{C}_\mathcal{L}$.*

Définition 6 *Considérons un système réparti $S = (\mathcal{C}, \mathcal{T})$. On dit que S* ***converge faiblement*** *vers un sous-ensemble $\mathcal{C}_\mathcal{L}$ de $\mathcal{C}$ si à partir de toute configuration atteinte lors d'une exécution de S, il existe une exécution permettant d'atteindre une configuration de $\mathcal{C}_\mathcal{L}$.*

La convergence est la propriété clé des systèmes auto-stabilisants. Elle assure que toute exécution du système atteindra une configuration légitime (convergence forte).

2.4.3.3 Clôture

Dans [25] l'auteur définit deux types de clôture : la clôture forte et la clôture faible. D'après les définitions ci-après, il est à noter que tout ensemble fortement clos est également faiblement clos, mais que la réciproque n'est pas vrai.

Définition 7 *Considérons un système réparti $S = (\mathcal{C}, \mathcal{T})$. Un sous-ensemble $\mathcal{C}_\mathcal{L}$ de $\mathcal{C}$ est* ***fortement clos*** *si chaque transition de $\mathcal{T}$ à partir d'une configuration de $\mathcal{C}_\mathcal{L}$ atteint une configuration de $\mathcal{C}_\mathcal{L}$.*

Définition 8 *Considérons un système réparti $S = (\mathcal{C}, \mathcal{T})$. Un sous-ensemble $\mathcal{C}_\mathcal{L}$ de $\mathcal{C}$ est* ***faiblement clos*** *si pour toute exécution de S dont la première configuration est dans $\mathcal{C}_\mathcal{L}$, il existe un suffixe où chaque configuration atteinte est dans $\mathcal{C}_\mathcal{L}$.*

La notion de clôture est particulière au domaine de l'algorithmique auto-stabilisante. Elle exprime le fait que l'ensemble des configurations légitimes décrit l'intégralité du comportement du système une fois la convergence obtenue. Ainsi, toute configuration accessible à partir d'une configuration légitime est, elle même, une configuration légitime (clôture forte).

2.4.4 Auto-stabilisation affaiblie

A partir des définitions de la convergence et de la clôture, on peut déduire trois variantes de l'auto-stabilisation. Ces concepts présentent des affaiblissements de la définition d'auto-stabilisation :

1. Convergence forte avec clôture faible : Pseudo-stabilisation [12].
2. Convergence faible avec clôture forte : Auto-stabilisation probabiliste [31].
3. Convergence faible avec clôture faible : Pseudo-stabilisation probabiliste.

La figure 2.4 montre une façon d'ordonner l'auto-stabilisation et ces variantes selon la contrainte qu'elles imposent sur les exécutions.

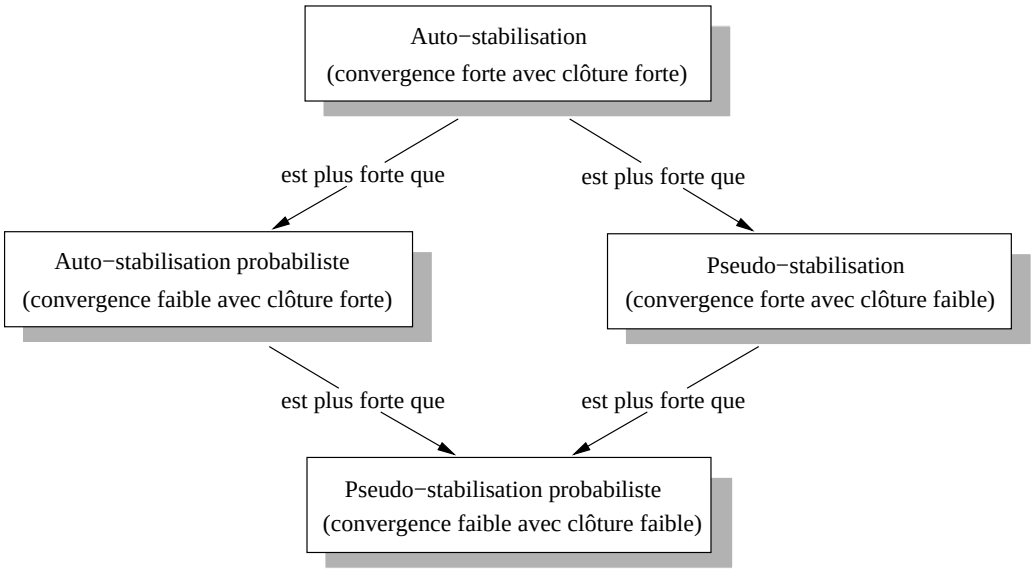


FIG. 2.4 – Variantes affaiblies de l'auto-stabilisation.

2.4.5 Preuve de l'auto-stabilisation

Les systèmes auto-stabilisants se prouvent en deux phases : la phase de correction et la phase de convergence. Prouver la correction d'un système auto-stabilisant est en générale, la tâche la plus facile. Cette phase ne diffère pas des preuves de correction des systèmes répartis classiques : partant de l'ensemble de configuration initiales (les configurations légitimes dans le cas des systèmes auto-stabilisants), il faut prouver que toute exécution effectuée à partir de ces configurations satisfait la spécification de la tâche du système.

En ce qui concerne la preuve de convergence, la tâche est plus difficile. Il est nécessaire de montrer qu'il n'existe pas d'exécution ne contenant que des configurations illégitimes. La méthode la plus utilisée est celle de la fonction décroissante [37] ou sa généralisation : la méthode des attracteurs [26].

Preuves par fonction décroissante

Cette technique consiste à définir une fonction à valeurs entières positives sur les configurations du système. Cette fonction doit avoir deux caractéristiques :

1. Elle atteint une valeur minimale dans chacune des configurations légitimes.
2. Elle décroît strictement à chaque exécution d'une action du système à partir d'une configuration illégitime.

Puisque l'ensemble des entiers est bien ordonné, une telle fonction garantit qu'à partir de toute configuration du système, et pour toute exécution possible à partir de cette configuration, le système finit par atteindre une configuration légitime (il n'existe pas de suite strictement décroissante non bornée inférieurement dans $\mathbb{N}$ "l'ensemble des entiers").

Preuves par attracteurs

La méthode des attracteurs est une généralisation de la méthode de la fonction décroissante. Elle consiste à démontrer que le système converge successivement d'un ensemble de configurations vers un autre plus petit. Cette technique s'applique lorsqu'il est trop difficile de trouver intuitivement une fonction décroissante. Un attracteur est un ensemble de configurations tel qu'à partir de toute configuration d'un ensemble donné, toute exécution atteint une configuration de l'attracteur. Prouver la convergence avec des attracteurs revient alors à trouver des ensembles de configurations $C_1, ..., C_n$ tels que pour tout $i \in \{2, ..., n\}$, C_i est un attracteur pour C_{i-1}, et tels que $C_1 = \mathcal{C}$ et C_n est l'ensemble des configurations légitimes.

2.5 Mesures de complexité

Afin de comparer les performances de différents algorithmes répartis, il est donc nécessaire de définir des mesures de complexité. Cette complexité s'exprime en espace et en temps.

2.5.1 Complexité en espace

La complexité en espace d'un algorithme réparti est simplement la taille en nombre de bits des variables utilisées par chaque processus, ainsi que la taille des registres (dans le cas du modèle de communication à registres) utilisés par les processus pour communiquer entre eux. Dans le modèle de communication par échange de messages, on considère séparément la taille et le nombre des messages qui circulent sur le réseau afin d'évaluer les ressources en bande passante utilisées par cet algorithme. On peut distinguer trois classes importantes de complexité en espace [27] :

1. Constante : la taille mémoire utilisée par chacun des composants ne dépend pas des caractéristiques du système.

2. Degré : la taille mémoire utilisée par chacun des composants de base est seulement dépendante du degré maximal d'un processus[1] ou du graphe[2] de communication du système.

3. Fonction de la taille du système : la taille mémoire utilisée par chacun des composants de base est une fonction dépendante de la taille du système.

2.5.2 Complexité en temps

Du fait que les différents composants du système évoluent à des vitesses différentes, il est difficile de définir un temps global absolu permettant d'évaluer si la tâche est résolue rapidement ou non, ou de comparer des implantations de systèmes répartis destinés à résoudre une même tâche. Pour cette raison, il existe dans la littérature diverses unités logiques pour mesurer la complexité en temps. Dans les systèmes synchrones, l'unité de mesure couramment admise est la *phase synchrone*. Alors que dans le cadre de l'algorithmique asynchrone, plusieurs unités de mesure ont été proposées, comme le *pas atomique* et le *round asynchrone*. Parmi les différentes définitions du round asynchrone, la définition la plus couramment admise est le temps nécessaire pour que tous les processus activables aient fait au moins un pas atomique.

Etant donné un système réparti, il est possible d'étudier plusieurs mesures temporelles comme [49] :

1. *Temps de stabilisation* : le temps maximum nécessaire pour atteindre une configuration légitime à partir d'une configuration quelconque.

2. *Temps de calcul* : le temps maximum nécessaire pour atteindre une configuration légitime d'une tâche statique à partir d'une configuration à connaissances purement locales.

3. *Temps de service* : le temps maximum nécessaire entre deux occurrences d'une même configuration légitime pour une tâche dynamique.

[1]Le degré d'un sommet d'un graphe non orienté est le nombre d'arêtes qui lui sont incidentes.
[2]Le degré d'un graphe est le plus grand degré d'un sommet dans le graphe.

2.6 Conclusion

Dans ce chapitre, nous avons présenté les différents types de défaillances et les algorithmes qui tolèrent ces défaillances. Une classe importante d'algorithmes tolérant certaines défaillances transitoires sont les algorithmes auto-stabilisants. Après avoir présenté la notion d'auto-stabilisation, nous avons donné des définitions formelles de cette notion et de ses variantes selon les trois propriétés classiques : correction, convergence et clôture. Ensuite, nous avons exposé diverses techniques utilisées pour prouver l'auto-stabilisation. Enfin, nous avons présenté différentes mesures de complexité utilisées pour comparer les performances des algorithmes répartis.

CHAPITRE

3 Auto-stabilisation automatique d'algorithmes

Souvent, un algorithme ne possède pas la propriété d'auto-stabilisation, mais doit être exécuté dans un environnement où des défaillances peuvent survenir, ou bien la topologie du réseau peut être modifiée pendant l'exécution de l'algorithme. Il est alors intéressant de disposer d'un " modèle d'auto-stabilisation automatique ". C'est-à-dire un compilateur qui prend en entrée un algorithme non auto-stabilisant et qui fournit en sortie un autre algorithme qui effectue la même tache que le protocole fourni en entrée, mais qui possède la propriété d'auto-stabilisation.

3.1 Classification des modèles d'auto-stabilisation automatique

Les modèles d'auto-stabilisation automatique peuvent être regroupés en deux classes selon le type d'algorithmes auxquels nous voulons ajouter la propriété d'auto-stabilisation : algorithmes séquentiels ou répartis. Ainsi, la première classe regroupe les modèles d'auto-stabilisation automatique des algorithmes séquentiels en algorithmes répartis et auto-stabilisants, et la deuxième regroupe les modèles qui ajoutent la propriété d'auto-stabilisation aux algorithmes répartis.

L'idée principale, pour ajouter la propriété d'auto-stabilisation aux algorithmes répartis, est de supposer que toute incohérence peut être détectée, et que cette détection déclenche une opération de correction du système. La détection d'incohérence peut être le résultat d'une vérification locale de l'état d'un processus et ceux de ses voisins, ou d'une vérification

globale de la configuration du système. De même, la correction peut être locale ou globale. La correction locale concerne l'état d'un seul processus. Cependant, la correction globale peut concerner les états de tous les processus du système.

La distinction entre la vérification locale et globale, et entre la correction locale et globale, nous permet de déduire quatre sous-classes de modèles qui ajoutent la propriété d'auto-stabilisation aux algorithmes répartis :

1. Vérification globale avec correction globale.
2. Vérification locale avec correction globale.
3. Vérification locale avec correction locale.
4. Vérification globale avec correction locale.

Il est à noter qu'il n'y a pas de modèles d'auto-stabilisation automatique dans la quatrième sous-classe, car les problèmes corrigeables localement sont en générale vérifiables localement. De ce fait, il est préférable en terme de performances d'utiliser un mécanisme de vérification locale que d'utiliser un mécanisme de vérification globale.

On peut distinguer un autre mécanisme pour ajouter la propriété d'auto-stabilisation aux algorithmes qui résolvent des problèmes non-réactifs[1]. Il consiste à résoudre de façon auto-stabilisante le problème du maintien de la topologie, en coloriant chaque nœud de cette topologie par l'état du processus qu'il représente. Il s'agit de construire sur chaque processus une représentation complète de la configuration du système. Ainsi, tout processus peut simuler localement un algorithme réparti non-réactif, et calculer l'état qu'il devrait avoir. Ce mécanisme peut être utilisé aussi pour transposer une classe d'algorithmes séquentiels en algorithmes répartis et auto-stabilisants. Cette classe d'algorithmes séquentiels ne contient que les algorithmes déterministes de graphe. Dans ce cas, il suffit de maintenir sur chaque processus le graphe qui représente la topologie du système, au lieu de construire une représentation complète de la configuration du système. La figure 3.1 permet de résumer la classification précédente.

Dans ce qui suit, nous allons utiliser cette classification pour organiser les modèles d'auto-stabilisation automatique étudiés dans ce mémoire.

[1]Les problèmes dont la spécification consiste à obtenir une fonction de la topologie et de la configuration initiale du système.

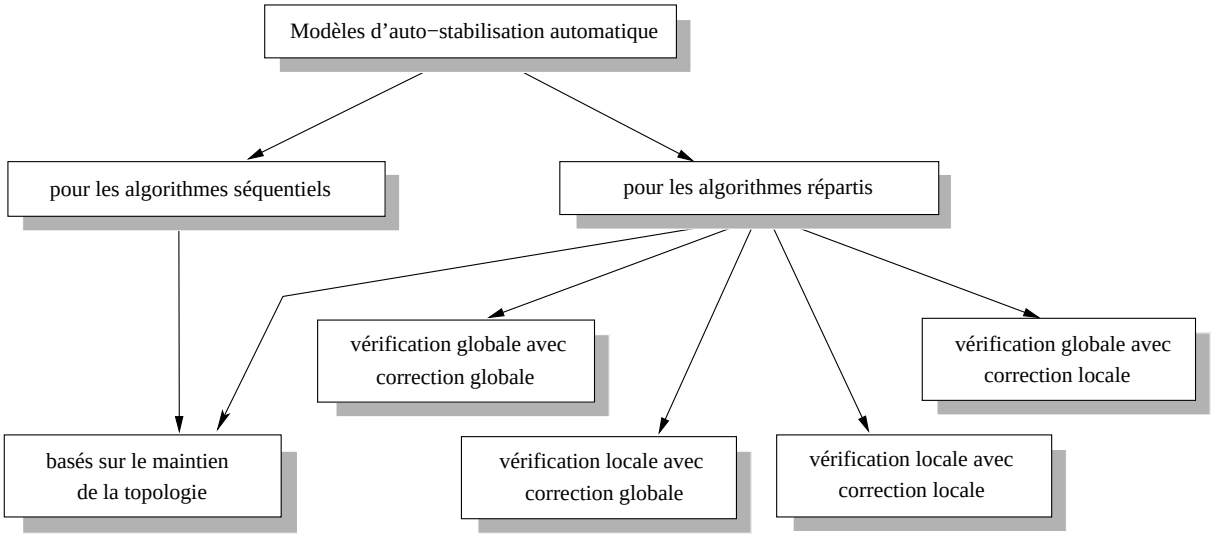


FIG. 3.1 – Classification des modèles d'auto-stabilisation automatique.

3.2 Modèles basés sur le maintien de la topologie

Comme nous l'avons mentionné précédemment, il s'agit des modèles où chaque processus simule localement des algorithmes sur une configuration globale du système. Cette configuration doit être calculée de façon répartie. La simulation permet au processus de corriger son état local s'il est incorrect.

3.2.1 Modèle de Kutten et Patt-Shamir

Dans [38], les auteurs ont proposé un algorithme auto-stabilisant et adaptable en temps pour le problème du bit persistant. Ce problème consiste à maintenir la valeur d'un bit commun dupliqué à travers le système. C'est-à-dire que tout processus du système maintient une valeur booléenne (vrai ou faux, ou ici 1 ou 0) et que cette valeur doit être la même pour tous les processus du système. Cela malgré les défaillances transitoires qui peuvent corrompre arbitrairement les états des processus, à condition que les états de la majorité des processus ne soient pas corrompus.

En utilisant cet algorithme, les auteurs présentent un transformateur qui prend un algorithme réparti non auto-stabilisant pour un problème non-réactif $\mathcal{P}$ et produit un algorithme auto-stabilisant $\mathcal{P}'$ qui résout le même problème. L'idée de ce transformateur est de dupliquer les entrées de $\mathcal{P}$ dans tous les processus du système. Chaque bit de ces

entrées est maintenu par une instance de l'algorithme de bit persistant. De cette façon, toutes les entrées sont disponibles localement dans chaque processus du système qui peut donc simuler $\mathcal{P}$ localement dans une seule unité de temps.

Ce transformateur n'est pas efficace, puisque l'overhead en espace est très grand. Mais, il peut être vu comme un résultat d'existence. C'est-à-dire, pour chaque algorithme résolvant un problème non-réactif, il existe une version auto-stabilisante qui résout le même problème.

3.2.2 Modèle de Dolev

Le principe du modèle de Dolev décrit dans [16] est le même que celui de Kutten et Patt-Shamir [38]. C'est-à-dire que le modèle est basé sur la simulation d'un algorithme réparti non-réactif sur une configuration globale du système maintenue dans chaque processus. Cependant, dans le modèle de Dolev et à la place de l'algorithme de bit persistant, la configuration globale du système est maintenue dans chaque processus en utilisant un algorithme de maintien de la topologie.

L'algorithme présenté pour le maintien de la topologie détermine d'abord la relation d'adhésion : chaque processus connaît les processus qui appartiennent à sa composante connexe, et puis les informations sur les statuts des liaisons sont rassemblées en utilisant un simple arbre. Cet algorithme est auto-stabilisant et dynamique. Mais, il suppose une borne sur le nombre des processus dans le système.

Le coût de cette technique est principalement en espace : chaque processus maintient une copie de l'état des autres processus du système.

3.3 Modèles utilisant une vérification globale et une correction globale

Le seul modèle dans ce chapitre qui utilise des opérations globales pour vérifier et corriger le système est celui de Katz et Perry [36].

3.3.1 Modèle de Katz et Perry

Le travail de Katz et Perry [36] fournit un modèle général pour convertir des algorithmes non-stabilisants en algorithmes auto-stabilisants dans un système asynchrone avec échange de messages. Ceci est accompli par l'introduction d'une plateforme auto-stabilisante. La superposition d'un algorithme non-stabilisant sur cette plateforme produit une version auto-stabilisante de l'algorithme. Leur modèle est limité aux systèmes pour lesquels il existe un prédicat pour déterminer si une configuration globale du système est légitime ou non.

Dans ce modèle, on trouve trois composants superposés : un algorithme auto-stabilisant pour déterminer l'état global d'un système réparti (une version auto-stabilisante de l'algorithme de Chandy et Lamports [13]), un algorithme auto-stabilisant de reset et l'algorithme à auto-stabiliser.

L'idée de base du modèle est la suivante : un processus distingué prend périodiquement des *snapshot*s de l'état global du système. Il est supposé qu'il existe un prédicat qui identifie les configurations globales illégitimes. Une fois que le processus distingué obtient un *snapshot*, il évalue ce prédicat. Si l'évaluation indique une configuration illégitime, l'exécution de l'algorithme de reset est lancée mettant la configuration globale du système à une certaine configuration légitime prédéfinie.

Les algorithmes de *snapshot* et de reset peuvent être considérés comme une "plateforme" auto-stabilisante. Tout algorithme superposé sur cette plateforme devient auto-stabilisant.

La supposition d'existence d'un prédicat par lequel on peut différencier entre les configurations légitimes et illégitimes n'est pas acceptable dans tous les cas. La plupart des algorithmes ne sont pas conçus avec une définition précise d'une configuration légitime ou illégitime. Un autre point intéressant à noter est que l'algorithme de *snapshot* ne produit pas la configuration actuelle, mais plutôt un successeur possible de la configuration de laquelle il a été lancé.

Ce modèle est général, mais la vérification centralisée et la nécessité d'élire un chef avec la complexité du modèle de communication par échange de messages réduisent sa performance.

3.4 Modèles utilisant une vérification locale et une correction globale

3.4.1 Modèle de Arora et Gouda

L'article [3] décrit comment modifier un système réparti arbitraire de sorte que chacun de ses processus puisse remettre le système à un état global prédéfini dès qu'il est nécessaire. La modification n'introduit pas de nouveaux processus ou de nouvelles voies de transmission au système. Elle présente simplement les modules additionnels aux processus existants. Les modules supplémentaires, communiquant entre eux à travers les canaux existants, comportent ce que on appelle le sous-système de reset.

Il y a beaucoup d'occasions dans lesquelles il est souhaitable que quelques processus dans un système réparti lancent des resets par exemple :

Reconfiguration : Quand le système est modifié, par exemple, en ajoutant des processus ou des canaux de communication.

Changement de mode : Le système peut être conçu pour s'exécuter dans différents modes ou phases. Si c'est le cas, alors changer le mode d'exécution courant peut être réalisé en remettant le système à un état global approprié au mode prochain.

Perte de coordination : Quand un processus observe un comportement inattendu d'autres processus, il identifie que la coordination entre les processus dans le système a été perdue. Dans une telle situation, la coordination peut être regagnée par un reset.

Maintenance périodique : Le système peut être conçu de telle sorte qu'un processus désigné lance périodiquement un reset comme précaution.

Ce sous-système de reset peut tolérer la perte de coordination entre différents processus dans le système (qui peut être provoqué par des fautes transitoire ou perte de mémoire), et également, peut tolérer le fail-stop et la réparation d'un sous-ensemble de processus et de canaux. Il est conçu d'une façon modulaire simple. La conception se compose de trois composants principaux. Chacun de ces composants est auto-stabilisant :

- élection d'un leader.
- construction d'arbre couvrant.
- distribution de calcule.

Chaque processus $P.i$ dans un système réparti, se compose de deux modules : $adj.i$ et $app1.i$ (figure 3.2(a)). La tâche du module $adj.i$ est de maintenir un ensemble $N.i$ des identificateurs de tous les processus voisins de $P.i$. Alors que le module $appl.i$ est la spécification de l'application du système.

L'augmentation de tel système réparti avec un sous-système de reset consiste à ajouter deux modules à chaque processus $P.i$: $tree.i$ et $wave.i$ (figure 3.2(b)).

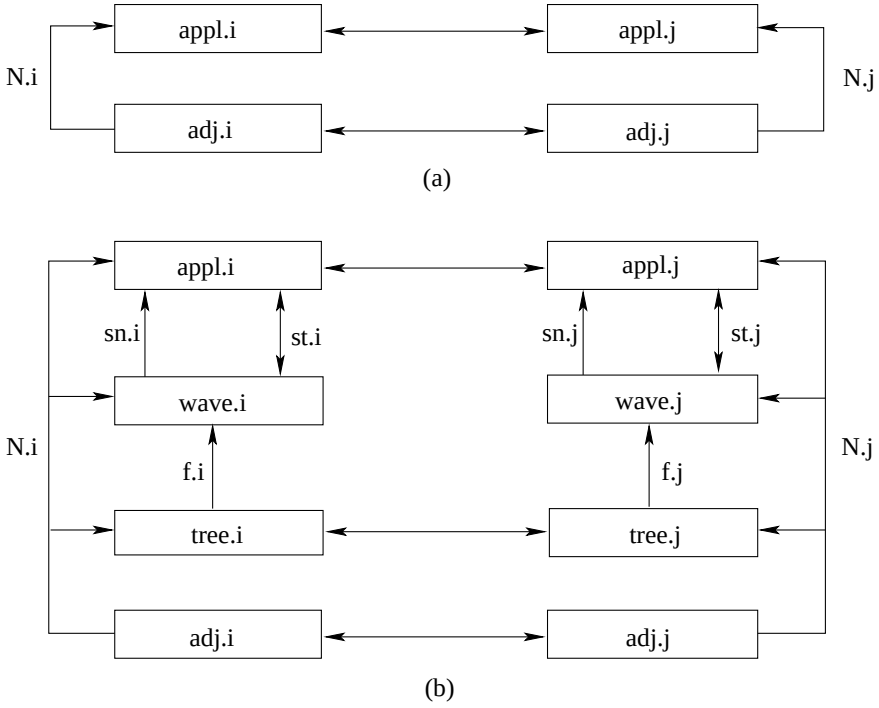


FIG. 3.2 – Reset distribué [3].

Les modules $tree.i$ des processus voisins communiquent afin de maintenir un arbre couvrant sur tous les processus opérationnels dans le système. L'arbre couvrant maintenu est conforme à la relation de voisinage courante. Ainsi, tout changement de la relation de voisinage (changement de la topologie du système) est par la suite suivi de changement correspondant dans l'arbre couvrant. Chaque module $tree.i$ garde l'index de son processus "père" dans l'arbre maintenu. Cette information est utilisée par le module $wave.i$ dans l'exécution du reset.

Le reset est exécuté par les modules $wave.i$ en trois phases. Dans la première phase, un certain $appl.i$ demande un reset du système de son $wave.i$ local qui fait suivre la demande à la racine de l'arbre couvrant. Si d'autres demandes de reset sont faites dans d'autres

processus, alors ces demandes sont également expédiées au processus racine. Il est commode de penser que toutes ces demandes forment une seule demande de reset. Dans la deuxième phase, après la réception de la demande de reset, le module *wave.i* du processus racine remet l'état de son *appl.i* local à l'état *appl.i* dans l'état global prédéfini, et lance une "*vague de reset*". Quand la vague de reset atteint un processus feuille, il la reflète comme "*vague d'accomplissement*" vers le processus racine. Cette dernière vague constitue la troisième phase. En fin, quand la vague d'accomplissement atteint la racine, le reset est terminé, et un nouvel reset peut être commencée à chaque fois qu'un module *appl.i* considère qu'il est nécessaire.

Dans ce sous-système de reset, les auteurs supposent que le système se compose de K processus qui doivent posséder des identités distinctes. En plus, le graphe qui représente la topologie du système doit rester connexe. Ce qui n'est pas toujours garanti dans un système dynamique.

3.4.2 Modèle de Awerbuch *et al.*-1

Dans [7], les auteurs ont combiné la vérification locale et la correction globale afin d'obtenir un nouveau modèle d'auto-stabilisation. Ce modèle peut être utilisé pour auto-stabiliser une classe large d'algorithmes répartis en utilisant le modèle de communication par échange de messages.

Les configurations illégitimes sont détectées en utilisant un mécanisme de vérification locale. Ce mécanisme vérifie en parallèle les sous-systèmes de lien tel qu'un sous-système de lien est une paire de processus voisins et les liens entre eux (figure 3.3). Une fois une configuration illégitime est détectée, une action globale de correction "reset" est employée pour remettre le système dans une configuration légitime prédéfinie.

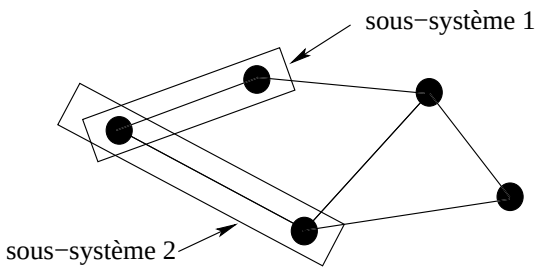


FIG. 3.3 – Sous-système de lien.

Ce modèle utilise le modèle de communication par échange de messages, ce qui le rend facile à implémenter. Cependant, il est limité aux algorithmes vérifiables localement. C'est-à-dire les algorithmes répartis qui vérifient la condition suivante : pour toute configuration illégitime du système, il existe un certain sous-système de lien qui pourra détecter ce fait localement.

3.5 Modèles utilisant une vérification locale et une correction locale

3.5.1 Modèle de Browne *et al.*

Le travail de Browne *et al.* [10] a été effectué dans le contexte des systèmes réactifs, plus précisément les systèmes de décision en temps réel qui réagissent aux lectures périodiques de capteurs. Dans ce contexte, l'état d'un processus se compose de ses variables internes et de variables d'entrées. Les variables d'entrées sont considérées incorruptibles. Ceci peut être motivé par le fait que dans un système réactif, les valeurs des variables d'entrée sont déterminées par des lectures à partir de l'environnement externe.

Browne *et al.* ont présenté une classe d'algorithmes pour lesquels il existe un compilateur qui ajoute la propriété d'auto-stabilisation. Ces algorithmes, qu'ils appellent *acycliques*, adoptent les critères suivants :

- Chaque règle affecte une valeur à une seule variable.
- Pour n'importe quelle paire de règles activables avec la même variable cible, les deux règles affectent la même valeur à cette variable.
- Le graphe de dépendance de données est acyclique (sans cycle).

Les auteurs supposent un démon séquentiel (les règles sont exécutées un par un), à condition que chaque règle soit exécutée infiniment souvent dans chaque calcul. Ils supposent également que les variables d'entrées ne sont pas corruptibles.

En utilisant ce modèle, la construction d'un algorithme auto-stabilisant P qui effectue la même tache qu'un algorithme acyclique non auto-stabilisant Q, peut être résumé dans l'algorithme (Algorithme 1).

Algorithm 1 Modèle de Browne *et al.*

Chaque règle dans Q est une règle dans P.

Pour chaque variable interne x_i dans Q, faire le suivant :

Laisser toutes les règles dans Q avec x_i dans le côté gauche :

$x_i := B_i(x)$ if $C_i(x)$ /* x représente le vecteur de toutes les variables de Q */

...

...

$x_i := D_i(x)$ if $E_i(x)$

Et ajouter la règle suivante à P :

$x_i := \hat{x}_i$ if $\overline{C_i(x)} \wedge ... \wedge \overline{E_i(x)}$ /* $\hat{x}_i$ est la valeur initiale de la variable x_i */

L'algorithme résultant P est acyclique, auto-stabilisant et effectue la même tache que Q.

Ce modèle est simple et efficace en terme d'espace mémoire et de temps de stabilisation, mais il n'est pas applicable qu'à une classe très limitée d'algorithme répartis : les algorithmes acycliques.

3.5.2 Modèle de Awerbuch *et al.*-2

Dans [6], les auteurs ont introduit une nouvelle méthode pour transformer les algorithmes répartis en algorithmes auto-stabilisants résolvant le même problème. Leur idée est de remplacer la vérification globale et centralisée présentée dans [36] par une vérification locale et décentralisée. En effet, ils ont divisé le système en un certain nombre de sous-systèmes de lien qui peuvent être vérifiés en parallèle. Un sous-système de lien se compose de deux processus voisins et les liens entre eux (figure 3.3).

Cette méthode n'est pas applicable qu'aux algorithmes corrigeables localement. Un algorithme réparti est localement corrigeable si on peut corriger toute configuration illégitime par la correction de chaque sous-système de lien de façon indépendante.

Le principe de cette méthode est simple : un chef est élu pour chaque sous-système de lien. Il vérifie périodiquement ce sous-système de lien et le corrige s'il y a des erreurs. Puisque la vérification et la correction procèdent en parallèle pour chaque sous-système de lien, ce modèle fournit une stabilisation rapide pour plusieurs algorithmes [51].

Cette méthode est limitée aux algorithmes vérifiables et corrigeables localement, mais évite l'inefficacité résultante de l'utilisation d'opérations globales.

3.5.3 Modèle de Varghese *et al.*

Tandis que la vérification et la correction centralisées de [36] peuvent être trop lentes, la condition pour un système d'être localement corrigeable de [6] peut être trop restrictive. Le modèle de Varghese *et al.* décrit dans [52] permet d'auto-stabiliser les algorithmes vérifiables localement mais qui peuvent être incorrigible localement. Ainsi, Varghese *et al.* ont enlevé la condition pour un système d'être localement corrigeable, mais ils ont ajouté la supposition que la topologie du système est en arbre.

Ce modèle se base essentiellement sur une action périodique de correction ajoutée à tous les processus autre que le processus racine de l'arbre qui est supposé incorruptible. Cette action de correction est simple : le processus lit l'état de son père dans l'arbre et vérifie si l'état du sous-système de lien[2] est correct. Dans le cas contraire, il le corrige par la modification de son état locale. L'objectif de ces actions de correction est d'arriver à une configuration globale légitime en corrigeant localement les sous-systèmes de lien qui sont dans des états incorrects.

Le point faible de ce modèle est l'hypothèse que la topologie du système est en arbre. En plus, il est limité aux algorithmes vérifiables localement.

3.6 Conclusion

Nous avons proposé dans ce chapitre une classification des modèles d'auto-stabilisation automatique. Cette classification est basée sur le type d'actions ou modules ajoutés à un algorithme réparti pour le rendre auto-stabilisant. Ensuite, nous avons présenté l'essentiel des modèles qui ont été introduits dans la littérature.

Le prochain chapitre est consacré à notre proposition.

[2] Avec le modèle de communication par mémoire partagée, comme il a été considéré dans cet article, un sous-ensemble de lien est juste une paire de processus voisins.

CHAPITRE

4 Maintien de la topologie pour l'auto-stabilisation automatique

Dans ce chapitre, nous nous intéressons au problème du maintien de la topologie dans les systèmes dynamiques : maintenir sur chaque processus une représentation de la topologie du système. Comme les changements de topologie dans de tels systèmes peuvent se produire à tout moment, un processus n'est jamais sûr de connaître la topologie correcte à un instant donné. Cependant, les algorithmes distribués peuvent être conçus pour garantir que chaque processus est mis au courant du statut correct de chaque lien auquel il a un chemin physique, à condition que la topologie ne change pas pendant suffisamment de temps. Pour ce propos, nous proposons un algorithme auto-stabilisant et dynamique pour le maintien de la topologie. Cet algorithme est uniforme et il ne suppose aucune borne sur le nombre de processus et/ou de liens dans le système.

En utilisant l'algorithme proposé, nous décrivons un modèle d'auto-stabilisation automatique des algorithmes séquentiels et déterministes de graphe en algorithmes répartis, auto-stabilisants et dynamiques qui résolvent le même problème.

4.1 Maintien de la topologie

Le maintien de la topologie dans les systèmes répartis et dynamiques revient à construire sur chaque processus un graphe qui représente la topologie du système, c'est-à-dire la liste des processus dans le système et les liens qui les relient. Pour ce problème, l'ensemble des configurations légitimes peut être caractérisé de la manière suivante : soit S un système

dont la topologie à un instant donné est représentée par le graphe $G = (V, E)$. Chaque processus p de S possède sa propre vision de la topologie du système notée Gp dans sa mémoire locale. Dans toute configuration légitime c, $\forall p \in V : G_p = (V, E)$. Il est à noter qu'il existe une unique configuration légitime pour ce problème pour un système donné (figure 4.1).

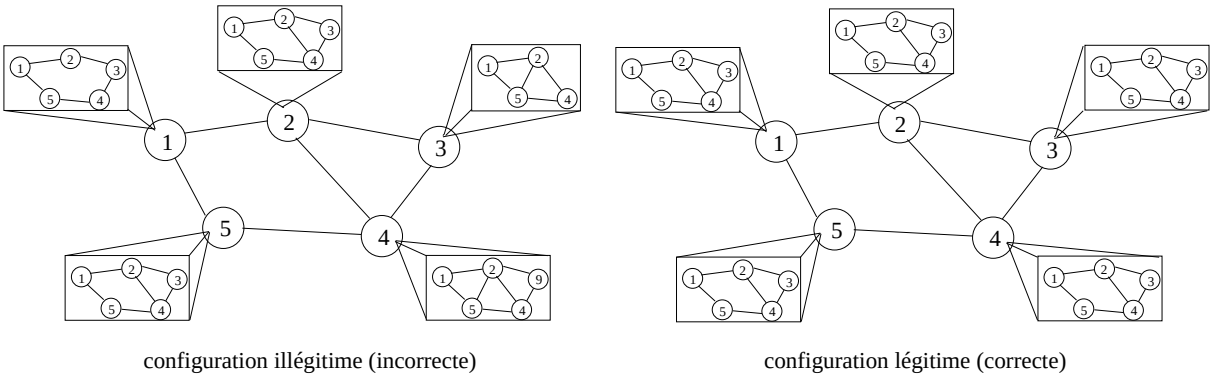


FIG. 4.1 – Maintien de la topologie.

Afin d'assurer qu'au moins un processus détecte une configuration illégitime, un algorithme auto-stabilisant de maintien de la topologie doit être conçu de telle sorte que chaque processus possède, en plus des connaissances locales, des connaissances concernant leurs voisins à distance 1 (le problème du maintien de la topologie est 1-distance locale [8]). Une façon possible de détecter une configuration illégitime consiste à comparer les états de tout couple de processus voisins. Si deux processus voisins ont des visions différentes du système, la configuration est illégitime. Cependant, cela ne suffit pas, car tous les processus pourraient avoir des visions équivalentes mais incorrectes du système. On ajoute alors la vérification que la représentation interne du graphe correspond à la vision effective du processus à distance 1. S'il existe un processus dont la topologie voisine (à distance 1) ne correspond pas à sa représentation interne, la configuration est également illégitime.

La résolution du problème de maintien de la topologie permet de résoudre tout problème de graphe, en calculant sur la représentation interne de la topologie du système une solution à ce problème. Comme chaque processus du système doit avoir une même représentation de la topologie, tous les processus obtiennent alors un même résultat au problème à résoudre.

4.2 Algorithme auto-stabilisant pour le maintien de la topologie

Nous proposons dans cette section un algorithme pour le maintien de la topologie. L'objectif de notre algorithme est que chaque processus, après le dernier changement de topologie ou bien après la dernière défaillance, doit arriver à connaître l'ensemble des liaisons opérationnelles dans le système. Pour ce faire, chaque processus se base sur des informations certaines qui sont les statuts des liens adjacents et des informations qui peuvent être incorrectes lues à partir des processus voisins.

Cette section est organisée comme suit. Dans un premier temps, nous présentons le modèle du système et les hypothèses utilisées par l'algorithme. Ensuite, nous expliquons le principe de l'algorithme, suivi de sa preuve. Enfin, nous présentons le calcul de la complexité en temps.

4.2.1 Modèle et hypothèses

Nous considérons un système réparti dynamique. Le réseau de communication du système à un instant donné est représenté par un graphe non-orienté $G = (V, E)$. L'ensemble V des sommets de G représente l'ensemble des processus (nœuds) et l'ensemble E des arêtes représente les liens bidirectionnels. Chaque lien est une paire non-ordonnée (p, q) de processus de V. Nous supposons que les processus possèdent des identifiants distincts. Si nous notons par $d(p, q)$ la distance[1] entre deux processus p et q, nous définissons la distance entre un processus p et un lien $u = (q, r)$ par le minimum des distances entre p et les extrémités de u, nous avons donc $d(u, p) = min(d(p, q), d(p, r))$.

Chaque processus dans le système peut communiquer directement avec certains processus appelés voisins. Nous dénotons par $\mathcal{N}(p)$ l'ensemble des voisins du processus p. Nous utilisons le modèle de communication par mémoire partagée, plus précisément c'est le modèle à état où chaque processus peut lire les états locaux des processus voisins, mais il ne peut mettre à jour que son état local.

[1]La distance entre deux sommets a et b d'un graphe G est le nombre minimal d'arêtes dans un chemin de a à b dans G.

Dans les systèmes dynamiques, les changements de topologie peuvent causer la partition du graphe de communication du système. Puisqu'il n'y a aucune communication possible entre deux composantes connexes distinctes, nous considérons chaque composante connexe comme un système indépendant qui contient un sous-ensemble de processus et de liens du système original (dans la suite nous utilisons le terme composante connexe et système indifféremment).

Nous supposons que l'exécution de chaque processus p est faite en cycles. Chaque cycle est constitué d'une séquence d'étapes "steps". Nous définissons l'étape comme unité atomique d'exécution. Elle est constituée des trois sous-étapes suivantes : (1) lire les états de tous les voisins, (2) calculer l'état local prochain, et (3) mettre à jour l'état local [35]. Nous disons qu'un round est fini lorsque chaque processus termine l'exécution d'un cycle, le deuxième round est fini après l'exécution d'un cycle pour chaque processus suite à la terminaison du premier round, et ainsi de suite.

Nous supposons aussi que chaque processus exécute la même étape en parallèle de façon synchronisée. Cette supposition semble forte. Cependant, un tel modèle d'exécution peut être réalisé sur le modèle asynchrone en utilisant un synchroniseur [33, 32].

4.2.2 Structure de données

Chaque processus p maintient un ensemble G_p qui représente sa vision courante de la topologie du système, et partage cet ensemble en lecture avec ses voisins. Un élément x de G_p est un couple $(x.link, x.dis)$, où $x.link$ représente un lien entre deux processus, et $x.dis$ est un nombre naturel qui représente la distance entre ce lien et le processus p après la stabilisation du système. Nous utilisons $G_q|p$ pour dénoter l'ensemble des éléments x de G_q tel que p est une extrémité de $x.link$. Pour calculer G_p, le processus p utilise un autre ensemble local T_p. Les éléments de T_p ont la même forme que celle des éléments de G_p.

4.2.3 Description de l'algorithme

Nous expliquons maintenant notre algorithme en détail. La description formelle de l'algorithme est montrée dans (Algorithme 2). L'algorithme est uniforme : tous les processus exécutent le même code. Cette exécution est faite en cycles, et chaque cycle est constitué de deux étapes.

Algorithm 2 Algorithme pour le maintien de la topologie

Process p

Shared Variable :

1 : G_p : set of $\{(u,k)\ /(u \in E)$ AND $(k \in \mathbb{N})\}$;

Local variable :

2 : T_p : set of $\{(u,k)\ /(u \in E)$ AND $(k \in \mathbb{N})\}$;

Functions :

3 : $\mathcal{N}(p) \equiv$ the set of neighbors of p ;

4 : $(\forall q \in \mathcal{N}(p) \cup \{p\}) : G_q|p \equiv \{((r,s),k) \in G_q\ /(r = p)$ OR $(s = p)\}$;

5 : $Cond_1 \equiv G_p|p \neq \{((p,q),0)\ /q \in \mathcal{N}(p)\}$;

6 : $Cond_2 \equiv \exists q \in \mathcal{N}(p) : \{x.link\ /x \in (G_p - G_p|p)\} \neq \{y.link\ /y \in (G_q - G_q|p)\}$;

7 : $Cond_3 \equiv \exists x \in (G_p - G_p|p) : x.dis \neq [1 + Min_{q \in \mathcal{N}(p)}(y.dis\ /(y \in (G_q - G_q|p))$ AND $(y.link = x.link))]$;

Guarded rules :

/* Step 1 : */

8 : R_0 : : $Cond_1 \longrightarrow$

9 : $G_p := (G_p - G_p|p) \cup \{((p,q),0)\ /q \in \mathcal{N}(p)\}$;

/* Step 2 : */

10 : R_1 : : $Cond_2$ OR $Cond_3 \longrightarrow$

11 : $T_p := \phi$;

12 : **for all** q such that $q \in \mathcal{N}(p)$ **do**

13 : $T_p := T_p \cup \{(x.link, x.dis + 1)\ /x \in (G_q - G_q|p)\}$;

14 : **for all** v such that $v \in \{y.link\ /y \in T_p\}$ **do**

15 : $T_p := \{(v, min_{(x \in T_p / x.link = v)}(x.dis))\} \cup (T_p - \{z \in T_p / z.link = v\})$;

16 : **end for**

17 : **end for**

18 : $T_p := T_p - \{x \in T_p /\ (\exists 1 \leq i < x.dis, \forall y \in T_p : y.dis \neq i)\}$;

19 : $G_p := G_p|p \cup T_p$;

Dans la première étape, chaque processus p vérifie que sa vision G_p de la topologie du système correspond à la vision effective du processus à distance 1. Pour cela, il effectue le teste suivant :

$$Cond_1 \equiv G_p|p \neq \{((p,q),0)\ /q \in \mathcal{N}(p)\}$$

Si la condition est vérifiée, le processus p remplace les liens incidents de p dans G_p par les liens opérationnels adjacents dans le système avec des distances égales à 0 (Algorithme 2, lignes 9). Ainsi, le processus p corrige sa vision G_p de la topologie du système par sa vision effective qui est l'ensemble des liens adjacents.

La deuxième étape permet à tout processus p de connaître les liens du système qui sont à distance supérieure ou égale à 1. Dans cette étape, chaque processus p effectue le teste suivant :

$Cond_2$ OR $Cond_3$ tel que :

$$Cond_2 \equiv \exists q \in \mathcal{N}(p) : \{x.link\ /x \in (G_p - G_p|p)\} \neq \{y.link\ /y \in (G_q - G_q|p)\}$$

$$Cond_3 \equiv \exists x \in (G_p - G_p|p) : x.dis \neq [1 + Min_{q \in \mathcal{N}(p)}(y.dis\ /(y \in (G_q - G_q|p))\ \text{AND}\ (y.link = x.link))]$$

Ce test permet au processus p de vérifier la correspondance entre sa vision du système et celles de ses voisins. Si le test est vrai (l'ensemble des liens qui ne sont pas incidents de p dans G_p et différent de celui dans la vision d'un voisin. Ou, il existe un lien non-adjacent de p dans G_p, tel que sa distance est différente de la distance minimale dans l'ensemble des éléments qui représentent ce lien dans les visions des voisins de p plus 1), le processus p recalcule dans T_p l'ensemble des liens qui sont à distance supérieure ou égale à 1, et l'ajoute à l'ensemble des liens adjacents dans G_p pour construire la nouvelle vision du système (Algorithme 2, lignes 19). Cela passe par plusieurs opérations. La première consiste à rassembler dans T_p les éléments qui appartiennent aux visions de tous les voisins, tel que p n'est pas une extrémité des liens que ces éléments représentent. Le champ de distance de ces éléments est incrémenté par 1 (Algorithme 2, lignes 13). Ensuite, pour chaque lien représenté dans T_p, on choisit l'élément qui possède la distance minimale (Algorithme 2, lignes 15). En fin, on supprime tous les éléments qui ont des distances supérieures à une distance k ($k > 0$), où il n'existe aucun élément dans T_p dont sa distance est égale à k (Algorithme 2, lignes 18).

Notre algorithme n'utilise jamais la taille du réseau ni une borne sur le nombre de processus et/ou de liens. Par conséquent, tout processus ou lien peut être dynamiquement rajouté au système, à condition que tout identifiant d'un processus soit unique.

Exemple d'exécution

Considérons un système réparti composé de cinq processus. Nous supposons que la configuration 0 présentée dans la figure 4.2 est la configuration initiale du système, ou la configuration résultante après qu'une défaillance transitoire vient perturber le système.

$G_1 = \{((1,2),0),((1,5),2),((3,4),0)\}$
$G_2 = \{((1,2),0),((1,5),1),((2,3),0),((2,4),0),$
$((3,4),1),((4,5),1)\}$
$G_3 = \{((1,2),1),((1,5),3),((2,4),2),((2,5),1),$
$((4,5),0)\}$
$G_4 = \{((5,8),1)\}$
$G_5 = \{((1,5),2),((2,5),0),((2,9),0),((4,5),1),$
$((4,9),0)\}$

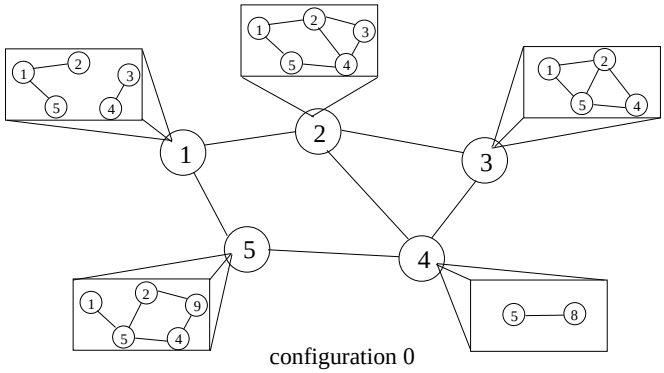


FIG. 4.2 – Configuration initiale.

L'exécution de notre algorithme à partir de la configuration 0 est décrite sur la figure 4.3. La configuration 1 résulte après un round de l'exécution, et la configuration 2 résulte après l'exécution d'un round en démarrant de la configuration 1, et ainsi de suite jusqu'à arriver à la configuration 4 (configuration légitime).

4.2.4 Preuve de l'auto-stabilisation

La preuve consiste à vérifier que l'exécution de l'algorithme finit par atteindre une configuration légitime (preuve de convergence) et puis, à partir de cet instant, qu'elle vérifie toujours la spécification de l'algorithme (preuve de correction).

4.2.4.1 Preuve de la convergence

Dans cette section nous démontrons que le système arrive à une configuration légitime après au plus $d+2$ rounds (d est le diamètre[2] du graphe qui représente la topologie initiale du système ou la topologie résultante après la dernière défaillance transitoire venant perturber le système).

[2]Le diamètre d'un graphe G est la plus grande distance entre deux sommets de G

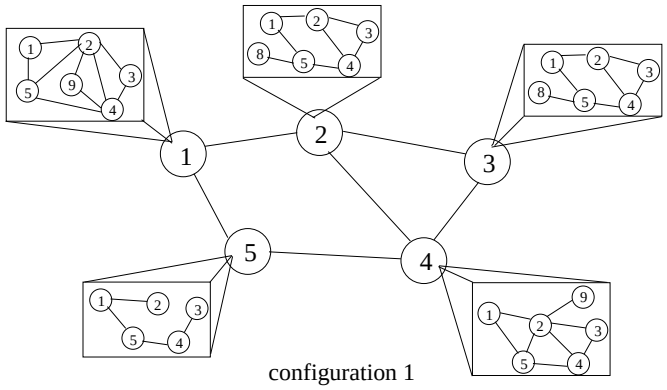
$G_1 = \{((1,2),0),((1,5),0),((2,3),1),((2,4),1),$
$((2,5),1),((2,9),1),((3,4),2),((4,5),2),$
$((4,9),1)\}$

$G_2 = \{((1,2),0),((1,5),3),((2,3),0),((2,4),0),$
$((3,4),1),((4,5),1),((5,8),2)\}$

$G_3 = \{((1,2),1),((1,5),2),((2,3),0),((2,4),1),$
$((3,4),0),((4,5),2),((5,8),2)\}$

$G_4 = \{((1,2),1),((1,5),2),((2,3),1),((2,4),0),$
$((2,5),1),((2,9),1),((3,4),0),((4,5),0)\}$

$G_5 = \{((1,2),1),((1,5),0),((3,4),1),((4,5),0)\}$



configuration 1

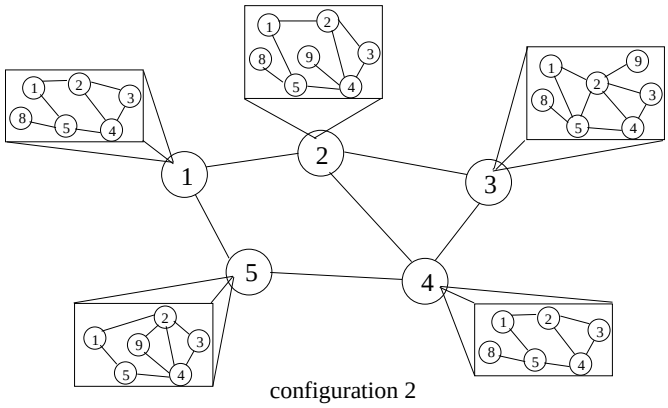
$G_1 = \{((1,2),0),((1,5),0),((2,3),1),((2,4),1),$
$((3,4),2),((4,5),1),((5,8),3)\}$

$G_2 = \{((1,2),0),((1,5),1),((2,3),0),((2,4),0),$
$((3,4),1),((4,5),1),((4,9),2),((5,8),3)\}$

$G_3 = \{((1,2),1),((1,5),3),((2,3),0),((2,4),1),$
$((2,5),2),((2,9),3),((3,4),0),((4,5),1),$
$((5,8),3)\}$

$G_4 = \{((1,2),1),((1,5),1),((2,3),1),((2,4),0),$
$((3,4),0),((4,5),0),((5,8),3)\}$

$G_5 = \{((1,2),1),((1,5),0),((2,3),2),((2,4),1),$
$((2,9),2),((3,4),1),((4,5),0),((4,9),2)\}$



configuration 2

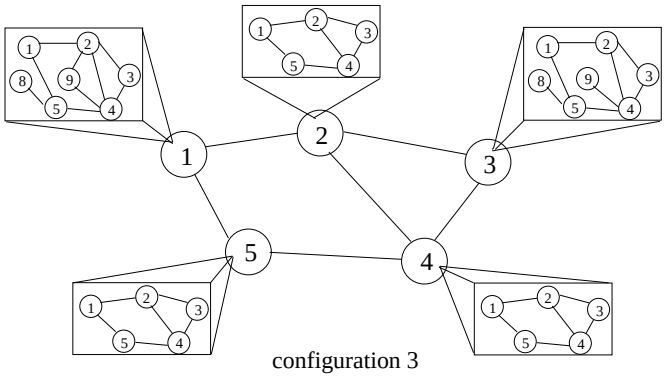
$G_1 = \{((1,2),0),((1,5),0),((2,3),1),((2,4),1),$
$((2,9),3),((3,4),2),((4,5),1),((4,9),3),$
$((5,8),4)\}$

$G_2 = \{((1,2),0),((1,5),1),((2,3),0),((2,4),0),$
$((3,4),1),((4,5),1)\}$

$G_3 = \{((1,2),1),((1,5),2),((2,3),0),((2,4),1),$
$((3,4),0),((4,5),1),((4,9),3),((5,8),4)\}$

$G_4 = \{((1,2),1),((1,5),1),((2,3),1),((2,4),0),$
$((3,4),0),((4,5),0)\}$

$G_5 = \{((1,2),1),((1,5),0),((2,3),2),((2,4),1),$
$((3,4),1),((4,5),0)\}$



configuration 3

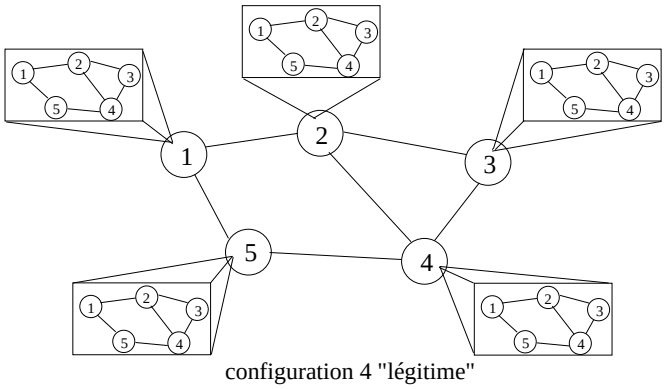
$G_1 = \{((1,2),0),((1,5),0),((2,3),1),((2,4),1),$
$((3,4),2),((4,5),1)\}$

$G_2 = \{((1,2),0),((1,5),1),((2,3),0),((2,4),0),$
$((3,4),1),((4,5),1)\}$

$G_3 = \{((1,2),1),((1,5),2),((2,3),0),((2,4),1),$
$((3,4),0),((4,5),1)\}$

$G_4 = \{((1,2),1),((1,5),1),((2,3),1),((2,4),0),$
$((3,4),0),((4,5),0)\}$

$G_5 = \{((1,2),1),((1,5),0),((2,3),2),((2,4),1),$
$((3,4),1),((4,5),0)\}$



configuration 4 "légitime"

FIG. 4.3 – Exemple d'exécution de l'algorithme proposé.

Lemme 1 *Soit p un processus ($p \in V$), après i rounds tel que $i \in \mathbb{N}$ et $i \geq 1$, G_p satisfait :* $\forall k \in \mathbb{N} \ / k < i : \exists u \in E \ / d(u,p) = k \Leftrightarrow \exists (u,k) \in G_p$

Preuve : Nous faisons une preuve par induction sur le nombre de rounds i.

INITIALISATION :

Durant le premier round, chaque processus p vérifie la garde de la règle R_0 ($G_p | p \neq \{((p,q),0) \ / q \in \mathcal{N}(p)\}$). Dans le cas où la garde est fausse, alors tous les liens incidents de p dans G_p possèdent des distances égales à 0 et se réfèrent correctement à tous les liens opérationnels incidents de p dans le système. Dans le cas contraire (la garde est vraie), le processus p exécute l'action de R_0. Ainsi, G_p contient un élément qui se réfère correctement à chaque lien opérationnel incident de p dans le système. Pendant la deuxième étape du premier round, la seule possibilité que G_p est changé est dûe à l'opération (Algorithm. 2, ligne 19). Cependant, cette opération ne change pas les liens incidents de p dans G_p. Maintenant, nous démontrons que sauf les liens incidents de p dans G_p qui ont des distances égales à 0 et tous les autres liens possèdent des distances supérieures ou égales à 1 après le premier round. Pour le faire, nous faisons une preuve par l'absurde. Supposons qu'après le premier round, G_p contient un élément $(u,0)$ où u n'est pas incidents de p. Cela signifie que cet élément appartient à T_p avant l'opération (Algorithme 2, ligne 19). Comme il n'existe aucune opération dans l'algorithme qui change les valeurs des champs des éléments de T_p et la seule opération qui ajoute des éléments à T_p est l'opération (Algorithme 2, ligne 13), donc il existe un processus q voisin de p tel que $(u,-1)$ appartenant à G_q avant ce round. Ce qui est en contradiction avec la définition du domaine du champ *dis* des éléments de G_q (l'ensemble des nombres naturels $\mathbb{N}$). Donc, tous les liens qui ont des distances égales à 0 sont incidents de p dans le système.

RECURRENCE :

Nous supposons que le lemme 1 est vrai pour i rounds et nous prouvons qu'il est vrai pour le $(i+1)^{eme}$ rounds.

L'existence d'un lien u dans le système tel que $d(u,p) = i$ signifie qu'il existe un processus q voisin de p tel que $d(u,q) = i-1$. Donc par hypothèse de récurrence, il existe un élément $(u, i-1)$ dans G_q après le i^{eme} round. Ainsi, dans le $(i+1)^{eme}$ round le processus p ajoute l'élément (u,i) à T_p (Algorithme 2, ligne 13). Cet élément n'est jamais supprimé de T_p dans l'opération (Algorithme 2, ligne 15) puisqu'il n'existe aucun processus s voisin de p, tel que le lien u appartient à G_s avec une distance inférieure à $i-1$ après le i^{eme} round (par hypothèse de récurrence). Il n'est jamais supprimé également de T_p dans l'opération

(Algorithme 2, ligne 18) puisque tous les liens dans le plus court chemin entre q et une extrémité de u sont dans T_p et possèdent des distances de 1 jusqu'à $i-1$. Car, ces liens sont dans G_q après le i^{eme} round avec des distances de 0 jusqu'à $i-2$ (par hypothèse de récurrence) et ils sont ajoutés à T_p dans le $(i+1)^{eme}$ round avec l'incrémentation de ses distances par 1 (Algorithme 2, ligne 13). Par conséquent, l'élément (u, i) appartient à G_p après le $(i+1)^{eme}$ round.

Maintenant, il reste à prouver que l'existence d'un élément (u, i) dans G_p après le $(i+1)^{eme}$ round implique que $d(u, p)$ est égale à i. Supposons qu'après le $(i+1)^{eme}$ round, G_p contient un élément (u, i). Cela signifie que cet élément appartient à T_p avant l'opération (Algorithme 2, ligne 19). Comme il n'existe aucune opération dans l'algorithme qui change les valeurs des champs des éléments de T_p, et la seule opération qui ajoute des éléments à T_p est l'opération (Algorithme 2, ligne 13), alors il existe un processus q voisin de p tel que $(u, i-1)$ appartient à G_q après le i^{eme} round, et donc $d(u, q)$ est égale à $i-1$ (par hypothèse de récurrence). Puisque l'élément (u, i) n'a pas été supprimé à l'opération (Algorithme 2, ligne 15), alors après le i^{eme} round il n'existe aucun processus s voisin de p tel que u avec une distance inférieure à $i-1$ appartient à G_s. Donc, la distance entre u et p dans le système est égale à i. □

Lemme 2 *Soit p un processus ($p \in V$), après $d+2$ rounds, G_p satisfait :*
$\forall x \in G_p : x.dis \leq d$

Preuve : Nous faisons une prouve par l'absurde. Supposons qu'après le $(d+2)^{eme}$ round, G_p contient un élément (u, k) où $k > d$. Cela signifie que cet élément appartient à T_p avant l'opération (Algorithme 2, ligne 19). Comme l'élément (u, k) n'a pas été supprimé à l'opération (Algorithme 2, ligne 18), alors T_p contient au moins un élément avec une distance l pour chaque $1 \leq l \leq k$. Ainsi, pour prouver ce lemme il suffit de trouver une contradiction pour le cas $l = d+1$. En effet, l'existence d'un élément $(u, d+1)$ dans T_p signifie qu'il existe un processus q voisin de p tel que l'élément (u, d) appartient à G_q après le $(d+1)^{eme}$ round. Par lemme 1, $d(u, q)$ est égale à d. Puisque l'élément $(u, d+1)$ n'a pas été supprimé à l'opération (Algorithme 2, ligne 15), alors après le $(d+1)^{eme}$ round il n'existe aucun processus s voisin de p tel que u avec une distance inférieure à d appartient à G_s. Donc, $d(u, p)$ est égale à $d+1$. Ce qui est en contradiction avec la définition du diamètre d'un graphe. □

D'après le lemme 1 et le lemme 2, chaque processus p possède, après $(d+2)$ rounds dans sa vision locale G_p, un élément pour chaque lien du système et tout élément de G_p se réfère correctement à un lien du système.

Corollaire 1 *La configuration attente après (d+2) rounds est légitime.*

4.2.4.2 Preuve de la correction

Lemme 3 *A partir d'une configuration légitime toute exécution de l'algorithme vérifie la spécification du maintien de la topologie.*

Preuve : Dans la seule configuration légitime pour un système donné, tous les processus possèdent la représentation correcte de la topologie du système. Il est clair que dans cette configuration les règles R_0 et R_1 pour chaque processus ne sont pas activables (les conditions $cond_1$, $cond_2$ et $cond_3$ sont toujours fausses). Donc, le système reste dans la configuration légitime. □

Théorème 1 *L'algorithme 2 est auto-stabilisant.*

Preuve : Conséquence directe du corollaire 1 et du lemme 3. □

4.2.5 Complexité en temps

D'après le lemme 1, le temps de convergence dans le pire cas de l'algorithme 2 est donc exactement $d + 2$ rounds. Comme le temps nécessaire pour transporter une information d'un côté du système à l'autre côté est non inférieur à $\Omega(d)$ [16], donc la complexité en temps de notre algorithme est optimal.

4.3 Transformateur général

Comme le font remarquer les auteurs de [38], le problème du maintien de la topologie permet de réduire tout problème non réactif, au coût de dupliquer l'état de tous les processus du système dans chaque processus du système, et ainsi permettre à tout processus de simuler localement un algorithme distribué, et de calculer l'état qu'il devrait avoir. Le coût de cette technique est principalement en espace mémoire : chaque processus maintient non seulement la topologie du système, mais de plus si b est le nombre de bits nécessaire

pour représenter l'état d'un processus, il faut $n * b$ bits par processus pour maintenir une copie de l'état des autres processus du système.

Notre transformateur consiste à utiliser l'algorithme décrit précédemment (Algorithme 2 pour résoudre le problème du maintien de la topologie. Au bout d'un temps fini, chaque processus possède une vision correcte du graphe de communication du système. Un algorithme séquentiel, déterministe de graphe peut alors être appliqué par tous les processus, pour aboutir à une configuration globale légitime. La composante correctrice qui corrige la topologie ainsi corrige le problème à résoudre. De cette manière, nous pouvons transposer tout algorithme séquentiel, déterministe et de graphe en algorithme réparti auto-stabilisant et dynamique qui résout le même problème.

La conception de ce transformateur est décrite sur la figure 4.4. Chaque processus p est constitué de trois module : *p.topologie*, *p.algo-squentiel* et *p.application*.

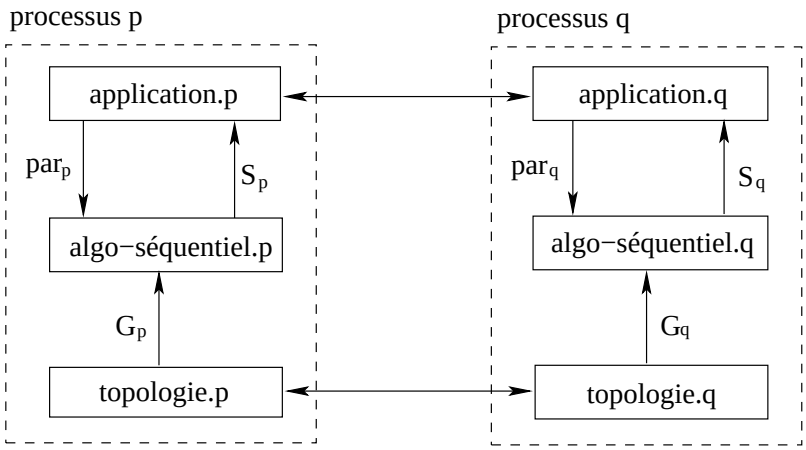


FIG. 4.4 – Transformateur général.

Pour accomplir sa tâche, le module *p.application* peut communiquer avec *q.application* de chaque processus q voisin de p. Il maintient une variable par_p qui doit contenir les paramètres nécessaires et corrects pour exécuter l'algorithme séquentiel dans le module *p.algo-squentiel*. Une manière pour assurer ça est de vérifier périodiquement la valeur de par_p et de la corriger si elle est incorrecte. L'implémentation du module *p.topologie* est l'algorithme 2. Ce module fournit une représentation de la topologie du système (G_p) au module *p.algo-squentiel*. Ce dernier utilise la représentation de la topologie et les paramètres (par_p) fournis par le module *p.application*, pour appliquer un algorithme séquentiel de graphe. Plus formellement, *p.algo-squentiel* exécute périodiquement l'opération suivante :

$S_p := SequentialAlgorithm(G_p, par_p)$;

Le résultat de cette opération (S_p) est partagé en lecture avec le module *p.application*.

4.4 Conclusion

Dans ce chapitre, nous avons proposé un algorithme réparti auto-stabilisant pour le maintien de la topologie dans les systèmes dynamique. Ensuite, nous avons montré comment utiliser cet algorithme pour transposer les algorithmes séquentiels et déterministes de graphe sur des systèmes répartis auto-stabilisants et dynamiques.

Conclusion et Perspectives

Quand un système réparti est sujet à des défaillances transitoires qui modifient arbitrairement les états des processus et le contenu des liens qui le constituent, il est très important de retrouver un comportement correct au bout d'un temps fini. L'auto-stabilisation est une technique offrant une telle garantie. Elle fournit une tolérance naturelle aux défaillances transitoires, et ce quels que soit leur nombre et leur nature. Le coût de cette propriété est l'existence d'une phase de stabilisation pendant laquelle les défaillances peuvent influencer le système, se propager et produire un comportement incorrect.

Afin de simplifier la conception et la preuve d'algorithmes auto-stabilisants qui sont des activités difficiles surtout si l'algorithme maintient beaucoup de variables, des modèles d'auto-stabilisation automatique (transformateurs) ont été proposés dans la littérature. Dans ce travail, nous avons donné une classification de ces modèles et étudié les modèles les plus intéressants. En générale, les modèles qui utilisent des opérations globales pour vérifier ou corriger le système sont lourds, alors que l'utilisation des opérations locales rend ces modèles inapplicables qu'à une classe limitée d'algorithmes répartis. D'un autre côté, le coût des modèles qui se base sur le maintien de la topologie est principalement en espace mémoire : chaque processus maintient une copie de l'état des autres processus du système.

Nous avons proposé un algorithme réparti auto-stabilisant pour le maintien de la topologie dans les systèmes dynamique. Cet algorithme est utilisé pour concevoir un transformateur qui transpose les algorithmes séquentiels en algorithmes répartis auto-stabilisants et dynamiques. Dans notre transformateur, l'algorithme initial (i.e., l'algorithme à transformer) doit être un algorithme déterministe et de graphe. De ce fait, les processus n'ont pas besoin de maintenir une copie de l'état des autres processus du système, mais seulement la topologie du système. Ce qui minimise l'espace mémoire utilisé par chaque processus. Cependant, cet espace reste toujours considérable puisqu'il est en fonction du nombre des processus et des liens dans le système.

Comme perspective directe de ce travail, il serait intéressant de calculer la complexité en espace de notre transformateur. Il serait aussi intéressant de rajouter des mécanismes pour palier aux limites de l'auto-stabilisation, tels que la K-stabilisation, le confinement de fautes, l'adaptabilité en temps et l'auto-stabilisation instantanée.

L'application de l'auto-stabilisation dans les réseaux de capteurs et l'auto-stabilisation en présence de défaillances byzantines sont également deux voies intéressantes à explorer.

Bibliographie

[1] H. Abu-Amara, B. Coan, S. Dolev, A. Kanevsky, and J. Welch. Self-stabilizing topology maintenance protocols for high-speed networks. *IEE/ACM Transactions on Networking*, 4(6) :902–912, 1996.

[2] D. Angluin. Local and global properties in networks of processors. In *Proceedings of the twelfth annual ACM symposium on Theory of computing*, pages 82–93, New York, NY,USA, 1980. ACM Press.

[3] A. Arora and M.G. Gouda. Distributed reset. *IEEE Transactions on Computers*, 43(9) :1026–1038, september 1994.

[4] H. Attiya and J. Welch. *Distributed computing : fundamentals, simulations, and advanced topics*. McGraw-Hill Publishing Company, 1998.

[5] B. Awerbuch and Y. Mansour. An efficient topology update protocol for dynamic networks. In *Proceedings of the 6th International Workshop on Distributed Algorithms*, pages 185–202, 1992.

[6] B. Awerbuch, B. Patt-Shamir, and G. Varghese. Self-stabilization by local checking and correction. In *Proceedings of the 32nd IEEE Symposium on Foundations of Computer Science*, October 1991.

[7] B. Awerbuch, B. Patt-Shamir, G. Varghese, and S. Dolev. Self-stabilization by local checking and global reset. In *Proceedings of the 8th International Workshop on Distributed Algorithms*, pages 326–339. Springer-Verlag, October 1994.

[8] J. Beauquier, S. Delaet, S. Dolev, and S. Tixeuil. Transient fault detectors. In *Proceedings of the 12th International Symposium on Distributed Computing (DISC'1998)*, pages 62–74. Springer-Verlag, 1998.

[9] J. Beauquier, C. Genolini, and S. Kutten. k-stabilization of reactive tasks. In *PODC'98 17th Annual ACM Symposium on Principles of Distributed Computing*, page 318, 1998.

[10] J.C. Browne, A.E. Emerson, M.G. Gouda, D.Miranker, A. Mok, and L.E. Rosier. Bounded-time fault-tolerant rule-based systems. *Telematics and Informatics*, 7 :441–454, 1990.

[11] O. Brukman and S. Dolev. Self-stabilizing autonomic recoverer for eventual byzantine software. In *Proceedings of the IEEE International Conference on Software*, 2003.

[12] J.E. Burns, M.G. Gouda, and R.E. Miller. Stabilization and pseudo-stabilization. *Distributed Computing*, 7(1) :35–42, 1993.

[13] C.M. Chandy and L. Lamport. Distributed snapshots : determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1) :63–75, 1985.

[14] S. Devismes. *Quelques contributions à la stabilisation instantanée*. PhD thesis, Université de Picardie Jules Verne, Décembre 2006.

[15] E.W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Communications of the ACM*, 17(11) :643–644, 1974.

[16] S. Dolev. Optimal time self-stabilization in dynamic systems. In *Proceedings of the 7th International Workshop on Distributed Algorithms*, pages 129–144, 1993.

[17] S. Dolev. *Self-Stabilization*. The MIT Press, Cambridge, MA, 2000.

[18] S. Dolev and T. Herman. Superstabilizing protocols for dynamic distributed systems. *Chicago Journal of Theoretical Computer Science*, 3(4), 1997.

[19] S. Dolev, A. Israeli, and S. Moran. Self-stabilization of dynamic systems assuming only read/write atomicity. *Distrib. Comput.*, 7(1) :3–16, 1993.

[20] S. Dolev, A. Israeli, and S. Moran. Resource bounds for self-stabilizing message-driven protocols. *SIAM on computing*, 26(1) :273–290, 1997.

[21] M.J. Fischer, N.A. Lynch, and M.S. Patterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2) :374–382, April 1985.

[22] F.C. Gartner. A survey of self-stabilizing spanning-tree construction algorithms. Technical report, Swiss Federal Institute of Technology (EPFL), School of Computer and Communication Sciences, Distributed Programming Laboratory, CH-1015 Lausanne, Switzerland, June 2003.

[23] S. Ghosh, A. Gupta, T. Herman, and S.V. Pemmaraju. Fault-containing self-stabilizing algorithms. In *PODC'96 15th Annual ACM Symposium on Principles of Distributed Computing*, pages 45–54, 1996.

[24] S. Ghosh and X. He. Scalable self-stabilization. *Journal of Parallel and Distributed Computing*, 62 :945–960, 2002.

[25] M.G. Gouda. The triumph and tribulation of system stabilization. In *Proceedings of the 9th International Workshop on Distributed Algorithms (WDAG'1995)*, pages 1–18, London UK, 1995. Springer-Verlag.

[26] M.G. Gouda and N. Multari. Stabilizing communication protocols. *IEEE Transactions on Computers*, 40(4) :448–458, 1991.

[27] M.V. Gradinariu. *Modélisation, vérification et raffinement des algorithmes auo-stabilisants*. PhD thesis, Laboratoire de Recherche en Informatique, Université Paris XI, 2000.

[28] J. El Haddad. *Auto-stabilisation : gestion de robots mobiles et confinement de fautes*. PhD thesis, Université Paris Dauphine, Décemre 2004.

[29] B. Hamid and M. Mosbah. An automatis approach to self-stabilization. In *Proceedings of the Sixth IEEE Conference on Software Engineering*, pages 1–6, 2005.

[30] T. Hérault. *Correction de défaillances transitoires dans les systèmes auto-stabilisants*. PhD thesis, Université Paris XI, Orsay, France, Mai 2003.

[31] T. Herman. Probabilistic self-stabilization. *Information Processing Letters*, 35 :63–67, 1990.

[32] T. Herman. Phase clocks for transient fault repair. *IEEE Trans. Parallel Distrib. Syst.*, 11(10) :1048–1057, 2000.

[33] T. Herman and S. Ghosh. Stabilizing phase-clocks. *Process. Lett.*, 54(5) :259–265, 1995.

[34] C. Johnen, F. Petit, and S. Tixeuil. Auto-stabilisation et protocoles réseaux. *Technique et Science Informatiques*, 23(8) :1027–1056, 2004.

[35] S. Kamei and H. Kakugawa. A self-stabilizing approximation algorithm for the distributed minimum k-domination. *IEICE Transactions on Fundamentals of Electronics Communications and Computer Sciences E Series A*, 88(5) :1109–1116, 2005.

[36] S. Katz and K.J. Perry. Self-stabilizing extensions for message-passing systems. In *Proceedings of the 10th ACM Symposium on Principles of Distributed Computing*, August 1990.

[37] J.L.W. Kessels. An exercise in proving self-stabilization with a variant function. *Information Processing Letters*, 29 :39–42, 1988.

[38] S. Kutten and B. Patt-Shamir. Time-adaptive self-stabilization. In *PODC'97 16th Annual ACM Symposium on Principles of Distributed Computing*, pages 149–158, August 1997.

[39] S. Kutten and B. Patt-Shamir. Time-adaptive self-stabilization. *Theoretical Computer Science*, 220(1) :93–111, 1999.

[40] L. Lamport. *Solved problems, unsolved problems and non-problems in concurrency*. PhD thesis, Proceedings of the 3rd Symposium on Principles of Distributed Computing, 1984.

[41] L. Lamport, R. Shostak, and M. Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3) :382–401, July 1982.

[42] W. Leal and A. Arora. Scalable self-stabilization via composition. In *Proceedings of the 24th International Conference on Distributed Computing Systems (ICDCS'04)*, 2004.

[43] N. Lynch. *Distributed algorithms*. Morgan Kaufmann, 1996.

[44] F. Magniette. *Preuves d'algorithmes auto-stabilisants*. PhD thesis, Université Paris XI, Orsay, France, Juin 2002.

[45] E. Marsden. *Caractérisation de la sûreté de fonctionnement de systèmes à base d'Intergiciel*. PhD thesis, Institut National Polytechnique de Toulouse, Février 2004.

[46] L. Pilard. *Observer la stabilisation*. PhD thesis, Université Paris XI, Orsay, France, Décembre 2005.

[47] S. Shukla, D. Rosenkrantz, and S. Ravi. Developing self-stabilizing coloring algorithms via systematic randomization. In *Proceedings of the International Workshop on Parallel Processing*, pages 668–673, Bangalore, India, 1994. Tata-McGrawhill, New Delhi.

[48] G. Tel. *Introduction to distributed algorithms*. Cambridge University Press, November 1994.

[49] S. Tixeuil. *Auto-stabilisation efficace*. PhD thesis, Université Paris XI, Orsay, France, 2000.

[50] S. Tixeuil. Vers l'auto-stabilisation des systèmes à grande échelle. Technical report, RR LRI 1452, LRI, Université Paris XI, Orsay, France, Mai 2006.

[51] G. Varghese. *Self-stabilization by Local Checking and Correction*. PhD thesis, Massachussetts Institute of Technology, October 1993.

[52] G. Varghese, A. Arora, and M.G. Gouda. Self-stabilization by tree correction. *Chicago Journal of Theoretical Computer Science*, 3(3) :1–32, November 1997.

[53] F. Vernier. *Algorithmique itérative pour l'équilibrage de charge dans les réseaux dynamiques*. PhD thesis, Université de Franche-Comté, Décembre 2004.

Transposition Automatique des Algorithmes sur des Systèmes Distribués et Dynamiques

Résumé

Le concept d'auto-stabilisation a été introduit en 1974 par E.G. Dijkstra. Un système réparti est défini comme auto-stabilisant s'il se comporte selon sa spécification au bout d'un temps fini quelque soit sa configuration initiale. Par cette propriété, les algorithmes auto-stabilisants tolèrent tout genre et tout nombre fini de défaillances transitoires. Puisque la conception et la preuve d'algorithmes auto-stabilisants sont généralement des tâches complexes, ainsi que l'existence d'un nombre important d'algorithmes répartis mais qui ne possèdent pas la propriété d'auto-stabilisation, certains modèles d'auto-stabilisation automatique (appelés aussi transformateurs) ont été proposés dans la littérature afin d'ajouter la propriété d'auto-stabilisation aux algorithmes répartis. Dans ce travail, nous avons donné une classification de ces modèles et étudié les modèles les plus intéressants. Nous avons proposé aussi un algorithme réparti auto-stabilisant pour le maintien de la topologie dans les systèmes dynamiques. Cet algorithme est utilisé pour concevoir un transformateur qui transpose une classe d'algorithmes séquentiels en algorithmes répartis auto-stabilisants et dynamiques.

Mots clés **:** algorithmes distribués, communication, systèmes distribués, systèmes dynamiques, réseaux.

Abstract

The concept of self-stabilization was introduced in 1974 by E.G. Dijkstra. A distributed system is defined to be self-stabilizing if regardless of the initial state, it behaves according to its specification after a finit time. By this property, the self-stabilizing algorithms tolerate any kind and any finished number of transitory failures. Since the design and the proof of self-stabilizing algorithms are generally a complex tasks, as well as the existence of an important number of distributed algorithms but which are not self-stabilizing, some models of automatic self-stabilization (called also transformers) were proposed in the literature in order to add the property of self-stabilization to the distributed algorithms. In this work, we gave a classification of these models and studied the most interesting models. We also proposed a distributed self-stabilizing algorithm for topology maintenance in dynamic systems. This algorithm is used to design a transformer which transposes a class of sequential algorithms in distributed self-stabilizing and dynamic algorithms.

Keywords **:** distributed algorithms, communication, distributed systems , dynamic systems, networks.