Mécanismes pour la cohérence, l'atomicité et les communications au niveau des clusters : application au clustering hiérarchique distribué adaptatif

François Avril

UNIVERSITÉ DE VERSAILLES ST-QUENTIN-EN-YVELINES

École Doctorale Science et Technologie de Versailles

NNT : 2015SACLV034

Mécanismes pour la cohérence, l'atomicité et les communications au niveau des clusters : application au clustering hiérarchique distribué adaptatif

THÈSE

présentée et soutenue publiquement le 29 septembre 2015

pour l'obtention du

Doctorat de l'Université de Versailles Saint-Quentin-en-Yvelines

(mention informatique)

par

François Avril

Composition du jury

Président :	Jean-Christophe Lapayre	Professeur à l'Université de Franche-Comté
Rapporteurs :	Maria Potop Butucaru	Professeur à l'Université de Paris 6
	Colette Johnen	Professeur à l'Université de Bordeaux
Examinateur :	Johanne Cohen	Chargée de recherche à l'Université de Paris-Sud
Directeur :	Alain Bui	Professeur à l'Université de Versailles Saint-Quentin-en-Yvelines
Encadrant :	Devan Sohier	Maître de conférence à l'Université de Versailles Saint-Quentin-en-Yvelines

Laboratoire PRiSM - Équipe CaRO - Université de Versailles Saint-Quentin-en-Yvelines

Remerciements

Mener à bien une thèse est une tâche qui dépend d'un grand nombre de facteurs, et en particulier d'un grand nombre d'interactions humaines. Je tiens à remercier mon directeur de thèse Alain Bui de m'avoir accepté comme doctorant, d'autant plus sur un status qui laissait dubitatif son entourage professionnel. Ces efforts répétés pour me détourner du monde merveilleux des mathématiques n'ont pas totalement abouti, mais ne sont clairement pas un échec.

J'ai une pensée particulière pour mon encadrant, Devan Sohier. Je te remercie pour ta disponibilité et pour tes conseils avisés. Travailler avec toi fut très stimulant. Ton interdiction formelle d'utiliser le mot "trivial" a également eu un impact positif sur ma clarté, et je t'en remercie.

Je remercie Maria Potop Butucaru et Colette Johnen pour avoir accepté de rapporter ma thèse, et pour l'intérêt qu'elles ont portés à mes travaux. Je remercie également Jean-Christophe Lapayre et Johanne Cohen pour avoir accepté de participer à ce jury.

Je remercie tout particulièrement mes amis Yann Hermans, sans qui cette thèse n'aurait simplement pas commencé et Simon Clavière, qui m'a appris à penser "distribué". Travailler avec vous est tout simplement génial.

Je remercie également tous les proches qui m'ont apportés encouragements et soutien durant les moments de doutes, et qui ont su me changer les idées lorsque c'était nécessaire.

Ce parcours scientifique n'aurait pas été possible sans l'étincelle de la curiosité que mes parents m'ont transmis. Merci d'avoir tout fait pour que je me pose des questions, de m'avoir apporté un soutien sans faille, et de ne vous être jamais lassé de la question la plus fondamentale : "Pourquoi ?".

Enfin, ces remerciements ne sauraient être complets sans évoquer celle qui partage ma vie, Emmanuelle. Il y a tellement de choses pour lesquelles je te suis reconnaissant qu'aucun écrit ne saurait les contenir toutes. Merci de m'avoir suivi, soutenu, encouragé, d'avoir pris plus que ta part en tâches ménagères et malgré tout d'avoir réussi à me supporter. Cette thèse n'aurait pas vu le jour sans ton concours.

Résumé

Nous nous intéressons dans cette thèse à l'organisation des systèmes distribués dynamiques de grande taille : ensembles de machines capables de communiquer entre elles et pouvant à tout instant se connecter ou se déconnecter. Nous proposons de partitionner le système en groupes connexes, appelés clusters. Afin d'organiser des réseaux de grande taille, nous construisons une structure hiérarchique imbriquée dans laquelle les clusters d'un niveau sont regroupés au sein de clusters du niveau supérieur. Pour mener à bien ce processus, nous mettons en place des mécanismes permettant aux clusters d'être les nœuds d'un nouveau système distribué exécutant l'algorithme de notre choix. Cela nécessite en particulier des mécanismes assurant la cohérence de comportement pour le niveau supérieur au sein de chaque cluster. En permettant aux clusters de constituer un nouveau système distribué exécutant notre algorithme de clustering, nous construisons une hiérarchie de clusters par une approche ascendante. Nous démontrons cet algorithme en définissant formellement le système distribué des clusters, et en démontrant que chaque exécution de notre algorithme induit sur ce système une exécution de l'algorithme de niveau supérieur. Cela nous permet, en particulier, de démontrer par récurrence que nous calculons bien un clustering hiérarchique imbriqué. Enfin, nous appliquons cette démarche à la résolution des collisions dans les réseaux de capteurs. Pour éviter ce phénomène, nous proposons de calculer un clustering adapté du système, qui nous permet de calculer un planning organisant les communications au sein du réseau et garantissant que deux messages ne seront jamais émis simultanément dans la portée de communication de l'un des capteurs.

Abstract

To manage and handle large scale distributed dynamic distributed systems, constituted by communicating devices that can connect or disconnect at any time, we propose to compute connected subgraphs of the system, called clusters. We propose to compute a hierarchical structure, in which clusters of a level are grouped into clusters of the higher level. To achieve this goal, we introduce mechanisms that allow clusters to be the nodes of a distinct distributed system, that executes an algorithm. In particular, we need mechanisms to maintain the coherence of the behavior among the nodes of a cluster regarding the higher level. By allowing clusters to be nodes of a distributed system that executes a clustering algorithm, we compute a nested hierarchical clustering by a bottom-up approach. We formally define the distributed system of clusters, and prove that any execution of our algorithm induces an execution of the higher level algorithm on the distributed system of clusters. Then, we prove by induction that our algorithm computes a nested hierarchical clustering of the system. Last, we use this approach to solve a problem that appears in sensor networks : collision. To avoid collisions, we propose to compute a clustering of the system. This clustering is then used to compute a communication schedule in which two messages cannot be sent at the same time in the range of a sensor.

Sommaire

Chapitre 1

Introduction

1.1 Mise en contexte

Nous nous intéressons dans cette thèse à l'organisation des systèmes distribués de grande taille. Un système distribué est un ensemble de machines, appelées nœuds du système, qui exécutent un algorithme et qui sont dotées d'une interface de communication. Chaque nœud peut communiquer avec une partie des nœuds du système, ce qui définit une relation de voisinage, représentée par un graphe de communication.

Depuis quelques années, nous côtoyons un nombre toujours croissant d'appareils dotés de la capacité de communiquer sans fil. Ces appareils mobiles nous forcent à considérer des systèmes distribués dynamiques, c'est-à-dire des systèmes dont le graphe de communication peut changer au cours du temps. À tout moment un nœud peut quitter le système distribué, de nouveaux nœuds peuvent le rejoindre, et des liens de communication peuvent-être ajoutés ou coupés. Dans des systèmes de plus en plus grands, ces modifications du graphe de communication (appelées des changements topologiques) doivent être prises en compte et ne peuvent plus être considérées comme des évènements rares. Le caractère dynamique rend ainsi les systèmes distribués compliqués à gérer.

Une autre difficulté des systèmes distribués dynamiques est qu'ils ont un comportement non déterministe, ce qui va à l'encontre des habitudes de programmation. Un ordinateur possède en effet un comportement déterministe : à partir d'un état donné, en suivant ligne par ligne l'algorithme exécuté, il est possible de prédire les états futurs de la machine. Dans un système distribué, le comportement de chaque appareil dépend à la fois de son état, mais aussi de ses interactions avec les autres appareils du système. Le temps nécessaire pour effectuer une communication pouvant être impacté par un grand nombre de facteurs (fonctionnement du protocole de communication, perturbation ou saturation du média de communication, ...), l'évolution future du système n'est pas entièrement prévisible. De plus, des modifications de la topologie du réseau peuvent survenir à tout moment, ce qui renforce le caractère non déterministe du système.

Pour organiser ces systèmes, nous proposons de regrouper plusieurs appareils au sein d'entités appelées *clusters*. Un cluster est ainsi un ensemble de plusieurs appareils formant une partie connexe du graphe de communication. Cette proximité dans le graphe de communication permet aux appareils d'un même cluster de communiquer facilement. Les clusters forment ainsi un nouveau système, plus petit que le système distribué initial, qui sera ainsi plus facile à organiser et à gérer.

Au delà des difficultés rencontrées pour calculer des clusters dans un système dynamique, il faut aussi pouvoir utiliser ce clustering. Nous construisons en effet les clusters comme les

composants d'un système distribué, plus simple que le système initial. Cependant, pour que ce clustering soit utilisable, il faut pouvoir gérer indépendamment les différents clusters. Nous proposons ainsi de doter les clusters de la capacité d'exécuter un algorithme. Nous proposons également de mettre en place une couche d'abstraction permettant au clustering d'émuler un système distribué : les clusters sont les nœuds d'un nouveau système distribué distinct du système initial.

1.2 État de l'art

1.2.1 Clustering

La problématique du clustering est liée au problème du routage. Lorsque les réseaux informatiques ont commencé à grandir, il est devenu impossible de maintenir sur chaque machine une table de routage de l'ensemble du réseau. Dans [BE81] et [McQ74], les auteurs proposent de diviser le réseau en sous-ensembles, et de déléguer le routage interne à un seul nœud choisi dans ce sous-ensemble. L'un des plus anciens algorithmes de clustering distribué, nommée LCA, est présenté dans [BE81]. Cet algorithme a été conçu en premier lieu pour des réseaux statiques de moins de 100 nœuds. Dans cet algorithme, les clusters sont construits par l'élection de leaders, appelés clusterheads, qui forment des clusters en recrutant leurs voisins. Pour élire le clusterhead, chaque nœud envoie son identifiant à tous ses voisins ; lorsqu'un nœud a le plus petit identifiant parmi ses voisins, il se déclare clusterhead et en informe ses voisins. Lorsqu'un nœud a un voisin qui se déclare clusterhead, il devient nœud ordinaire, s'attache à lui, et en informe ses voisins. Lorsqu'un nœud détecte que tous ses voisins de plus petit identifiant sont des nœuds ordinaires, il se déclare clusterhead et en informe ses voisins. Les nœuds ayant deux clusterheads dans leur voisinage deviennent des nœuds passerelles, utilisés pour le routage inter-clusters. Les auteurs proposent une amélioration de l'algorithme dans [EWB87], qui diminue le nombre de clusterheads créés par l'algorithme.

La majorité des algorithmes de clustering qui ont été écrits par la suite utilisent de façon similaire la notion de clusterheads, qui sont élus de diverses façons.

Ainsi, dans [GTCT95], en se basant sur [Par94], les auteurs présentent un algorithme de clustering ou les clusterheads sont choisis en fonction de leur degré. Dans un premier temps, les nœuds s'échangent leurs degrés. Ensuite, comme dans l'algorithme [EWB87], les nœuds ayant le plus grand degré sont élus clusterheads, et recrutent leur voisins. Ensuite, les nœuds de plus grands degrés parmi les nœuds non clusterisés deviennent clusterheads, et recrutent leurs voisins. Ce processus se poursuit jusqu'à ce que tous les nœuds soient clusterisés.

D'après [GTCT95], la mobilité dans les réseaux sans fil a dans un premier temps été prise en compte en relançant régulièrement la phase de calcul des clusters, en espérant retrouver les mêmes clusterheads. La méthode de choix des clusterheads impacte la stabilité des clusters, comme le montre une étude comparative également présente dans [GTCT95].

L'auteur de l'algorithme DCA, présenté dans [Bas99a], généralise cette heuristique dans le choix des clusterheads, en introduisant un poids abstrait calculé par chaque nœud. Il suppose que chaque nœud du réseau possède un poids distinct. Les nœuds s'échangent dans un premier temps ce poids, puis s'ensuit une étape de calcul des clusters, similaire aux algorithmes précédents, mais choisissant les nœuds de plus hauts poids comme clusterheads. L'auteur propose notamment de prendre pour poids l'inverse de la vitesse des nœuds, permettant d'avoir des clusterheads plus stables lors des reclusterisations.

L'algorithme DCA a été adapté aux réseaux anonymes dans [MCS10], où les nœuds commencent par s'attribuer aléatoirement des identifiants distincts entre 1 et n^3. Ensuite, les clus-

terheads sont élus en considérant la concaténation du poids et de cet identifiant comme critère de choix.

Tous ces algorithmes supposent, pour fonctionner, un certain degré de synchronisme entre les nœuds. En effet, la phase d'élection des clusterheads nécessite que les temps de communication soient bornés. Ces algorithmes supposent également qu'aucun changement topologique ne survienne durant la phase de clusterisation. Cette dernière contrainte est levée dans [Bas99b], où l'algorithme DMAC autorise les changements topologiques durant cette phase.

Par la suite, divers poids ont été étudiés, pour donner une plus grande stabilité aux clusters en cas de changement topologique. [BKL01] utilise les puissances de réception pour mesurer les vitesses relatives des nœuds. Le poids utilisé est ensuite une moyenne des vitesses relatives entre le nœud et ses voisins. L'algorithme sélectionne ensuite comme clusterheads les nœuds ayant la plus petite mobilité relativement à leurs voisins. De cette façon, les clusterheads ont moins de chances de passer hors de portée des noeuds qu'ils administrent.

[CDT01] utilise une combinaison de critères, incluant le degré des nœuds, la distance entre ceux-ci, leurs vitesses relatives ainsi que le temps passé en tant que clusterhead.

[YC03] force deux clusterheads a être à au moins 3 sauts d'écarts. Quand un nœud devient clusterhead, il recrute ses voisins. Ses voisins à deux sauts deviennent des nœuds *unspecified* qui ne peuvent devenir clusterheads. Si, après un mouvement, deux clusterheads deviennent voisins, celui de plus grand identifiant abandonne son rôle et devient nœud membre. Cela permet, tant que les changements topologiques sont assez espacés, d'éviter les reclusterisations en chaîne du système.

La croissance des réseaux a amené à rechercher un compromis entre la taille des clusters et leur nombre. En effet, dans tous les algorithmes précédents, les clusters sont de rayon 1, c'est à dire formés uniquement des voisins du clusterhead. Divers algorithmes construisent des clusters plus grands en recrutant les nœuds ordinaires dans un voisinage à plusieurs sauts du clusterhead, comme par exemple [LG97], [APVH00] ou [LC00]. Dans [KVCP97], des clusters de rayon k sont construits sans aucun clusterhead. Cependant, cet algorithme repose sur une connaissance par chaque nœud de son voisinage à k sauts, qui nécessite un grand nombre de messages pour être maintenue à jour.

Une autre façon de gérer la dynamicité des systèmes distribués est de considérer les changements topologiques comme des fautes transitoires. Dijkstra introduit dans [Dij74] le concept d'auto-stabilisation pour répondre à cette situation. Après qu'une faute se soit produite dans un système exécutant un algorithme auto-stabilisant, celui-ci peut se trouver dans un état illégal. Cependant, l'auto-stabilisation garantit que le système revient en un temps fini à un état permettant l'exécution de l'algorithme, et s'y maintient en l'absence de nouvelle faute.

On trouve plusieurs algorithmes de clustering auto-stabilisants. Par exemple, [JN06] est une version auto-stabilisante de l'algorithme DMAC présenté dans [Bas99b]. Dans [JN09], une version robuste de l'algorithme est présentée, et dans [JM10], une version robuste et auto-stabilisante. Dans [MFLT05], les auteurs présentent un algorithme de clustering auto-stabilisant, construisant les clusters sur un critère de densité similaire à celui de [MBF04].

Les algorithmes précédents sont tous basés sur l'élection d'un clusterhead, et les clusters sont construits sur un critère de diamètre. Ces algorithmes n'ont aucun contrôle sur la taille des clusters, qui sont formés de nœuds connectés au clusterhead. Les clusters peuvent être constitués d'un seul nœud, le clusterhead, ou d'un très grand nombre de nœuds selon le nombre de voisins du clusterhead.

Il est également possible de construire des clusters sur un critère de taille. Dans [BKS09], les auteurs proposent un tel algorithme. Chaque cluster est construit par un message particulier, appelé jeton, qui suit une marche aléatoire. Les nœuds sont recrutés un par un dans un clusters,

permettant ainsi de construire des clusters sur un critère de taille. Cet algorithme garantit de plus que le système ne comportera aucun cluster de taille 1.

Les marches aléatoires ont l'avantage de ne pas dépendre de la topologie du réseau, et donc de continuer à circuler correctement en cas de changement topologique. Grâce à cela, le clustering construit dans [BKS09] s'adapte aux changements topologiques : lorsqu'un changement topologique affecte un cluster (tous ne le font pas), celui-ci peut être réparé au cours de la marche aléatoire du jeton.

Contrairement aux algorithmes basés sur l'élection de clusterheads, les reclusterisations en chaîne ne peuvent se produire en cas de changement topologique. Seule une petite partie du réseau peut être affectée : le cluster où le changement topologique a lieu et éventuellement les clusters adjacents.

Pour toutes ces raisons, c'est cet algorithme que nous utilisons pour construire nos clusters.

1.2.2 Clustering hiérarchique

Les algorithmes de clustering présentés à la section précédente suffisent amplement à organiser les réseaux de petite taille. D'après les auteurs, l'algorithme [BE81] permet d'assurer efficacement le routage dans des réseaux d'une centaine de nœuds. Cependant, la croissance de plus en plus rapide des systèmes distribués nécessite la mise en place de méthodes évoluées pour organiser les systèmes distribués de grande taille. [TR98] construit ainsi un arbre couvrant des clusters afin de mettre en place un routage inter-clusters.

Cette méthode s'avère cependant insuffisante sur des systèmes de grande taille. Sur des systèmes comprenant des centaines de milliers de machines, les clusters calculés sont en nombre trop important, ou contiennent un nombre trop élevé de machines pour être utilisables.

Pour organiser de tels systèmes, il est possible de créer une hiérarchie imbriquée, dans laquelle les clusters du niveau k constituent les nœuds du niveau $k+1$, auxquels un algorithme de clustering est appliqué. Cette idée apparait dans [McQ74] à des fins de routage.

La littérature contient divers algorithmes construisant une telle hiérarchie par une approche ascendante. Dans un premier temps, des clusters sont construits d'une manière similaire aux algorithmes évoqués dans la section 1.2.1. Ensuite, l'algorithme de clustering est relancé en impliquant uniquement les clusterheads, comme dans [SM02], [Bea03] ou [YC09].

La dynamicité des réseaux est une fois encore une source d'instabilité de la structure. L'impact des changements topologiques sur la structure est encore plus important que dans un algorithme de clustering à un niveau. Un changement topologique peut en effet entrainer une reclusterisation en chaîne comme dans les algorithmes précédents, et cette reclusterisation peut impacter les niveaux logiques supérieurs de la structure.

[DT09] présente ainsi un algorithme de clustering hiérarchique auto-stabilisant.

Dans [SSS06], les clusters sont construits en regroupant des nœuds dont la mobilité relative est au dessous d'un seuil défini en paramètre de l'algorithme. Les clusters peuvent ensuite être regroupés, toujours sur un critère de mobilité relative. Ce critère de mobilité vise à rendre plus stable la structure hiérarchique construite.

Dans ces algorithmes, la structure hiérarchique est construite par une approche ascendante. Les clusters sont calculés par un algorithme de clustering basé sur le diamètre. Ensuite, le même algorithme de clustering est itéré sur les clusters.

Pour construire une structure plus équilibrée, il est possible de faire une structure à l'aide d'un algorithme de clustering orienté taille. Les algorithmes de clustering orientés taille présent dans [BCS12] ou [BBPS10] construisent une hiérarchie de clusters par une approche descendante.

Dans ces algorithmes, un premier cluster est construit par marche aléatoire. Quand il devient plus grand qu'une taille donnée en paramètre de l'algorithme, il est divisé en deux clusters, contenus dans le premier. Ces deux clusters continuent de grandir, jusqu'à être divisés à leur tour. Il en résulte une hiérarchie imbriquée. Cet algorithme, utilisant les marches aléatoires, est de plus résistant aux changements topologiques.

Chaque cluster de cette structure est découpé en deux clusters au niveau inférieur. Cependant, cette structure n'est pas toujours équilibrée. En effet, lorsqu'un cluster est divisé en deux sous cluster, l'un de ceux-ci peut grandir et être divisé à nouveau, tandis que l'autre ne change pas. L'arbre représentant la structure est un arbre binaire, mais l'algorithme ne garantit aucun équilibre entre ces sous arbres.

Nous proposons de construire une hiérarchie de clusters orientés tailles par une approche ascendante. Pour cela, nous créons des clusters d'une taille fixée, puis nous regroupons ceux-ci au sein de clusters de niveau supérieur, toujours de taille fixée. Cette façon de procéder nous garantit que la structure hiérarchique sera équilibrée.

Pour rendre cette structure hiérarchique résistante aux changements topologiques, et pour limiter l'impact de ceux-ci dans la structure, les nœuds (ou les clusters dans les niveaux supérieurs) seront regroupés par une méthode similaire à celle exposée dans [BKS09].

1.2.3 Nœud virtuel

Pour construire une hiérarchie de clusters par approche ascendante, il faut être capable de regrouper les clusters via un algorithme de clustering. La difficulté ici est qu'un cluster, constitué de plusieurs appareils, doit se comporter comme une entité unique au niveau supérieur.

Lorsque les clusters sont construits à l'aide de clusterheads, ceux-ci représentent leur cluster au niveau supérieur. Un mécanisme de routage est mis en place entre les clusterheads, permettant aux clusterheads de deux clusters voisins de communiquer. L'algorithme de clustering est ensuite relancé en impliquant uniquement les clusterheads, qui communiquent entre eux via ce mécanisme de routage.

Nos clusters étant construits sans clusterhead, nous ne pouvons appliquer cette méthode. Nous implémentons donc les primitives nécessaires pour que les clusters se comportent comme les nœuds d'un nouveau système distribué. Nous donnons aux clusters la capacité d'exécuter un algorithme, ainsi que la capacité de communiquer entre eux. Contrairement aux algorithmes de clustering hiérarchique présentés précédemment, nos clusters ont ainsi la capacité d'exécuter n'importe quel algorithme distribué.

De précédents travaux ont été conduits sur l'émulation de nœuds virtuels, mais dans un contexte très différent. L'écriture d'algorithmes distribués pour les systèmes dynamiques est compliquée, notamment parce que les mouvements des nœuds dans le système ne sont pas connus à l'avance. Si les mouvements des nœuds étaient connus, ou même seulement ceux d'une partie des nœuds, certains algorithmes pourraient être simplifiés. Par exemple, un nœud explorant périodiquement l'intégralité du réseau pourrait assurer la distribution des messages, et simplifier les méthodes de routage.

C'est en partant de ce constat que [DGL+04] propose de construire des nœuds virtuels, dont les mouvements dans le réseau sont contrôlés. Ces nœuds virtuels communiquent avec les machines du réseau, et participe à l'exécution d'un algorithme distribué. Les auteurs proposent de s'appuyer sur ces nœuds virtuels afin de simplifier la conception des algorithmes distribués exécuté par le système.

Dans cet algorithme, tous les nœuds disposent d'un système de localisation, d'une horloge, et ont une portée de communication r. À chaque instant, la position prévue du nœud virtuel est

connue par l'ensemble du système. Tous les nœuds dans un disque de rayon $D < 2r$ autour d'un nœud virtuel participent à son émulation. Toute la difficulté est de maintenir la cohérence entre les machines participant à l'émulation.

Afin de maintenir cette cohérence, plusieurs mécanismes sont mis en place. Un service de diffusion locale des messages est mis en place, imposant un délai aux messages envoyés. Les messages envoyés par ce service sont également ordonnés par un mécanisme similaire aux horloges de Lamport ([Lam78]). Ce service transmet ensuite ces messages localement, transmettant une seule fois les messages multiples qu'ont pu envoyer les divers nœuds participant à la simulation. Lorsqu'un nœud arrive dans ce disque, il demande l'état des variables du nœud virtuel, puis surveille les messages adressés au nœud virtuel. Lorsqu'il reçoit la valeur des variables du nœud virtuel, il peut simuler leur évolution au vu des messages reçus entretemps par le nœud virtuel.

Une version auto-stabilisante de cet algorithme est présenté dans [NL07], qui nécessite en plus l'élection d'un leader dans l'émulation de chaque nœud virtuel.

Nos besoins sont très différents. L'ensemble des nœuds participant à l'émulation d'un nœud virtuel est l'ensemble des nœuds appartenant au cluster correspondant. Puisque des mécanismes sont déjà présents pour assurer la cohérence au sein d'un cluster, nous nous appuyons dessus pour maintenir la cohérence de l'émulation. En outre, les nœuds virtuels que nous émulons n'interagissent pas avec les nœuds du système, mais uniquement entre eux et forment un nouveau système distribué, distinct du système distribué initial.

Ces caractéristiques nous permettent de mettre en place l'émulation à l'aide de mécanismes plus légers.

1.3 Problématique

À travers cette thèse, nous apportons une contribution au problème du clustering. Pour gérer efficacement un réseau, nous proposons de regrouper les machines qui le composent. Ces groupes, appelés clusters forment une partition du réseau. Nous cherchons à construire ces clusters sur un critère de taille. Pour organiser de grands réseaux, nous proposons ensuite de construire, par une approche ascendante, une structure hiérarchique équilibrée. Les clusters sont regroupés en clusters de clusters, chacun possédant le même nombre d'éléments.

Calculer cette structure hiérarchique soulève le problème d'exécuter un algorithme de clustering sur les clusters. La difficulté est qu'un cluster est composé de plusieurs machines, qui doivent constituer une seule entité du niveau supérieur. Cela nécessite de mettre en place des mécanismes pour que les nœuds d'un cluster s'accordent sur les actions de ce cluster au niveau supérieur. Il est également nécessaire que les clusters puissent communiquer entre eux.

Nous cherchons ainsi à mettre en place des mécanismes permettant aux clusters d'être les nœuds d'un nouveau système distribué, capable d'exécuter un algorithme de clustering.

1.4 Modèle

Un système distribué est constitué d'un ensemble d'entités capables d'effectuer un calcul, dotées d'une capacité de communication, et exécutant un algorithme distribué. Ces appareils sont appelés les nœuds du système (cf [Tel94]). Chacun de ces nœuds dispose d'un identifiant que nous supposons unique. Par la suite, nous ne faisons plus la distinction entre un nœud et son identifiant.

Les communications entre les nœuds sont représentées par passage de messages. Ainsi, chaque nœud a accès à deux primitives de communication *envoyer à* et *à la réception de*, lui permettant

de communiquer avec une partie des nœuds du système. Nous ne considérons que des relations de communication symétriques, c'est-à-dire que si un nœud peut envoyer des messages à un autre nœud, il peut également recevoir des messages de ce dernier.

Cette capacité de communication définit une relation de voisinage. Pour un nœud i, les nœuds avec lesquels il peut communiquer sont appelés ses voisins, dont l'ensemble est noté N_i. Cette relation est représentée par un graphe de communication $G = (V, E)$ où V est l'ensemble des nœuds du système, et $E \subset V \times V$ l'ensemble des liens de communications.

Nous considérons des systèmes dynamiques. Dans un tel système, les voisins d'un nœud peuvent changer au cours du temps, entrainant une modification de la relation de voisinage. Nous considérons de plus qu'un nœud peut à tout moment se déconnecter du système, et que de nouveaux nœuds peuvent rejoindre le système. Le graphe de communication G peut donc changer au cours du temps, ce que l'on appelle un changement topologique. Si un changement topologique affecte le voisinage d'un nœud i, nous supposons qu'il peut y réagir (éventuellement après un délai arbitraire) grâce aux primitive *à la perte de connexion vers* et *à la connexion avec*.

Les nœuds du système possèdent deux états différents. Ils sont endormis ou éveillés. Nous considérons que chaque nœud rejoignant le système est endormi. Lorsqu'un événement survient dans le système (expiration d'un compte-à-rebours, réception d'un message,...), le nœud concerné s'éveille et exécute la procédure associée à cet événement ([Lyn96]). L'algorithme distribué est ainsi constitué d'une collection d'algorithmes exécutés par les nœuds. Ces algorithmes, dits locaux, régissent la façon dont les nœuds doivent réagir aux évènements survenant dans le système. Ils sont écrits selon les habitudes de la programmation structurée ([Dij68], [DDH72]), et utilisent les primitives *envoyer à* et *à la réception de*.

Nous considérons que l'exécution des algorithmes locaux est atomique. Cela signifie que l'exécution d'un algorithme local ne peut pas être interrompue. Si un événement survient sur un nœud durant l'exécution d'un algorithme local, le nœud attend que l'exécution de cet algorithme soit terminée pour exécuter la procédure associée au nouvel évènement.

Nous nous intéressons à des algorithmes distribués uniformes, c'est-à-dire que tous les nœuds possèdent les mêmes algorithmes locaux, utilisant les mêmes variables.

L'algorithme distribué définit également les types de messages que les nœuds peuvent s'échanger. On représente les messages en transit sous la forme de quadruplets $(msg, param, em, dst)$ où em est l'identifiant du nœud émetteur, dst l'identifiant du nœud destinataire, $(em, dst) \in V$ et $param$ l'ensemble des paramètres du message. Plusieurs messages identiques peuvent être en transit en même temps. Les messages en transit forment donc un multi-ensemble que nous notons $\mathcal{M}$.

Puisque l'exécution des algorithmes locaux est atomique, les états intermédiaires que prend un nœud durant une telle exécution ne sont pas pertinents. La configuration du système est la donnée de toutes les informations pertinentes pour décrire son évolution future. Une configuration est donc constituée du graphe de communication du système G, de l'ensemble des messages en transit $\mathcal{M}$, ainsi que de l'état de chaque nœud du système, c'est-à-dire l'ensemble des valeurs de leurs variables ([Tel94]).

L'algorithme distribué définit un ensemble de transformations pouvant se produire dans le système, que l'on nomme les *transitions* de l'algorithme. Une transition correspond ainsi à l'exécution d'un algorithme local par un nœud, et se traduit par une modification des variables de ce nœud, ainsi que des messages en transit via les primitives *envoyer à* et *à la réception de*. Lors d'une transition, le graphe de communication est inchangé.

Le système peut donc évoluer de deux façons différentes :

- par une transition ;
- par un changement topologique.

Lors d'un changement topologique, les variables des nœuds sont inchangées. Si un lien de communication est retiré de G, les messages en transit sur ce lien sont enlevés de $\mathcal{M}$. C'est le seul cas ou un message peut être perdu.

Nous supposons que lorsqu'un nœud envoie un message à l'un de ses voisins, ce dernier finira par le recevoir, sauf si le lien de communication entre ces deux nœuds disparaît. En particulier, au chapitre 3, nous calculons un clustering qui émule un système distribué. Les communications dans ce système ne vérifient pas l'hypothèse FIFO : les messages entre deux clusters peuvent être reçus dans un ordre différent de leur ordre d'émission. Les messages peuvent également prendre un temps arbitrairement long avant d'être reçus.

Nous supposons ainsi que la durée entre l'émission et la réception d'un message est une variable aléatoire qui suit une loi de probabilité non bornée. Ce modèle correspond ainsi à un modèle aléatoire d'asynchronisme.

Les durées de deux communications ne sont pas forcément indépendantes. En effet, une communication de longue durée peut monopoliser le médium de communication, entrainant un allongement de la durée des autres communications en cours (phénomène de congestion). Nous supposons donc que la loi de probabilité régissant les durées de transit des messages est susceptible d'évoluer au cours du temps.

Nous supposons par contre que les tirages aléatoires effectués par les nœuds (et par conséquent les trajets des marches aléatoires) sont indépendants des temps de communication.

Considérons deux messages en transit au même moment msg_1 et msg_2. Nous appelons D_{msg_1} et D_{msg2} les durées de transit de ces deux messages. Nous supposons que chacun de ces messages à une chance d'être reçu en premier, c'est-à-dire que $0 < P(D_{msg_1} > D_{msg_2}) < 1$.

Nous supposons de plus que les durées des communications ont un même ordre de grandeur en probabilité : en règle générale, la durée de transmission d'un message n'est pas plus longue que la durée nécessaire pour faire transiter plusieurs fois un autre message.

On traduit cela de la façon suivante :

$$\forall k \in \mathbb{N}, \exists \alpha \in]0,1[, P(t_{msg_1} > k \times t_{msg_2}) < \alpha.$$

Pour un k donné, la grandeur α ne varie pas au cours du temps.

Ces suppositions ne font pas de notre système un système synchrone : le système peut en effet se comporter de façon fortement asynchrone pendant un moment. Cependant, de telles situations ne constituent pas la norme.

1.5 Plan

Nous commençons par présenter, au chapitre 2, un algorithme de clustering résistant aux changements topologiques. Cet algorithme, adapté de [BKS09], utilise des marches aléatoires pour construire un clustering orienté taille du système. Dans la mesure du possible, chaque cluster possède une partie dominante connexe dont la taille est un paramètre de l'algorithme.

Pour organiser les réseaux de grande taille, nous proposons de construire un clustering hiérarchique par une approche ascendante. Pour cela, nous définissons, au chapitre 3, les primitives nécessaires pour que les clusters forment un nouveau système distribué. Nous présentons un algorithme de clustering dans lequel les clusters ont la capacité d'émuler un nouveau système distribué. Les clusters seront ainsi capables d'exécuter des algorithmes locaux, et de s'échanger des messages. Nous utilisons cet algorithme à la section 3.4 pour concevoir un algorithme de clustering hiérarchique.

Au chapitre 4, nous exprimons formellement le fait qu'un clustering émule un système distribué. Nous définissons un nouveau système distribué, dont les nœuds sont les clusters calculés sur le système initial. Nous démontrons ensuite que l'algorithme du chapitre 3 émule correctement un système distribué. Cela signifie en particulier que les modifications apportées au système des clusters sont des transitions de l'algorithme exécuté au niveau supérieur. En nous appuyant sur cette démonstration, nous démontrons également que l'algorithme de clustering hiérarchique de la section 3.4 calcule un clustering hiérarchique du système, résistant aux changements topologiques.

Enfin, au chapitre 5, nous utilisons le clustering pour répondre à un phénomène se produisant dans les réseaux de capteurs : les collisions. En effet, si deux appareils envoient en même temps un message à un troisième appareil, la réception de ces messages peut s'avérer impossible. Les messages sont alors perdus, ce qui nécessite de les envoyer à nouveau et gaspille de l'énergie. Pour éviter ce phénomène, nous cherchons à planifier les communications au sein d'un réseau de capteurs. Alors qu'une telle planification est réalisable à l'échelle locale, elle devient rapidement difficile à mettre en place lorsque la taille du système augmente. Nous présentons, au chapitre 5, un algorithme de clustering adapté à ce problème. Sur chaque cluster, une méthode locale est déployée, permettant de planifier les communications au sein du cluster. Nous présentons ensuite, à l'aide du clustering, une méthode permettant de planifier les communications dans tout le système.

Chapitre 2

Clustering à un niveau

2.1 Introduction

Nous nous intéressons au problème de l'organisation d'un système distribué, et plus particulièrement, à l'établissement d'un clustering hiérarchique. Pour cela, nous nous basons sur un algorithme distribué construisant un clustering orienté taille, et résistant aux changements topologiques. De plus, nous voulons que de tels changements affectent une partie limitée du réseau. Ces caractéristiques nous amènent à choisir l'algorithme présenté dans [BKS12].

En effet, cet algorithme construit des clusters dont une partie dominante connexe, appelée le cœur, est de taille bornée par un paramètre de l'algorithme. Cette partie peut éventuellement être entourée de nœuds appartenant au cluster, mais sans faire partie du cœur, dits nœuds ordinaires. Cet algorithme est conçu pour des réseaux dynamiques, et résiste aux changements topologiques. De plus, lors d'une déconnexion, ou lors de la perte d'un lien de communication, seule une partie du réseau sera affectée : le cluster où a lieu le changement topologique, et éventuellement les clusters adjacents.

Nous apportons quelques modifications à cet algorithme. Premièrement, notre but est de construire un clustering hiérarchique. Nous devons donc mettre en place des communications inter-clusters. Nous modifions donc l'algorithme pour que les nœuds ordinaires soient aussi présents dans l'arbre couvrant de leur cluster.

Deuxièmement, l'algorithme présenté dans [BKS12] est auto-stabilisant. Pour cela, il s'appuie sur des marches aléatoires auto-stabilisantes, utilisant des vagues de réinitialisation (cf [BBF05]). Nous ne recherchons pas la propriété d'auto-stabilisation dans cette étude, nous concentrant sur une gestion rapide des changements topologiques. Le mécanisme des vagues de réinitialisation nécessite en outre des hypothèses sur la synchronicité qui ne sont pas présentes dans notre modèle, et qui ne sont pas vérifiées par le système distribué des clusters, présenté aux chapitres 3 et 4.

L'algorithme présenté n'utilisera donc pas de vagues de réinitialisation, ce qui nous amène à utiliser un numéro d'ordre dans l'identifiant des clusters.

En effet, dans [BKS12], l'identifiant d'un cluster est l'identifiant du nœud ayant initié sa construction. Un mécanisme utilisant les vagues de réinitialisation est présent, qui garantit que si l'initiateur quitte le cluster, par exemple après un changement topologique, alors tout le cluster sera détruit. Cela permet à ce nœud de créer de nouveau un cluster de même identifiant. En l'absence de ce mécanisme, un nœud ne doit pas pouvoir initier deux fois un cluster de même identifiant. Les identifiants des clusters sont donc constitués de la concaténation de l'identifiant du nœud initiateur et d'un numéro d'ordre, incrémenté à chaque création d'un cluster par ce nœud.

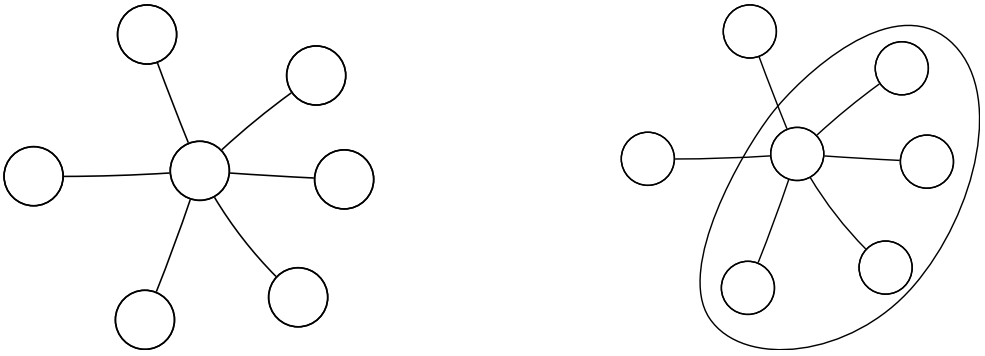

FIGURE 2.1 – Réseau en étoile

2.1.1 Caractéristiques du clustering

Pour organiser le réseau efficacement, nous voulons éviter les clusterings triviaux, constitué uniquement de clusters de taille 1 ou d'un seul cluster regroupant tous les nœuds du réseau. Pour cela, nous voulons garantir que tous les clusters sont de taille supérieure ou égale à deux. Dans l'idéal, nous aimerions également avoir le plus possible de clusters de taille K, un paramètre de l'algorithme. Cependant, il est impossible de diviser n'importe quel système distribué en parties connexes ayant une taille à la fois majorée par un paramètre K, et minorée par 2.

En effet, considérerons un réseau en étoile composé d'au moins $K + 1$ nœuds (figure 2.1). Dans cet exemple, on prend $K = 5$, et un réseau de sept nœuds. Tous clusters de taille supérieure à 1 devra comporter le nœud central pour être connexe. Construire un cluster contenant les sept nœuds amènerait à un cluster de taille supérieure à K. Un cluster de taille inférieure à 7 laisserait quant à lui des nœuds isolés, formant des clusters de taille 1.

Ainsi seule une partie dominante connexe des clusters, appelé *cœur* du cluster, peut avoir une taille bornée par K. Cet algorithme construit donc des clusters dont la taille du coeur est majorée par un paramètre K. Autour de ce cœur, dans un voisinage à au plus un saut, on peut trouver des nœuds appartenant également au cluster, mais pas à son cœur. Il sont appelés nœuds *ordinaires*. Ainsi, sur le réseau présenté à la figure 2.1, on construit un unique cluster, composé d'un cœur de cinq nœuds incluant le nœud central. Les deux autres nœuds du réseau sont des nœuds ordinaires de ce cluster.

Pour construire des cœurs de clusters de tailles bornées, il est nécessaire de connaître leurs tailles. Les nœuds seront recrutés un par un, ce qui ramène le recrutement de nouveaux nœuds à un problème d'exclusion mutuelle. Nous mettons en place un jeton, suivant une marche aléatoire, qui représente le droit d'entrée dans le cœur du cluster. Ainsi, lorsqu'un nœud possède le jeton d'un cluster, il a le droit de rentrer dans le cœur de ce cluster. S'il n'est actuellement dans aucun cluster, il devient un nœud de cœur de ce cluster, puis il transmet ce droit d'entrée à un de ses voisins, choisi uniformément au hasard. Ce processus se poursuit jusqu'à ce que le cœur du cluster ait atteint la taille maximale K.

Le cœur pouvant être entouré de nœuds ordinaires, chaque fois qu'un nœud entre dans le cœur d'un cluster, il invite ses voisins en tant que nœuds ordinaires. Un nœud de cœur invitera également ses voisins à chaque passage du jeton de son cluster.

Pour ne pas construire un clustering constitué uniquement de clusters de petites tailles, il faut être capable de détruire certains clusters. Pour mettre en place un tel mécanisme de destruction, ainsi que pour réagir aux possibles déconnexions dans le réseau, un arbre couvrant de chaque cluster est construit au cours de la circulation du jeton.

Les marches aléatoires jouant un rôle central dans cet algorithme, nous en présentons suc-

cinctement quelques propriétés avant de présenter l'algorithme de clustering.

2.1.2 Marches aléatoires

Nous considérons un message, que nous appelons *jeton*. On dit que ce message suit une marche aléatoire s'il se déplace dans le système distribué de la façon suivante : lorsqu'un nœud reçoit le jeton, il le transmet à l'un de ses voisins choisi uniformément au hasard (cf [Lov93], [IJ90], [BBS13]).

Un tel algorithme distribué est facile à mettre en place, et ne nécessite aucune connaissance de la topologie du système pour fonctionner. En effet, chaque nœud a uniquement besoin de connaître ses voisins afin d'en choisir un uniformément au hasard. De plus, seuls les voisins du nœud au moment de l'envoi influencent la circulation de la marche aléatoire. Ce processus, sans mémoire du passé, est un processus markovien ([Nor98]).

Les marches aléatoires possèdent plusieurs propriétés qui les rendent intéressantes :

- percussion : tous les nœuds du système reçoivent infiniment souvent le jeton ([Lov93]),
- couverture : le jeton visite tous les nœuds en un temps fini ([KR95]).
- rencontre : si deux jetons circulent dans le réseau, il finiront par être sur un même nœud du système ([TW91]).

Les propriétés de percussion et de couverture garantissent que les marches aléatoires sont des schémas de parcours du système : le jeton visite régulièrement chaque nœud.

Une marche aléatoire ne dépend que d'informations locales. Ainsi, après un changement topologique, une marche aléatoire continue de fonctionner et de visiter régulièrement tous les nœuds du système tant que celui-ci reste connexe.

Les marches aléatoires, de par la simplicité de mise en œuvre, et de par leur adaptativité aux changements topologiques, ont été utilisées pour résoudre divers problèmes des systèmes distribués. [IJ90] utilise une marche aléatoire pour résoudre le problème de l'exclusion mutuelle dans un système distribué. [BIZ89] utilise les marches aléatoires pour construire un arbre couvrant du système.

Nous présentons maintenant un algorithme distribué utilisant une marche aléatoire pour construire un arbre couvrant du système. Nous utiliserons cet algorithme plus tard pour construire un arbre couvrant de chacun des clusters que nous calculerons. Cet algorithme utilise un seul message, appelé jeton.

Cet algorithme construit deux représentations de l'arbre couvrant qu'il calcule : une représentation répartie sur les nœuds et une représentation complète portée par le jeton. Chaque nœud possède ainsi une variable, $father$, contenant l'identifiant de son père dans l'arbre couvrant calculé. Le jeton porte quant à lui un tableau, noté $tree$, tel que $tree[i]$ est l'identifiant du père de i dans l'arbre couvrant calculé.

Lorsqu'un nœud i reçoit le jeton venant d'un nœud e, il exécute la procédure 1. Le nœud i devient la racine de l'arbre couvrant, et il devient le père de l'expéditeur. Ces informations sont notées dans le tableau $tree$ du jeton (ligne 2 et 3). Le nœud e a précédemment reçu le jeton et l'a envoyé au nœud i. Par conséquent, il a également exécuté la procédure 1. Il a choisit uniformément au hasard le nœud i parmi ses voisins et lui a envoyé le jeton. Par conséquent, la variable $father_e = i$ indique déjà que i est le père de e.

Le nœud i sélectionne ensuite un de ses voisins uniformément au hasard, qui devient son père dans l'arbre couvrant, et il lui envoie le jeton. Cette information apparaîtra dans $tree$ à la réception du jeton.

Les deux représentations de cet arbre sont compatibles, c'est-à-dire que $tree[i] = j$ si et seulement si $father_i = j$ (sauf sur la racine puisque les deux représentations ne sont pas mises

Algorithme 1 À la réception du jeton en provenance du nœud e.

$father \leftarrow$ valeur aléatoire dans N_i
$Token.tree[i] \leftarrow i$
$Token.tree[e] \leftarrow i$
Envoyer $Token$ à $father$

à jour au même moment).

La propriété de couverture des marches aléatoires nous garantit que tous les nœuds du système finiront par être dans cette structure. De plus, cette structure est bien un arbre :

- elle est connexe : si deux nœuds sont dans cette structure, ils ont tous deux reçu le jeton. Le chemin suivi par le jeton entre ces deux nœuds fait partie de la structure, puisque chaque nœud recevant le jeton y est intégré.
- elle est acyclique : il y a une seule racine, chaque nœud ne peut avoir qu'un seul père, et le nœud racine n'en a pas. La structure étant connexe, elle est donc acyclique.

Dans cette thèse, les marches aléatoires vont jouer un rôle important dans la construction et la mise à jour des clusters que nous calculons.

2.1.3 Spécification : clustering

La description d'un clustering nécessite un certain nombre de variables et de messages. Cette spécification définit un clustering basé sur un jeton circulant, avec des clusters ayant des cœurs de tailles bornées, et ayant accès à des communications internes.

Les nœuds utilisent les variables suivantes :

- *cluster* : variable indiquant le cluster auquel le nœud appartient ;
- *core* : variable booléenne indiquant si le nœud est un nœud de cœur.

Les messages suivants participent de la définition des clusters :

- *Token* : les jetons, qui circulent dans le réseau et recrutent les nœuds de cœur.

Les messages *Token* doivent en outre contenir les informations suivantes :

- *id* : l'identifiant du cluster que ce jeton construit ;
- *tree* : un arbre couvrant du cluster d'identifiant *id*, représenté par un tableau.

Définition 1. *Un cluster C_x est un ensemble de nœuds $C_x = \{i \in V / cluster_i = x\}$, et son cœur est $K_x = \{i \in C_x / core = \textbf{\textit{faux}}\}$.*

Dans la suite, les variables dans un message *msg* seront notées *msg.var*.

Définition 2. *Un cluster C_x est dit* correct *si et seulement si :*

- *Structure du cluster :*
  - $1 \leq |K_x| \leq K$
  - *K_x est un ensemble dominant connexe de C_x (K_x est connexe et tous les nœuds de C_x sont adjacents à K_x)*
  - *C_x est connexe*
- *Messages en circulation : Il existe un unique jeton Token en circulation dans le réseau tel que $Token.id = x$ (ce jeton est noté $Token_x$ dans la suite), et tel que $Token_x.tree$ est un arbre couvrant du cluster C_x.*

L'arbre couvrant permet la communication intra-cluster. Le jeton jouera aussi un rôle dans l'émulation d'un système distribué par les clusters, abordée dans le chapitre suivant.

Définition 3. *Un cluster C_x est complet s'il a K nœuds de cœur, i.e. $|K_x| = K$.*

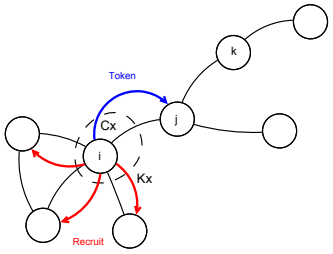

FIGURE 2.2 – Création d'un cluster

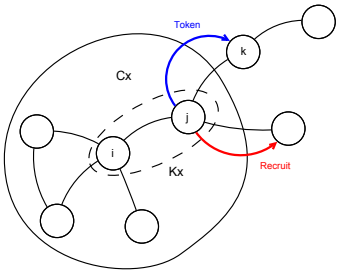

FIGURE 2.3 – Marche aléatoire

Définition 4 (spécification). *Une configuration respecte la spécification si et seulement si :*
- *Tous les clusters sont corrects et ont un cœur de taille* ≥ 2 *;*
- *Chaque noeud est dans un cluster ;*
- *Si deux clusters sont voisins, l'un au moins est complet.*

2.2 Description générale de l'algorithme

Chaque nœud est endormi au début de l'exécution. Les nœuds se réveillent à un instant aléatoire, ou lors de la réception d'un message. À son réveil, un nœud initialise ses variables et lance la procédure correspondant à la cause de son réveil :
- *wakeup*() s'il a atteint la fin de sa période de sommeil,
- *à la réception de* s'il reçoit un message avant.

Lorsqu'un nœud i s'éveille à la fin de sa période de sommeil, il crée un nouveau cluster avec un identifiant x, contenant l'identifiant i du nœud et un numéro d'ordre. Le nœud i devient le premier nœud de cœur du cluster C_x, et il invite ses voisins à devenir nœuds ordinaires du cluster C_x. Ensuite, il envoie un message $Token_x$ à un de ses voisins choisi uniformément au hasard, qui devient son père dans l'arbre couvrant du cluster C_x. Ce message est ainsi transmis de nœud en nœud et recrute un par un les nœuds qu'il rencontre au sein du cœur du cluster. Cette circulation permet de créer et de maintenir un arbre couvrant du cluster, d'une manière similaire à l'algorithme 1. Cet arbre est enraciné dans le nœud possédant le jeton, et possède deux représentations : l'une, portée par le jeton, et l'autre, distribuée, utilisant une variable $father$. Les messages $Token$ contiennent la taille actuelle du cœur de cluster, ainsi qu'une représentation de l'arbre couvrant du cluster. Tant que $Token.size < K$, un nœud ordinaire ou non clusterisé qui reçoit le jeton rejoint le cœur du cluster. Il invite ensuite ses voisins au sein du cluster en tant que nœuds ordinaires à l'aide de messages $Recruit[x]$, et transmet le jeton à un de ses voisins choisi uniformément au hasard. Ce voisin devient le père du nœud dans l'arbre couvrant du cluster. À la réception d'un message $Recruit[x]$, un nœud non clusterisé rejoint le cluster C_x en tant que nœud ordinaire, et l'expéditeur du message $Recruit[x]$ devient son père dans l'arbre couvrant du cluster.

Exemple 5. *Dans le réseau présenté en figure 2.2, le nœud i s'éveille et crée un nouveau cluster d'identifiant* $(i, 1)$. *Le jeton* $Token_{(i,1)}$ *est envoyé au nœud j, sélectionné aléatoirement. Les voisins du nœud i sont recrutés en tant que nœuds ordinaires à l'aide de messages* $Recruit[(i, 1)]$. *Cela amène à la situation de la figure 2.3, où le nœud j transmet le jeton* $Token_{(i,1)}$ *au nœud k. Après réception par le nœud k, on se retrouve dans la situation de la figure 2.4, qui présente l'arbre couvrant actuel du cluster.*

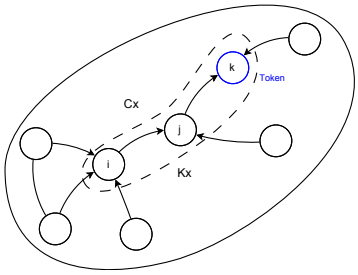

FIGURE 2.4 – Arbre couvrant

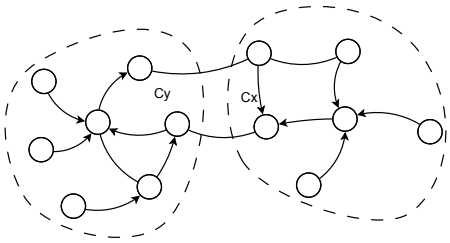

FIGURE 2.5 – Deux clusters

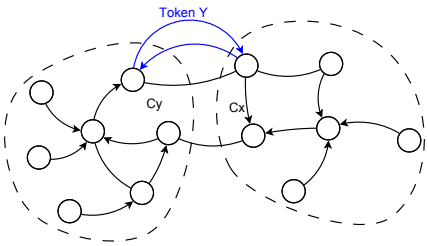

FIGURE 2.6 – $Token_y$ rencontre x

Le jeton *Token* va continuer, au cours de sa circulation dans le réseau, à recruter les nœuds de cœur jusqu'à ce que le cluster soit complet, ou jusqu'à ce qu'il rencontre un nœud de cœur d'un autre cluster. Dans ce cas, si les deux clusters sont incomplets, l'un des deux peut (et doit) être détruit pour laisser grandir l'autre. Étant donné qu'un jeton $Token_x$ transporte un arbre couvrant du cluster C_x enraciné sur le nœud détenant le jeton, si ce dernier rencontre un autre cluster C_y et qu'il doit y avoir destruction d'un cluster, c'est le cluster C_x qui s'auto-détruit en propageant une vague de message $Delete[x]$ sur l'arbre couvrant. Un nœud recevant un tel message le propage, puis quitte le cluster et s'endort.

Pour éviter des destructions mutuelles de clusters voisins, seul celui de plus petit identifiant peut être détruit.

Ainsi, lorsqu'un nœud de cœur d'un cluster C_y reçoit le jeton d'un autre cluster $Token_x$, si l'un des deux clusters est complet, ou si $x > y$, le jeton est simplement renvoyé à l'expéditeur.

Exemple 6. *Dans la configuration présentée dans la figure 2.5, les clusters C_x et C_y sont tous les deux incomplets. On a $x < y$. Si le jeton $Token_y$ est reçu par le nœud i, nœud du cœur de C_x, alors il est simplement réexpédié au nœud expéditeur (figure 2.6). Au contraire, si le jeton $Token_x$ est reçu par un nœud j membre du cœur du cluster C_y, alors le message $Token_x$ est détruit. Un message $Delete[x]$ est envoyé à l'expéditeur du jeton, ce qui initie la destruction du cluster C_x par propagation de messages $Delete[x]$ le long de l'arbre couvrant du cluster C_x (figure 2.7).*

Notre modèle ne reposant pas sur l'hypothèse FIFO, un mécanisme supplémentaire est nécessaire pour la gestion des nœuds ordinaires au cours de la destruction d'un cluster. En effet, un nœud j peut envoyer un message $Recruit[x]$ a un de ses voisins au passage du jeton $Token_x$. Par la suite, si le jeton passe en destruction, le nœud j envoie un message $Delete[x]$ au nœud k. Ainsi, si le nœud k reçoit le message $Delete[x]$ en premier, n'étant pas dans ce cluster, il l'ignore. Il reçoit ensuite le message $Recruit[x]$ et rejoint le cluster C_x en tant que nœud ordinaire.

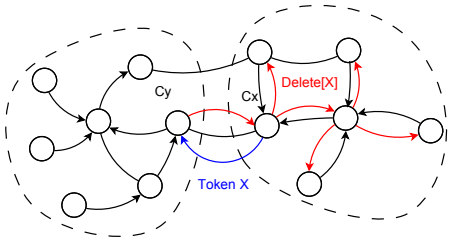

FIGURE 2.7 – $Token_x$ rencontre y

Par conséquent, lorsqu'un nœud rejoint un cluster C_x en tant que nœud ordinaire, il envoie un message $ACK[x]$ à son père. Si celui-ci n'est plus un nœud de cœur de ce cluster, il envoie un message $Delete[x]$ au nœud k, garantissant que celui-ci ne restera pas nœud ordinaire de C_x.

Quand $Token_x.size = K$, le cluster C_x est complet, et le jeton $Token_x$ continue sa marche aléatoire dans le cluster. Cette circulation du jeton va permettre de maintenir à jour l'arbre couvrant du cluster, de mettre à jour les variables locales et les informations dans le jeton, ainsi que de réagir aux changements topologiques qui pourront survenir au sein du cluster.

Si le jeton $Token_x$ rencontre un nœud qui n'est pas dans le cluster C_x alors que $Token_x.size \geq K$, alors le jeton est simplement renvoyé à l'expéditeur.

Ainsi, un jeton qui est reçu par un nœud extérieur à son cluster va soit le recruter, soit être renvoyé à l'expéditeur, soit être détruit tout comme le cluster qu'il représente. Cela garantit que le jeton d'un cluster reste dans un voisinage à au plus un saut du cœur de son cluster.

Lorsqu'un changement topologique a lieu, le cluster peut devenir invalide s'il devient non connexe. Notre algorithme le détecte en vérifiant la connexité de l'arbre couvrant du cluster. Les changements topologiques qui peuvent affecter la connexité d'un cluster sont :

- les pertes d'un lien de communication si celui-ci est une arrête de l'arbre couvrant du cluster,
- la déconnexion d'un nœud de cœur.

En effet, l'ajout d'un lien de communication entre deux nœuds n'invalide pas la structure des clusters. De même, l'ajout d'un nœud ne change rien à l'état des clusters adjacents : il finira par être recruté ou par créer son propre cluster par application de l'algorithme.

Si un lien de communication (i, j) disparaît, les nœuds i et j finiront par détecter cette disparition. Si cette arête fait partie de l'arbre couvrant, alors l'un des nœuds est le père de l'autre dans l'arbre couvrant, par exemple i est le père de j. La variable *father* permet au fils de savoir qu'il a perdu le lien de communication avec son père. Ainsi, lorsque j détecte la perte de communication avec son père i, il quitte le cluster. Il propage ensuite des messages $Delete[x]$ sur le sous-arbre enraciné en j et s'endort pour un temps aléatoire. Tous ces nœuds vont quitter le cluster et passer en sommeil jusqu'à ce que cette zone du réseau soit clusterisée à nouveau.

Comme le jeton du cluster est à la racine de l'arbre couvrant, on sait qu'il reste dans un voisinage à un saut de la partie restante du cluster. La propriété de percussion des marches aléatoires garantit donc que le jeton finira par visiter le nœud i. À ce moment, le nœud j étant inaccessible, l'arbre couvrant présent dans le jeton, ainsi que la taille du cœur $Token.size$, sont mis à jour pour décrire l'état actuel du cluster.

Exemple 7. *Dans le réseau présenté dans la figure 2.8, le lien de communication (i, j) disparaît. Le nœud i est le père de j dans l'arbre couvrant du cluster ($father_j = i$). Lorsque le nœud j détecte la perte du lien de communication, il envoie un message $Delete[x]$ à tous ses voisins,*

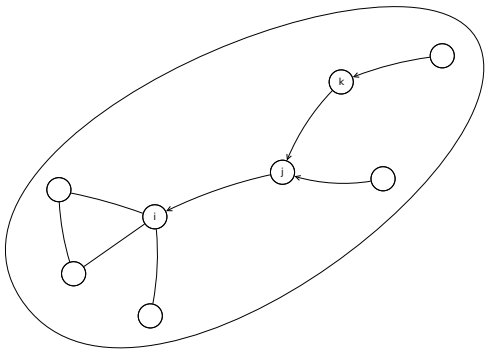

FIGURE 2.8 – Perte de connexion entre les nœuds i et j

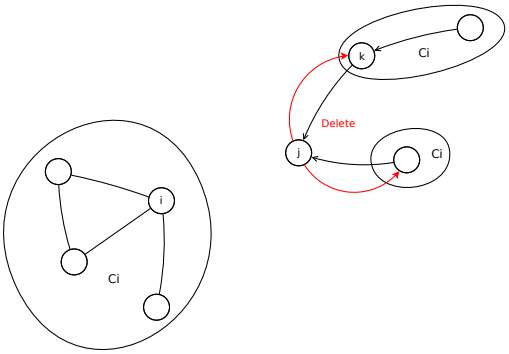

FIGURE 2.9 – Message *Delete* propagé

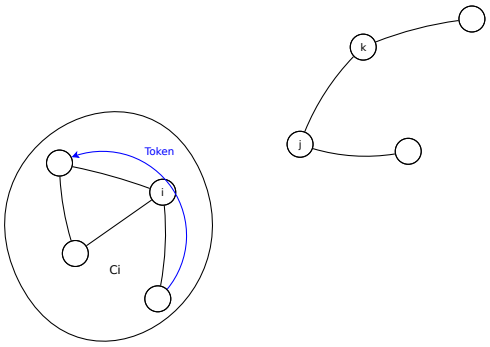

FIGURE 2.10 – Mise à jour du jeton *Token*

quitte le cluster et s'endort pour une durée aléatoire (figure 2.9). Tous les enfants du nœud j dans l'arbre vont recevoir ce message et le prendre en compte (les autres voisins l'ignorent). Ils quittent le cluster et propagent le message Delete[x]. En particulier, le nœud k envoie un message Delete[x] à tous ses voisins et s'endort. Cela aboutit à la configuration présentée dans la figure 2.10, où l'ensemble des nœuds du sous-arbre enraciné en j ont quitté le cluster. Parallèlement à cela, lorsque le jeton visite le nœud i, l'arbre couvrant Token.tree et la taille du cœur Token.size sont mis à jour pour correspondre à la situation présentée en figure 2.10.

2.3 Algorithme détaillé

Cet algorithme distingue deux types de nœuds différents, les nœuds de cœur et les nœuds ordinaires. Les nœuds de cœur composent la partie principale d'un cluster, formant un ensemble dominant connexe de ce dernier. Les nœuds ordinaires sont quant à eux présents pour pallier l'impossibilité, dans le cas général, de créer un clustering de taille bornée sans nœud isolé (voir exemple 2.1).

Chaque nœud possède les variables suivantes :

- *cluster* : l'identifiant du cluster dont fait partie le nœud. Cet identifiant est formé de la concaténation de l'identifiant du nœud créant le cluster et d'un numéro d'ordre ;
- *core* : un booléen indiquant si le nœud est un nœud de cœur ;

- *nexto* : le prochain numéro d'ordre à utiliser si ce nœud est amené à créer un nouveau cluster ;
- *complete* : un booléen indiquant (sur les nœuds de cœur) si le nœud est membre d'un cluster complet ;
- $father$: l'identifiant du père du nœud dans l'arbre couvrant du cluster ;
- $size1$: un booléen indiquant (sur les nœuds de cœur) si le cluster ne possède qu'un seul nœud de cœur.

De plus, un service de niveau inférieur assure la détection du voisinage de chaque nœud. Chaque nœud a ainsi accès à une variable N représentant l'ensemble des nœuds voisins, qui est maintenue à jour par ce service. Après un changement topologique, la variable N est donc mise à jour automatiquement. Lorsque le service de niveau inférieur enlève un nœud j de la variable N, il déclenche également l'exécution de la procédure *à la perte de la connexion vers* j.

L'algorithme utilise quatre types de messages :

- $Token[id, tree, corenodes, size]$: les jetons circulant aléatoirement dans le réseau, recrutent les nœuds de cœur et maintiennent un arbre couvrant de leur cluster ;
- $Recruit[id]$: messages recrutant les nœuds en tant que nœuds ordinaires ;
- $ACK[id]$: messages indiquant qu'un nœud vient de devenir nœud ordinaire du cluster indiqué ;
- $Delete[id]$: messages permettant la destruction d'un cluster.

Les jetons transportent les informations suivantes :

- id : l'identifiant du cluster associé au jeton.
- $tree$: un arbre couvrant du cluster, représenté par un tableau ($tree[i] = j$ indique que le nœud j est le père du nœud i).
- $corenodes$: un tableau indiquant les nœuds de cœur.
- $size$: la taille actuelle du cœur du cluster.

Dans la suite, on notera i l'identifiant du nœud en train d'exécuter la procédure que nous décrivons. Si cette procédure à été déclenchée par un message, on notera e l'identifiant du nœud expéditeur.

Au début de l'exécution, tous les nœuds sont endormis. Lorsqu'il s'éveille, un nœud exécute la procédure d'initialisation *init*, qui initialise toutes ses variables à zéro ou $\perp$ (excepté pour la variable *nexto*, qui n'est initialisée qu'au premier réveil). Un nœud peut se réveiller de deux façons différentes, soit à un instant aléatoire, auquel cas il lance la procédure *wakeup*(), soit à la réception d'un message, auquel cas il lance la procédure *à la réception de* adéquate.

Initialisation.

Algorithme 2 *Init*

$cluster \leftarrow (\perp, 0)$
$core \leftarrow$ **faux**
$nexto \leftarrow 1$
$complete \leftarrow$ **faux**
$father \leftarrow \perp$
$size1 \leftarrow$ **faux**

Procédure *wakeup*. Lors de l'exécution de la fonction *wakeup*() sur un nœud i ayant des voisins, celui-ci crée un nouveau cluster d'identifiant $x = (i, nexto_i)$, ainsi que le jeton $Token_x$ associé. Il rejoint le cœur du cluster C_x, invite ses voisins en tant que nœuds ordinaires, puis transmet le jeton à un voisin sélectionné uniformément au hasard, qui devient son père dans l'arbre couvrant du cluster. Le cluster possède alors un unique nœud de cœur, et la variable $size1$ a la valeur **vrai**.

Si le nœud est isolé, il ne pourra créer un cluster de taille au moins deux. Par conséquent, il ne crée pas de cluster et se rendort.

Algorithme 3 *wakeup*()

si $N \neq \emptyset$ **alors**
  $Token$ =nouveau message Token
  $Token.size \leftarrow 0$
  $Token.id \leftarrow (i, nexto)$
  $Token.tree \leftarrow$ Vecteur vide
  $Token.tree[i] \leftarrow i$
  $JoinCore()$
  $size1 \leftarrow$ **vrai**
  $nexto \leftarrow nexto + 1$
  Envoyer $Recruit[cluster]$ à tous les nœuds de N
  $father \leftarrow$ valeur aléatoire dans N
  Envoyer $Token$ à $father$
sinon
  Passer à l'état *endormi*
fin si

Fonction *JoinCore*.

La fonction *JoinCore* est appelée sur un nœud i pour qu'il rejoigne le cœur d'un cluster. Cette fonction est appelée par deux procédures, la procédure d'éveil, qui crée un cluster, et la procédure *à la réception d'un message Token*. Dans les deux cas, le jeton du cluster est détenu par le nœud. La fonction *JoinCore* va donc modifier les variables du nœud pour lui faire intégrer le cluster en tant que nœud de cœur, et elle va également modifier les variables du jeton $Token.corenodes$ et $Token.size$.

Algorithme 4 *JoinCore*()

$cluster \leftarrow Token.id$
$core \leftarrow$ **vrai**
$Token.size \leftarrow Token.size + 1$
$Token.corenodes[i] \leftarrow$ **vrai**
$complete \leftarrow (Token.size \geq K)$

À chaque fois qu'un nœud entre dans le cœur d'un cluster, il invite tous ses voisins à rejoindre le cluster en tant que nœuds ordinaires. Pour ce faire, il envoie à tous ses voisins un message *Recruit*, comme dans la procédure *wakeup*().

À la réception d'un message $Recruit[x]$.

Lorsqu'un nœud i reçoit un message $Recruit[x]$ en provenance d'un nœud e, il exécute la procédure *JoinOrdinary* prenant en paramètre l'expéditeur du message *Recruit*. Si le nœud i est non clusterisé, il rejoint le cluster C_x en tant que nœud ordinaire. L'expéditeur e devient son père dans l'arbre couvrant. Le nœud i informe ensuite son père qu'il a rejoint le cluster C_x en tant que nœud ordinaire à l'aide d'un message $ACK[x]$.

Dans les autres cas, le nœud ignore le message.

Dans chaque cluster, le jeton va continuer sa circulation et recruter au fur et à mesure les nœuds de cœur. Afin de construire le plus possible de clusters complets, il nous faut un mécanisme permettant de détruire un cluster. On autorise ainsi la destruction des clusters incomplets. Une telle destruction est réalisée en propageant un message $Delete[x]$ sur l'arbre couvrant du

Algorithme 5 *JoinOrdinary*(e) (Réception d'un message *Recruit*[x])

si $cluster = (\perp, 0)$ **alors**
  $cluster \leftarrow x$
  $core \leftarrow$ **faux**
  $father \leftarrow e$
  Envoyer $ACK[x]$ à $father$
fin si

cluster C_x. Ce mécanisme est également utilisé en cas de changement topologique nécessitant une destruction partielle ou totale d'un cluster.

À la réception d'un message *Delete*[x] **venant du nœud** e**.** Un tel message est propagé sur l'arbre couvrant (ou sur un sous-arbre) pour détruire le cluster C_x. Si un nœud du cluster C_x reçoit un tel message en provenance de son père, il quitte le cluster. Il propage également ce message à ses voisins, puis s'endort.

Il en résulte une propagation de message *Delete*[x] le long de l'arbre couvrant du cluster C_x qui amène à la destruction du cluster (ou d'une partie du cluster).

Pour quitter son cluster, un nœud exécute la procédure *Leave*, qui réinitialise toutes les variables du nœud, sauf la variable *nexto*.

Un nœud recevant un message *Delete*[x] venant d'un autre nœud que son père l'ignore. En particulier, les nœuds n'appartenant pas au cluster C_x ignorent les messages *Delete*[x]

Algorithme 6 *Leave*()

$cluster \leftarrow (\perp, 0)$
$core \leftarrow$ **faux**
$complete \leftarrow$ **faux**
$father \leftarrow \perp$

Algorithme 7 À la réception d'un message *Delete*[x] venant d'un voisin e

si $e = father \wedge x = cluster$ **alors**
  Envoyer *Delete*[x] à tous les nœuds dans N.
  Leave()
  Passer à l'état *endormi*
fin si

Les seuls changements topologiques qui peuvent rendre un cluster incorrect sont la suppression d'un lien de communication utilisé comme arête de l'arbre couvrant, ou la déconnexion d'un nœud de cœur. En effet, l'ajout d'un nœud ou d'un lien ne peut pas rompre la connexité des clusters. De plus, si un nœud ordinaire quitte le réseau, le cœur de son cluster reste un ensemble dominant connexe du cluster, et celui-ci reste valide. De même, si une arête disparaît alors qu'elle n'est pas utilisée dans l'arbre couvrant du cluster, alors ce cluster reste connexe et valide.

Si e **quitte** N **(déconnexion entre le nœud courant et** e**).** La détection du voisinage est effectuée par un service de niveau inférieur. Lorsqu'un nœud e quitte le voisinage du nœud i, ce service met à jour la variable N et déclenche l'exécution de la fonction *à la perte de la connexion vers* e (algorithme 8).

Lorsqu'un nœud perd la connexion avec un voisin e, si ce nœud est son père, il quitte le cluster et envoie un message *Delete* à tous ses voisins, comme s'il venait de recevoir un message *Delete* en provenance de e.

Algorithme 8 À la perte de la connexion vers e

si $e = father$ **alors**
  Envoyer $Delete[cluster]$ à tous les nœuds dans N.
  Leave()
  Passer à l'état *endormi*
fin si

Lorsqu'un lien de communication (i, e) disparaît, les éventuels messages en transit sur ce lien sont perdus. Si le message en transit est :

- $Recruit[x]$: si ce message n'avait pas été ignoré, le nœud recruté aurait quitté le cluster en détectant la perte de connexion. Ce qui amène à la même situation que d'ignorer le message.
- $Delete[x]$: Ce message aurait été ignoré, sauf si l'un des nœuds est le père de l'autre dans l'arbre couvrant du cluster C_x. Dans ce cas, il aurait quitté le cluster, ce qu'il fera en détectant la perte du lien, ce qui amène à la même configuration.
- $Token$: le nœud qui a envoyé le jeton considère l'autre comme son père dans l'arbre couvrant. Par conséquent, à la perte du lien de communication, considérant l'autre nœud comme son père, il quitte le cluster et enclenche sa destruction avec des messages $Delete$. Dans cette situation, le cluster sans jeton sera entièrement détruit, ce qui amène à une situation valide.

Dans tous ces cas, la perte d'un message durant un changement topologique n'empêche pas le bon déroulement de l'algorithme. Plus précisément, on montrera que l'on peut construire une exécution sur la nouvelle topologie qui amènerait à la même configuration.

À la réception d'un message $ACK[x]$.

Lorsqu'un nœud reçoit un message $ACK[x]$ venant d'un nœud e, cela signifie que le nœud e a rejoint le cluster C_x en tant que nœud ordinaire, et que le nœud i est le père du nœud e. Ainsi, si le nœud i n'est pas un nœud de cœur du cluster C_x, il envoie un message $Delete[x]$ à e.

Algorithme 9 À la réception d'un message $ACK[x]$ venant d'un voisin e

si $cluster \neq x \vee core =$ **faux alors**
  Envoyer $Delete[x]$ à e.
fin si

À la réception d'un message $Token[x, tree, corenodes, size]$.

Les deux rôles principaux d'un jeton sont :

- de recruter les nœuds de cœur dans son cluster,
- construire et maintenir un arbre couvrant du cluster.

Un jeton $Token_x$ circule aléatoirement dans le réseau en recrutant les nœuds non clusturisés qu'il rencontre. Durant ce parcours, un arbre couvrant du cluster est calculé. Deux représentations de cet arbre existe : la variable *tree* du jeton, et les variables $father$ des nœuds. Il existe un décalage entre ces deux représentations. Il y a le même décalage d'une étape que dans l'algorithme 1. En effet, le destinataire du jeton est le père dans l'arbre couvrant de l'expéditeur, bien que cette information ne soit ajoutée à la variable *tree* qu'à la réception du jeton. De plus, les nœuds ordinaires subissent un délai avant d'être ajoutés à la variable *tree*. En effet, un nœud ordinaire est le fils du nœud qui l'a recruté en tant que nœud ordinaire (algorithme 5). Cependant, cette information n'apparaît pas dans la variable *tree*. Le nœud ordinaire entrera dans la représentation *tree* lorsqu'il recevra le jeton.

Lorsqu'un nœud i reçoit un message $Token_x$ en provenance d'un nœud e, les cas de figure suivants sont possibles :

- Mise à jour des informations sur un nœud de cœur : si le nœud i est un nœud de cœur du cluster C_x, alors ce nœud reste dans le cœur. Il y a mise à jour de l'arbre couvrant, puis synchronisation des variables du nœud et du jeton à l'aide de la fonction *UpdateStatus* explicitée plus loin. Ensuite, le nœud i invite tous ses voisins à rejoindre le cluster en tant que nœuds ordinaires à l'aide d'un message $Recruit[x]$. Enfin, le jeton est transféré à un voisin choisi au hasard, et la variable $father$ est mise à jour pour indiquer le nœud à qui le jeton est envoyé.
- Mise à jour des informations sur un nœud ordinaire : si le nœud i est un nœud ordinaire du cluster C_x, et si ce cluster est complet, alors le nœud i ne peut être promu nœud de cœur. Les informations du jeton sont mises à jour via *UpdateStatus*, puis le jeton est renvoyé au nœud expéditeur e. Le fait que ce nœud e soit le père du nœud i dans l'arbre couvrant est enregistré dans la variable *tree* du jeton.
- Recrutement ou promotion d'un nœud dans le cœur : si i est un nœud ordinaire ou non clusterisé, et que le cluster C_x n'est pas complet, alors le nœud devient un nœud de cœur de ce cluster. Il rejoint le cluster via un appel à la fonction *JoinCore*, à la suite de quoi les informations locales sont synchronisées avec le jeton via *UpdateStatus*. Ensuite, le nœud invite ses voisins à rejoindre le cluster en tant que nœuds ordinaires à l'aide de messages $Recruit[x]$. Enfin, le jeton est relayé à un nœud voisin choisi uniformément au hasard, qui devient le père de i dans l'arbre couvrant du cluster.
- Destruction du cluster C_x : si le nœud i est un nœud de cœur d'un autre cluster C_y, que ce cluster ainsi que le cluster C_x sont tous deux incomplets, et si $y > x$, alors le nœud i initie la destruction du cluster C_x. Pour ce faire, il envoie un message $Delete[x]$ à l'expéditeur e. Cela initie la propagation d'une vague de messages $Delete[x]$ le long de l'arbre couvrant du cluster, ce qui détruit ce cluster. Le nœud i envoie également un message $Recruit[y]$ à l'expéditeur e pour l'inviter en tant que nœud ordinaire. De même, si le cluster C_x ne comporte qu'un seul nœud de cœur ($Token.size = 1$), il est détruit. Cela garantit qu'il n'y aura pas de cluster de taille 1. De plus, nous interdisons à un cluster ayant un seul nœud de cœur ($size1 =$ **vrai**) de détruire un cluster. Cet unique nœud de cœur devient un nœud ordinaire du cluster qu'il rencontre, ce qui accélère la convergence et simplifie la démonstration de cette convergence présentée au chapitre 4.
- Renvoi du jeton à l'expéditeur : dans tous les autres cas, c'est à dire si le cluster C_x est complet, ou si le nœud i est dans un cluster C_y complet ou tel que $x < y$, alors le jeton est simplement réexpédié à l'envoyeur e. Si le jeton est envoyé à un nœud ayant quitté le cluster suite à un changement topologique, le jeton $Token_x$ et un message $Delete[x]$ peuvent se croiser sur un canal de communication. Pour éviter que le jeton se retrouve renvoyé indéfiniment entre deux nœuds devenus non clusterisés, nous n'autorisons pas que le renvoi du jeton par la racine de l'arbre couvrant $Token.tree$ (ligne 36).

Lorsqu'un nœud j extérieur au cluster C_x renvoie le jeton $Token_x$ à un nœud i qui le lui avait envoyé, j ne doit pas entrer dans l'arbre couvrant du cluster. Lorsque cela se produit, la racine de l'arbre couvrant n'a pas été changée et $Token_x.tree[i] = i$. Dans ce cas la, la variable $Token.tree$ n'est donc pas modifiée (ligne 4 à 8 de l'algorithme 10).

Fonction *UpdateStatus*(). Le but de cette fonction est de maintenir la cohérence entre l'arbre $Token.tree$ et le cluster à l'aide des informations locales que possède le nœud, ainsi que la variable $Token.size$. Si une incohérence est détectée entre les informations locales et l'arbre $Token.tree$, cette fonction procède aux rectifications nécessaires dans l'arbre ou dans le cluster. Ainsi, si un nœud considère le nœud courant comme son père dans l'arbre couvrant du cluster, et si ce nœud

Algorithme 10 À la réception d'un message $Token_x$ venant du noeud e

si $(cluster = x) \wedge (core)$ **alors**
  {mise à jour des informations d'un nœud de cœur}
  $UpdateStatus()$
  si $(Token.tree[i] \neq i)$ **alors**
5: {le jeton ne vient pas d'être renvoyé à l'expéditeur}
    $Token.tree[e] \leftarrow i$
    $Token.tree[i] \leftarrow i$
  fin si
  $complete \leftarrow (Token.size \geq K)$
10: $size1 \leftarrow (Token.size \geq 2)$
  Envoyer $Recruit[x]$ à tous les nœuds dans N
  Envoyer $Token$ à $father$ choisi uniformément au hasard dans N
sinon si $(cluster = x) \wedge (\neg core) \wedge (Token.size \geq K)$ **alors**
  {mise à jour des informations d'un nœud ordinaire}
15: $UpdateStatus()$
  $Token.tree[e] \leftarrow i$
  $Token.tree[i] \leftarrow i$
  $father \leftarrow e$
  Envoyer $Token$ à $father$
20: **sinon si** $(Token.size < K) \wedge (\neg core)$ **alors**
  {recrutement d'un nœud dans le cœur}
  $UpdateStatus()$
  $JoinCore()$
  $Token.tree[e] \leftarrow i$
25: $Token.tree[i] \leftarrow i$
  $complete \leftarrow (Token.size \geq K)$
  $size1 \leftarrow (Token.size \geq 2)$
  Envoyer $Recruit[x]$ à tous les nœuds dans N
  Envoyer $Token$ à $father$ choisi uniformément au hasard dans N
30: **sinon si** $[(Token.size < K) \wedge (\neg complete) \wedge (\neg size1) \wedge (cluster > x)] \vee (Token.size = 1)$ **alors**
  {destruction du cluster x, qui n'est pas le cluster courant $cluster_i$}
  Envoyer $Delete[x]$ à e
  Envoyer $Recruit[cluster]$ à e
sinon
35: {renvoi à l'expéditeur}
  $UpdateStatus()$
  si $Token.tree[i] \neq i$ **alors**
    Envoyer $Token$ à e
  fin si
40: **fin si**

n'est pas un voisin de i, alors il y a eu un changement topologique. La fonction *UpdateStatus* va nettoyer l'arbre couvrant de toute cette branche inaccessible. Pour cela, cette fonction utilise une procédure récursive appelée *RemoveFromTree*, qui enlève un nœud de l'arbre, le décompte de *Token.size* si c'est un nœud de cœur, et s'appelle elle-même sur les descendants dans l'arbre.

De même, si le jeton est reçu par un nœud i ne faisant pas partie du cluster, mais dans l'arbre *Token.tree*, elle met à jour l'arbre *Token.tree* en enlevant le sous-arbre enraciné en ce nœud i.

Enfin, si le jeton est reçu par un nœud i appartenant au cluster, mais absent de l'arbre *Token.tree*, alors ce nœud i quitte le cluster et envoie un message $Delete[x]$ à tous ses voisins. En effet, le nœud ne dispose d'aucune information sur sa descendance dans l'arbre distribué. La fonction *UpdateStatus* ne peut donc pas ajouter ces nœuds à l'arbre *Token.tree*. Nous forçons donc tous les éventuels descendants de i à quitter le cluster pour que l'arbre et le cluster soit de nouveau cohérents.

Algorithme 11 *UpdateStatus*()

pour tout k **avec** $Token.tree[k] = i$ **faire**
  si $k \notin N$ **alors**
    $RemoveFromTree[k]$
  fin si
fin pour
si $cluster_i \neq x \wedge Token.tree[i] \neq \perp$ **alors**
  $RemoveFromTree[i]$
fin si
si $cluster_i = x \wedge Token.tree[i] = \perp$ **alors**
  Envoyer $Delete[cluster]$ à tous les nœuds dans N.
  Leave()
  Passer à l'état *endormi*
fin si

Algorithme 12 $RemoveFromTree[k]$

si $Token.tree[k] \neq \perp$ **alors**
  pour tout j **avec** $Token.tree[j] = k$ **faire**
    $RemoveFromTree[j]$
  fin pour
  si $Token.corenodes[k]$ **alors**
    $Token.size \leftarrow Token.size - 1$
  fin si
  $Token.corenodes[k] \leftarrow \perp$
  $Token.tree[k] \leftarrow \perp$
fin si

2.4 Exemple détaillé

Considérons le réseau présenté dans la figure 2.11. Nous présentons dans cette section une exécution possible de l'algorithme sur ce réseau, avec un paramètre $K = 4$. Tous les nœuds commencent l'exécution en sommeil, et s'éveillent au bout d'une durée aléatoire ou à la réception d'un message. Au bout d'un moment, le nœud 3 s'éveille et exécute la procédure *wakeup* (algorithme 2.3). Le nœud 3 ayant des voisins, il crée un nouveau cluster d'identifiant $(3, 1)$ (ligne 4) et le jeton associé (ligne 2 à 6). Nous nous attachons dans un premier temps aux évènements qui

affectent ce cluster. Le nœud 3 devient le premier nœud de ce cluster par un appel à la fonction *JoinCore* (ligne 5 algorithme 4). La variable *nexto* est incrémentée (ligne 8), afin que le nœud 3 ne puisse plus jamais créer un jeton d'identifiant $(3, 1)$. Ensuite, chacun de ses voisins est invité à intégrer le cluster $(3, 1)$ via des messages $Recruit[3, 1]$ (ligne 9). Enfin, le jeton $Token_{(3,1)}$ est envoyé au nœud 2 choisi uniformément au hasard, qui devient le père du nœud 3 dans l'arbre couvrant du cluster $(3, 1)$ (ligne 10). Cela nous amène à la situation présentée dans la figure 2.12.

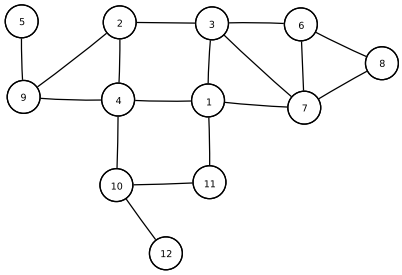

FIGURE 2.11 – 1

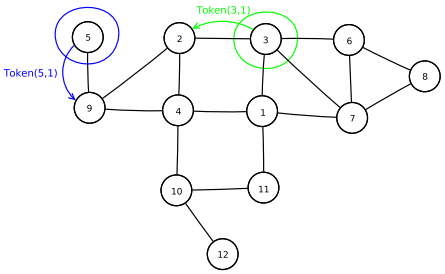

FIGURE 2.12 – 2

Les nœuds 1, 2, 6 et 7 reçoivent les messages $Recruit[3, 1]$. Étant non clusterisés, ils s'éveillent et exécutent la procédure *JoinOrdinary*, algorithme 5. Ces nœuds rejoignent le cluster $(3, 1)$ en tant que nœuds ordinaires, avec le nœud 3 pour père dans l'arbre couvrant.

Le nœud 2 reçoit ensuite le jeton $Token_{3,1}$. Étant un nœud ordinaire, et puisque $Token_{3,1}.size = 1$, il est recruté dans le cœur (ligne 20 à 29 algorithme 10). Il invite les nœuds 9, 4 et 3 à rejoindre le cluster en tant que nœuds ordinaires (figure 2.13).

On remarque que si le nœud 2 avait reçu les messages *Recruit* et *Token* dans l'ordre inverse, la configuration aurait tout de même été la même.

Les nœud 3 et 9 étant des nœuds de cœur, ils ignorent le message $Recruit[3, 1]$ qui leur est adressé (ligne 1 algorithme 5). Le nœud 4 rejoint le cœur du cluster, invite les nœuds 1, 9 et 10 et transmet le jeton au nœud 9 (figure 2.14).

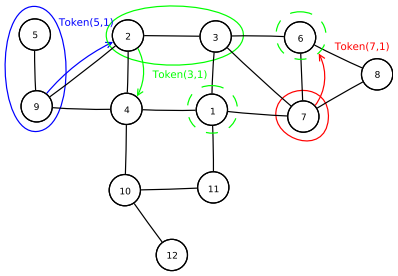

FIGURE 2.13 – 1

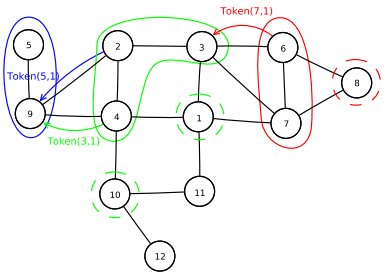

FIGURE 2.14 – 2

Le nœud 10 rejoint le cluster, le nœud 1 ignore le message $Recruit[3, 1]$.

Lorsque le nœud 9 reçoit le jeton $Token_{3,1}$, il fait partie du cœur du cluster $(5, 1)$, sa variable *complete* vaut **faux** et $Token_{3,1}.size < K$. Par conséquent, il initie la destruction du cluster $(3, 1)$ à l'aide d'un message $Delete[3, 1]$ (lignes 30 à 33 de l'algorithme 10). Cela amène à la situation de la figure 2.15.

Le nœud 4 reçoit ce message $Delete[3, 1]$. Il exécute l'algorithme 7. Puisque l'expéditeur est son père dans l'arbre couvrant du cluster $(3, 1)$, il quitte le cluster (fonction Init), envoie un

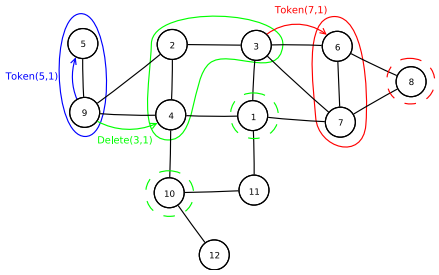

FIGURE 2.15 – 1

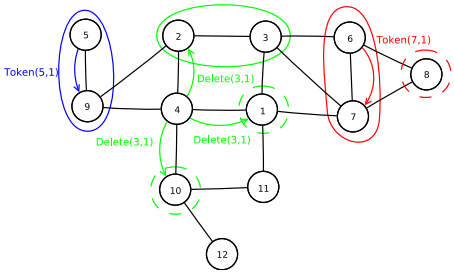

FIGURE 2.16 – 2

messages $Delete[3,1]$ à tous ses voisins puis s'endort (figure 2.16). Un vague de messages *Delete* est ainsi propagée sur l'arbre, entrainant la destruction du cluster $(3,1)$ (figure 2.17 et 2.18).

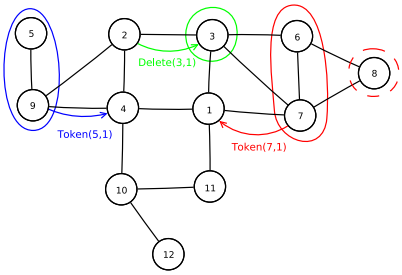

FIGURE 2.17 – 1

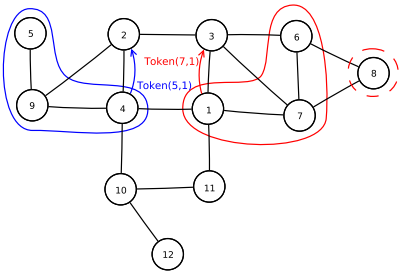

FIGURE 2.18 – 2

En parallèle, deux autres clusters sont construits de manière similaire. On remarque que, lorsque le nœud 2 reçoit le jeton $Token_{5,1}$ (figure 2.13), puisque $(5,1) > (3,1)$, le nœud exécute les lignes 35 à 38 de l'algorithme 10. Le jeton est simplement renvoyé au nœud 9 (figure 2.14). De la même façon, le jeton $Token_{7,1}$ est renvoyé à l'expéditeur entre les figures 2.14 et 2.15.

Une fois le cluster $(3,1)$ détruit, les clusters $(5,1)$ et $(7,1)$ ont la place pour devenir complets (figure 2.18 et 2.19). Les jetons continuent à circuler dans le réseau, en mettant à jour les variables des nœuds, et en particulier en mettant la variable *complete* à **vrai** sur les nœuds de cœur visités (algorithme 10 et 14).

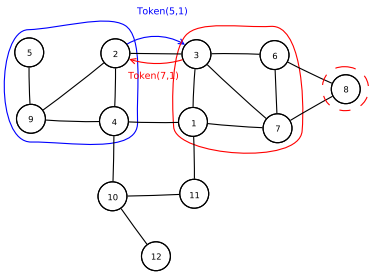

FIGURE 2.19 – 1

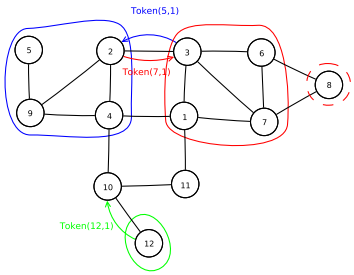

FIGURE 2.20 – 2

Un troisième cluster apparaît à l'éveil du nœud 12 (figure 2.20), et grandit jusqu'à posséder trois nœuds de cœur (figure 2.21 et 2.22). Lorsque le nœud 1 reçoit le jeton $Token_{12,1}$ (figure 2.22), la variable $complete_1$ a la valeur **vrai**. Ainsi, il exécute les lignes 35 à 38 de l'algorithme

10. Le jeton est renvoyé à l'expéditeur (figure 2.23).

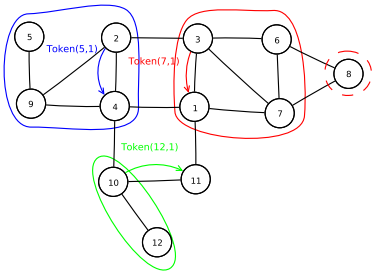

FIGURE 2.21 – 1

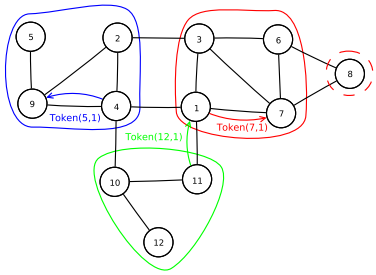

FIGURE 2.22 – 2

Le nœud 8, nœud ordinaire du cluster $(7,1)$, n'est pas présent dans la variable $Token_{7,1}.tree$ avant qu'il ne reçoive le jeton $Token_{7,1}$ (figure 2.23 et 2.24).

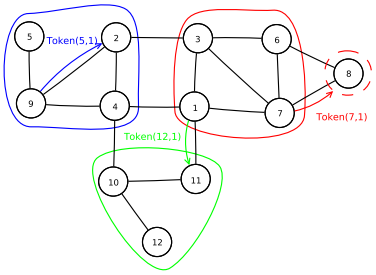

FIGURE 2.23 – 1

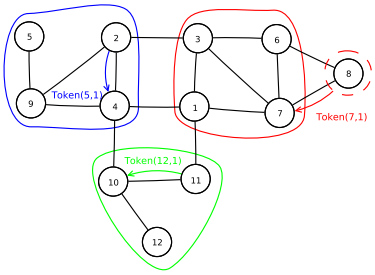

FIGURE 2.24 – 2

Les jetons continuent ensuite leur circulation, le clustering n'évoluera plus en l'absence de changement topologique.

2.5 Conclusion

Dans ce chapitre, nous avons présenté un algorithme distribué de clustering orienté taille. Chaque cluster possède une partie dominante connexe, appelé son cœur, dont la taille est bornée par un paramètre de l'algorithme.

Le cœur de chaque cluster est construit par un message particulier, appelé $Token$, qui suit une marche aléatoire dans le réseau en recrutant un à un les nœuds non-clusterisés qu'il rencontre. Ce processus de recrutement permet de mettre en place un arbre couvrant de chaque cluster.

Lorsqu'un changement topologique survient dans le système, le cluster où ce changement a lieu peut être réparé à l'aide de cet arbre. Cette réparation est locale : seul le cluster où le changement survient est modifié, et éventuellement ses clusters adjacents.

La présence d'un arbre couvrant au sein de chaque cluster nous permet par la suite d'établir des communications inter-clusters. De plus, le caractère très local de cet algorithme le rend adapté à un passage à l'échelle. En nous appuyant sur cet algorithme, nous présentons au chapitre suivant un algorithme capable d'organiser les réseaux de grande taille via un clustering hiérarchique.

Chapitre 3

L'Ensemble des clusters comme système distribué : mécanismes pour la cohérence, l'atomicité et la communication

3.1 Introduction

Pour construire un clustering hiérarchique, nous devons être capable de regrouper les clusters à l'aide d'un algorithme de clustering. Pour cela, les clusters doivent être en mesure d'exécuter l'algorithme de clustering présenté au chapitre précédent. Nous cherchons donc à mettre en place les mécanismes permettant aux clusters de communiquer entre eux, et d'exécuter l'algorithme distribué précédent. De manière plus générale, dans ce chapitre nous mettons en place les mécanismes permettant aux clusters de former un nouveau système distribué, capable d'exécuter n'importe quel algorithme distribué $\mathcal{A}$ conçu pour notre modèle.

Nous présentons, dans ce chapitre, un algorithme distribué qui calcule des clusters, et qui émule un système distribué dont les nœuds sont ces clusters, exécutant un algorithme distribué $\mathcal{A}$. Les clusters possèdent les variables locales définies par l'algorithme $\mathcal{A}$, et ont la capacité de communiquer avec leurs voisins par des messages. Les clusters sont ainsi les nœuds d'un nouveau système distribué.

Simuler un nœud virtuel présente un certain nombre de difficultés. En tant que nœud virtuel, le cluster doit disposer des variables utilisées par l'algorithme distribué $\mathcal{A}$. La véritable difficulté n'est pas tant de stocker ces variables, que de s'assurer que les nœuds du cluster s'accordent sur la valeur de ces variables. En effet, tous les nœuds composant un cluster doivent avoir un comportement unique pour ce qui concerne l'émulation du nœud virtuel, ce qui commence par un consensus sur la valeur des variables du nœud virtuel.

Puisque les clusters représentent les nœuds d'un système distribué, ils doivent disposer d'une capacité de communication avec les clusters qui les entourent. Un cluster devra donc, en tant que nœud virtuel, être capable d'envoyer un message à n'importe lequel de ses clusters voisins, et aussi être capable d'en recevoir. Cette capacité de communication nécessite en amont une détection des clusters adjacents.

Lors de la réception d'un message inter-cluster, un cluster devra appliquer les instructions prévues par l'algorithme $\mathcal{A}$ en réaction à ce message. Lors de cette réaction, il ne doit pas y avoir d'incohérence parmi les nœuds composant le cluster. En effet, le nœud virtuel doit avoir

une réaction unique, et reconstruire une exécution cohérente de l'algorithme $\mathcal{A}$ (en particulier, un ordre total sur la réception des messages).

Enfin, dans l'algorithme du chapitre 2, des clusters peuvent apparaître et disparaître durant l'exécution. De même, durant la croissance d'un cluster, celui-ci peut avoir de nouveaux clusters voisins. Le système simulé par les clusters est par conséquent dynamique, même si le système initial ne subit aucun changement topologique. L'algorithme $\mathcal{A}$ devra donc être tolérant aux changements topologiques. Les clusters doivent être capable de détecter les changements topologiques du système distribué qu'ils constituent. Lorsqu'un cluster détecte un changement topologique, les nœuds qui le composent doivent, là encore, s'accorder sur une unique réaction.

3.1.1 Caractéristiques de l'émulation

Nous proposons un algorithme de clustering donnant à chaque cluster la capacité d'émuler le comportement d'un nœud virtuel. Cela nous amène à considérer un nouveau système distribué, composé des éléments suivants :

- un graphe de communication ayant les clusters comme nœuds, et la relation d'adjacence entre les clusters comme arêtes,
- un algorithme distribué $\mathcal{A}$ écrit pour un système suivant le modèle présenté dans la section 1.4, et résistant aux changements topologiques,
- un ensemble de variables présentes sur chaque nœud et défini par $\mathcal{A}$,
- un ensemble de messages en transit entre ces nœuds virtuels.

Lors d'une transition de cet algorithme, la configuration du système est modifiée. En particulier, les variables des nœuds virtuels, ainsi que les messages en transit entre ces nœuds virtuels, peuvent être modifiées. Ces transformations doivent correspondre exactement à des transitions de l'algorithme $\mathcal{A}$, exécuté directement par le système virtuel, ou à des changements topologiques autorisés par le modèle.

De plus, l'algorithme induit un comportement du système virtuel qui doit correspondre au modèle présenté dans la section 1.4. En particulier, chaque message en transit dans le système virtuel doit finir par être reçu. De même, tout changement topologique modifiant le graphe de communication du système virtuel doit finir par être détecté.

Enfin, si l'algorithme $\mathcal{A}$ utilise la notion de nœuds endormis, l'algorithme doit garantir qu'ils ne restent pas tous en sommeil indéfiniment. En effet, une exécution maintenant tous les nœuds endormis infiniment longtemps résoudrait trivialement la spécification. La spécification formelle de ce problème se trouve à la section 4.4.

3.2 Description générale de l'algorithme

Simuler le comportement d'un nœud virtuel avec un ensemble de nœuds pose plusieurs difficultés.

En particulier, il est nécessaire d'assurer la cohérence de comportement des différents nœuds du cluster. Pour résoudre ce problème, plusieurs solutions sont possibles. Par exemple, chaque nœud du cluster peut posséder une instance de ces variables. C'est le choix qui est fait dans [DGL+04], où chaque nœud impliqué dans la simulation d'un nœud virtuel possède une instance des variables de ce nœud virtuel. Ce choix nécessite la présence d'un mécanisme assurant que les différents nœuds d'un même cluster ont les mêmes valeurs pour ces variables. Dans [DGL+04] la participation d'un nœud à l'émulation d'un nœud virtuel commence par une phase de négociation. Lors de cette phase, ce nœud demande la valeur actuelle des variables du nœud virtuel, et se met à l'écoute des messages destinés au nœud virtuel. Les messages concernant le nœud virtuel

disposent d'un horodatage basé sur des horloges de Lamport [Lam78], qui permet aux différents nœuds de traiter les événements successifs affectant le nœud virtuel tous dans le même ordre. Lorsque le nœud rejoignant l'émulation obtient la valeur des variables, il peut ainsi simuler la réception des messages destinés au nœud virtuel qu'il a reçu durant ce laps de temps. Les instances des variables du nœud virtuel ont ainsi une valeur commune, la réaction à un évènement étant exécutée par tous les nœuds participant à la simulation, et l'horodatage permettant de ne pas émettre plusieurs fois le même message.

Une seconde solution est d'élire un nœud représentant le nœud virtuel et détenant ses variables. Ce nœud possédera le privilège d'effectuer les actions du nœud virtuel et d'émettre les messages correspondants. De cette façon, il n'existe qu'une seule instance des variables du nœud virtuel, et un événement affectant ce nœud virtuel engendre une unique réaction du nœud virtuel. Les algorithmes de clusterings hiérarchiques [Bea03] ou [DT09] utilisent de tels mécanismes, chaque cluster étant maintenu par un clusterhead, qui exécute l'algorithme de clustering pour le niveau supérieur en dialoguant directement avec les clusterheads des cluster voisins. L'inconvénient, dans un système dynamique, est que la déconnexion d'un tel clusterhead oblige à détruire le cluster, ou au minimum à réinitialiser l'émulation du nœud virtuel. En effet, seul le clusterhead possédait les variables du nœud virtuel.

Dans l'algorithme de clustering présenté dans le chapitre 2, les clusters sont construits en se passant de clusterhead, et avec un jeton représentant un privilège de recrutement. Nous choisissons que, à chaque étape, le nœud possédant le jeton représente le nœud virtuel. Ainsi, les variables du nœud virtuel sont stockées dans le jeton du cluster, et à chaque instant, c'est le nœud détenant le jeton qui a le privilège d'effectuer les actions du nœud virtuel. Cette solution permet ainsi de résoudre le problème de la cohérence des variables et des actions du nœud virtuel, tout en étant résistante aux changements topologiques. En effet, aucun nœud n'étant privilégié dans le cluster, une déconnexion n'empêche pas la simulation du nœud virtuel tant que le jeton reste disponible.

Les clusters étant les nœuds d'un système distribué, chaque nœud virtuel doit disposer d'une capacité de communication avec les nœuds virtuels adjacents. Cela nécessite de mettre en place des mécanismes permettant dans un premier temps de détecter ces nœuds virtuels adjacents, puis des mécanismes permettant l'envoi et la réception de message. Au chapitre 2, les nœuds ont connaissance de leur voisinage via une variable N, mise à jour par un service de niveau inférieur. Le jeton de chaque cluster porte donc une variable $Token.N$ représentant les nœuds voisins dans le système émulé, que nous maintenons à jour au cour de l'exécution de l'algorithme.

La détection des clusters adjacents se fait en deux étapes, une locale au niveau des nœuds du système, et une globale au cluster collectée par le jeton. Ces deux mécanismes nécessitent d'introduire quelques variables supplémentaires, ainsi qu'un nouveau type de message :

- Au niveau des nœuds : *gate*, un tableau indexé par les voisins du nœud.
- Au niveau des jetons :
  - $Token.N$: l'ensemble des clusters voisins ;
  - $Token.gateway$: un tableau indexé par N indiquant pour chaque cluster voisin un lien de communication à emprunter pour l'atteindre.
- Nouveau message : $Cluster[x]$, indiquant que l'expéditeur a rejoint le cluster x.

Chaque nœud va enregistrer dans quel cluster se situe chacun de ses voisins. Pour ce faire, chaque fois qu'un nœud i change de cluster, c'est-à-dire qu'il modifie sa variable $cluster_i$, il envoie un message $Cluster[cluster_i]$ à tous ses voisins. Ainsi, chaque fois qu'un nœud reçoit un message $Cluster[x]$ venant d'un nœud e, il enregistre dans la variable *gate* que le nœud e fait désormais partie du cluster C_x. Ce mécanisme permet donc à tous les nœuds de savoir en permanence (moyennant les délais de transmission des messages) à quel cluster ses voisins appartiennent.

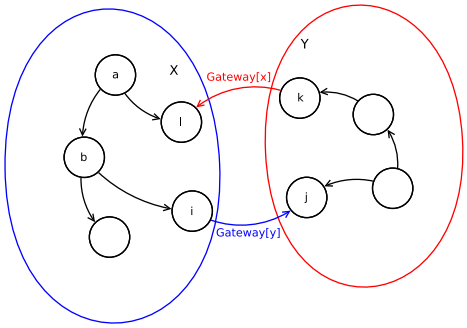

FIGURE 3.1 – Difficulté d'un *Gateway* asymétrique

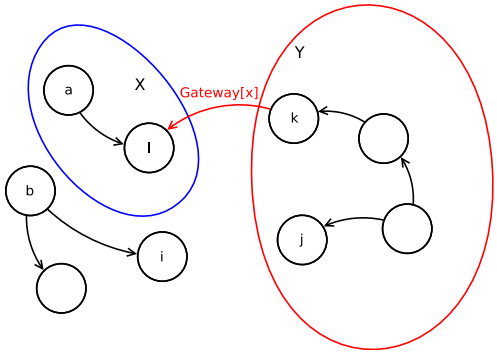

FIGURE 3.2 – Détection unilatérale d'un changement topologique

Parallèlement à cela, le jeton collecte au cours de sa circulation ces informations pour les transformer en une vision globale à l'échelle du cluster. En effet, lorsque le jeton $Token_x$ visite un nœud j, une exploration de la table *gate* de ce nœud permet de détecter d'éventuels clusters adjacents, et d'enregistrer cette information dans la variable $Token.N$ du jeton. Cependant, pour détecter correctement les éventuelles pertes de connexion entre clusters, nous cassons la symétrie dans la détection du voisinage. En effet, si les clusters C_1 et C_2 communiquent via deux canaux différents, la rupture du canal de C_1 à C_2 conduira C_1 à exécuter les instructions associées à une perte de lien, alors que C_2 poursuivra sans détecter de changement topologique. Ce phénomène est illustré dans l'exemple 8. Ainsi, dans un couple de clusters voisins, seul celui de plus grand identifiant est autorisé à détecter l'autre comme voisin. La détection inverse se fera lors de la réception du premier message et sera détaillée plus loin.

Exemple 8. *Supposons que nous n'imposions pas qu'un couple de clusters voisins choisisse le même lien de communication comme passerelle. La situation décrite dans la figure 3.1 est alors légale. Le cluster C_x utilise l'arête (i, j) comme passerelle ($Token_x.gateway[y] = (i, j)$), et le cluster C_y utilise l'arête (k, l) comme passerelle ($Token_y.gateway[x] = (k, l)$).*

Si le lien de communication (a, b) disparaît, alors une partie du cluster C_x quitte ce cluster, rendant inaccessible la passerelle $Token_x.gateway[y]$ (figure 3.2). Le nœud virtuel x va ainsi détecter la perte de connexion avec le nœud y, et exécuter les instructions associées prévues par l'algorithme $\mathcal{A}$. Le nœud virtuel y ne va quant à lui jamais détecter de perte de connexion, puisqu'il peut encore joindre le cluster C_x. Cela amène le système distribué du niveau supérieur dans une configuration illégale, avec une perte de connexion détectée d'un côté seulement.

Considérons un nœud j appartenant à un cluster C_x, et ayant un nœud voisin k dans le cluster C_y, avec $x > y$. Lorsque le nœud k a rejoint le cluster C_y, il en a informé j à l'aide d'un message $Cluster[y]$. À la réception de ce message, le nœud j a enregistré l'information, en affectant $gate[k] \leftarrow y$. Lorsque le jeton $Token_x$ visite le nœud j, il détecte que le nœud k fait partie d'un autre cluster C_y grâce à la variable *gate*. Ainsi, x et y sont deux nœuds virtuels adjacents, et puisque $x > y$, cette information est enregistrée dans le jeton. L'identifiant y est ajouté a $Token_x.N$. Le lien de communication (j, k) est par ailleurs un lien de communication entre ces deux clusters. Il devient un lien privilégié et est enregistré comme tel : $Token_x.gateway[y] = (j, k)$. Ce lien sera utilisé pour faire passer tous les messages entre les nœuds virtuel x et y.

Exemple 9. *Considérons la configuration présentée dans la figure 3.3. Dans cette configuration, le nœud i possède un voisin j à l'extérieur du cluster C_x. Quand le nœud j rejoint le cluster*

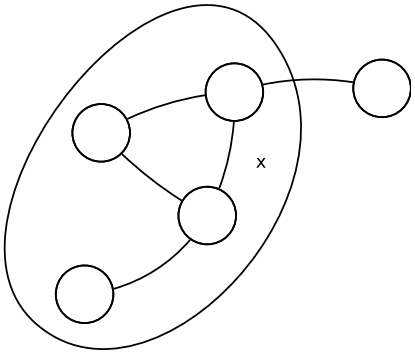

FIGURE 3.3 – Découverte du voisinage

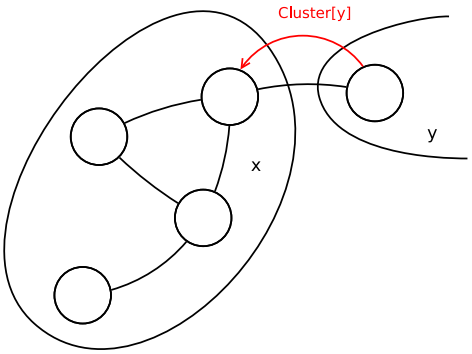

FIGURE 3.4 – Détection locale du voisinage

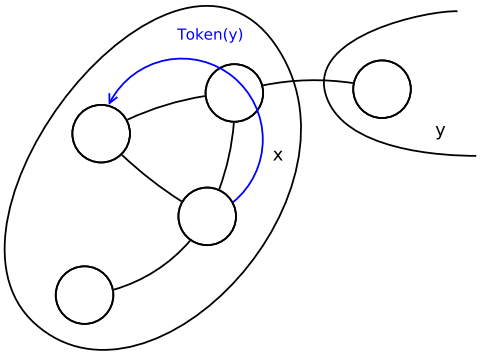

FIGURE 3.5 – Enregistrement de $gateway[y]$ dans $Token_x$

C_y, il en informe tous ses voisins, et en particulier le nœud i, à l'aide d'un message $Cluster[y]$ (figure 3.4). Le nœud i reçoit ce message et enregistre l'information ($gate_i[j] \leftarrow y$).

Ensuite, lorsque le jeton $Token_x$ visite le nœud i, si $x > y$, alors y est enregistré comme voisin dans $Token_x.N$, et $Token_x.gateway[y] = (i, j)$.

La détection réciproque aura lieu lors de la réception par le nœud virtuel y du premier message que lui enverra le nœud virtuel x.

Dès qu'un nœud virtuel voisin est enregistré dans $Token_x.N$, et qu'un lien privilégié vers ce voisin est enregistré dans $Token_x.gateway$, le nœud virtuel x a la capacité d'envoyer des messages à ce voisin. Le cluster C_x peut envoyer un message au cluster C_y au travers du lien $Token_x.gateway[y]$. Les actions du nœud virtuel sont toujours effectuées par le nœud physique détenant le jeton. Or, le jeton possède un arbre couvrant du cluster, et connaît un lien de communication privilégié $gateway[y]$. Par conséquent, le nœud physique détenant le jeton $Token_x$ est en mesure de calculer un chemin menant à un nœud appartenant au cluster C_y.

Pour mettre en place cette communication inter-clusters, nous introduisons une nouvelle variable sur les nœuds, ainsi qu'un nouveau type de message :

- $Transmit[msg, path, em, dst]$: message contenant un message en transit entre deux nœuds virtuels, le chemin à suivre jusqu'à un nœud du cluster de destination, l'identifiant du nœud virtuel émetteur et l'identifiant du nœud virtuel destinataire.
- $listmsg$: variable de nœud contenant une liste de messages virtuels en attente de traite-

ment.

Ainsi lorsqu'un nœud virtuel x est amené à envoyer un message à un nœud virtuel voisin y, le nœud détenant le jeton $Token_x$ calcule un chemin $path$ vers un nœud appartenant au cluster C_y, à l'aide de l'arbre couvrant $Token_x.tree$ et du lien $Token_x.gateway[y] = (j, k)$. Ce nœud physique encapsule le message du système virtuel dans un message $Transmit[msg, path, x, y]$ qui sera envoyé au premier nœud du chemin $path$. Ensuite, tout nœud du cluster C_x recevant un tel message $Transmit$ le relaiera au prochain nœud du chemin $path$, jusqu'à ce que le message $Transmit$ soit reçu par le dernier nœud du chemin $path$, k. Si celui-ci fait toujours partie du cluster C_y, il stocke le message dans $listmsg_k$, accompagné des informations suivantes :

- le nœud virtuel émetteur du message x,
- le nœud ayant envoyé le message $Transmit$, j.

Ces informations permettront de choisir le lien (k, j) comme lien de communication privilégié $Token_y.gateway[x]$ lors de la réception du message par le nœud virtuel y.

La réception d'un message par un nœud virtuel est, quant à elle, effectuée lorsque le jeton visite le nœud où ce message est stocké. Ainsi, chaque fois qu'un nœud reçoit le jeton de son cluster, en plus des traitements vus dans la partie 2, il simule la réception de tous les messages en attente dans $listmsg$ par le nœud virtuel.

Lorsque le jeton $Token_y$ visite un nœud k, il traite les messages en attente dans $listmsg_k$. Ces messages sont stockés avec l'identifiant du cluster émetteur x, et avec l'identifiant du dernier nœud physique j ayant relayé le $Transmit$ contenant ce message. Cela signifie que l'arrête (j, k) a été sélectionnée comme lien de communication privilégié par x. Par conséquent, si ce nœud virtuel n'est pas connu du nœud virtuel y, il est ajouté à $Token_y.N$ et l'arête (k, j) est sélectionnée comme lien de communication privilégié dans $Token_y.gateway[x]$.

Cette façon de gérer les communications virtuelles garantit que les nœuds virtuels traitent une seule fois les messages de niveau supérieur.

Lorsqu'un changement topologique a lieu dans le système distribué, il peut avoir des répercussions sur le système distribué virtuel. En effet, une perte de connexion au sein d'un cluster peut amener ce dernier à être amputé d'une partie de ses nœuds. Si c'est le cas, le cluster peut perdre la connexion avec certains de ses voisins, ce qui amène un ou plusieurs changements topologiques dans le système distribué de niveau supérieur. Ces changements doivent être détectés. Comme précédemment, la marche aléatoire du jeton va finir par détecter qu'une partie du cluster l'a quitté. Lorsque cela arrive, l'arbre couvrant présent dans le jeton est mis à jour. Chaque fois qu'une branche est amputée dans cet arbre, le nœud détenant le jeton vérifie si cela rend certains liens $Token.gateway$ inaccessibles. Lorsque c'est le cas, le nœud simule les procédures de déconnexion pour le nœud virtuel, puis enlève les clusters concernés de $Token.N$ et $Token.gateway$.

Exemple 10. *Dans la figure 3.6, le cluster C_x a connaissance du cluster C_y : $y \in Token_x.N$ et $Token_x.gateway[y] = (k, l)$. Le lien (i, j) disparaît, initiant la destruction d'une partie du cluster C_x (figure 3.7). La passerelle (k, l) n'est alors plus accessible pour le cluster C_x, amenant un changement topologique dans le système distribué virtuel. Lorsque le nœud i reçoit le jeton $Token_x$, puisque le nœud j n'est plus accessible, l'arbre couvrant du cluster est mis à jour. C'est à ce moment là que le changement topologique dans le système distribué virtuel est détecté (figure 3.8).*

De plus, les évènements du système distribué de niveau supérieur sont traités de manière atomiques. En effet, ces évènements sont détectés durant le traitement d'un message $Token$. Les algorithmes locaux prévus par $\mathcal{A}$ en réaction à ces évènements sont exécutés l'un après l'autre durant le traitement de ce message. L'hypothèse d'atomicité de notre modèle nous garantit que le

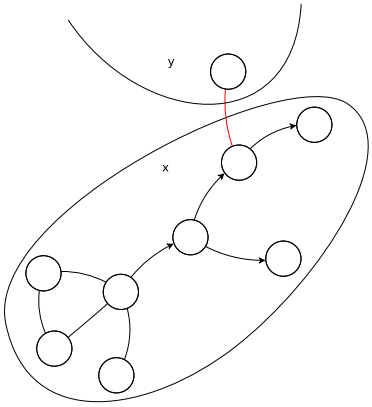

FIGURE 3.6 – Changement topologique

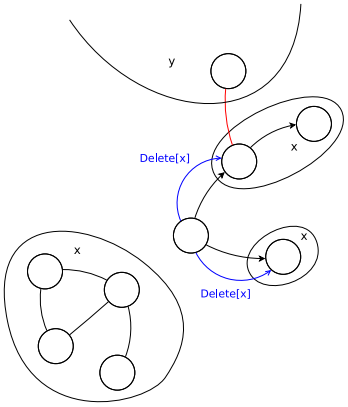

FIGURE 3.7 – Conséquence

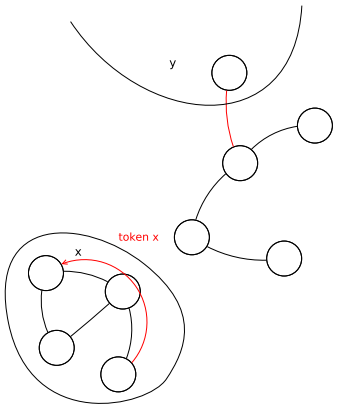

FIGURE 3.8 – Changement topologique au niveau supérieur

traitement d'un message $Token$ est atomique. Par conséquent, les algorithmes locaux de $\mathcal{A}$ sont également exécutés de façon atomique. De plus, tout évènement se produisant dans le système distribué de niveau supérieur pendant l'exécution d'un algorithme local de $\mathcal{A}$ n'entraînera une réaction du système de niveau supérieur qu'après la fin de cette exécution. En effet, les traitements affectant un même nœud virtuel sont effectués l' un après l'autre. Les traitements affectant deux nœuds virtuels différents sont quant à eux effectués durant le traitement de deux messages $Token$ différents, et ne sont donc pas simultanés d'après l'hypothèse d'atomicité de notre modèle.

3.3 Algorithme détaillé

Dans cette section, on présente les mécanismes permettant à un cluster de simuler le comportement d'un nœud virtuel. Cet algorithme reprend les variables et les procédures de l'algorithme présenté dans le chapitre 2, avec quelques ajustements.

En particulier, l'algorithme définit sur chaque nœud les mêmes variables que dans l'algorithme précédent :

- $cluster$,
- $core$,
- $nexto$,
- $father$,
- N.

L'algorithme définit également deux nouvelles variables sur les nœuds :

- $gate$: un tableau indexé par l'ensemble des nœuds voisins N, reflétant le cluster d'appartenance de chaque nœud adjacent ;
- $listmsg$: une liste de messages virtuels destinés au nœud virtuel $cluster_i$, en attente de traitement.

Cet algorithme reprend les messages $Recruit[x]$ et $Delete[x]$. Il introduit deux nouveaux types de messages, utilisés pour la détection du voisinage et pour la communication inter-clusters :

- $Cluster[x]$: indique que l'expéditeur a rejoint le cluster x ;
- $Transmit[msg, path, x, y]$: message encapsulant un message msg du système virtuel, ainsi qu'un chemin $path$ au sein du cluster émetteur x que ce message doit suivre pour atteindre le cluster y.

Le jeton porte les variables que possède son cluster en tant que nœud virtuel, ainsi qu'une variable représentant les nœuds voisins dans le système virtuel et les passerelles permettant d'y accéder.

Le jeton est ainsi modifié pour comporter les informations supplémentaires suivantes :

- $Token.status$: une structure regroupant les variables du nœud virtuel ;
- $Token.N$: l'ensemble des nœuds virtuels voisins ;
- $Token.gateway$: un tableau indexé par $Token.N$ indiquant quel lien utiliser pour joindre un cluster adjacent.

L'algorithme $\mathcal{A}$ exécuté par le système émulé utilise les variables des nœuds virtuels, c'est-à-dire contenues dans les jetons : $Token.status$.

3.3.1 Différence dans les traitements

Cet algorithme reprend toutes les procédures de l'algorithme présenté dans le chapitre 2. Cependant, certaines de ces procédures impactent la variable $listmsg$, et doivent donc être modifiées. En effet, sur le nœud i, la variable $listmsg_i$ contient des messages en attente de réception, à destination du nœud virtuel $x = cluster_i$. Ainsi, si le nœud quitte le cluster x, ces messages

ne pourront plus être transmis à leur destinataire. Ils doivent donc être supprimés. La procédure *Leave* (algorithme 6) est modifiée pour vider cette liste. La procédure *à la réception d'un message Token* (algorithme 10) est également modifiée. En effet, lorsqu'un nœud ordinaire devient nœud de cœur d'un autre cluster (ligne 20 à 27), la variable *listmsg* est nettoyée par un appel à la fonction *Leave*.

3.3.2 Détection du voisinage

La détection du voisinage est découpée en deux mécanismes. Le premier consiste, pour chaque nœud, à obtenir une vision locale des clusters voisins. La seconde consiste à collecter une vision globale à l'échelle du cluster, en utilisant la circulation aléatoire du jeton.

La vision locale est constituée à l'aide des messages $Cluster[x]$. En effet, chaque fois qu'un nœud modifie sa variable *cluster*, il en informe ses voisins en leur envoyant des messages $Cluster[cluster_i]$. Il enverra aussi de tels messages à chaque passage de son jeton. Les fonctions *JoinCore*, *JoinOrdinary*, *Leave* et *à réception d'un message Token* sont donc modifiées, afin d'envoyer un message $Cluster[cluster_i]$ à chaque voisin du nœud i l'exécutant.

Chaque nœud informe ainsi régulièrement ses voisins de son cluster d'appartenance.

A la réception d'un message $Cluster[x]$

Lorsqu'un nœud reçoit un message $Cluster[x]$ en provenance d'un voisin e, cela signifie que désormais le nœud e est membre du cluster x. Cette information est enregistrée dans la variable *gate*.

Algorithme 13 À la réception d'un message $Cluster[x]$ venant du nœud e

$gate[e] \leftarrow x$

La vision du voisinage à l'échelle du cluster est construite lors de la circulation du jeton, en collectant les informations présentes dans les variables *gate* de tous les nœuds du cluster. Dans l'algorithme vu au chapitre 2, la synchronisation entre les variables locales et celles du jeton est effectuée par la fonction *UpdateStatus*. Celle-ci est modifiée pour gérer également la mise à jour des variables $Token.N$ et $Token.gateway$, qui constitue la vision du voisinage à l'échelle du cluster.

Entre deux clusters, un canal de communication privilégié est choisi. Ce canal est ensuite utilisé lors des communications inter-clusters. Ce lien de communication, appelé passerelle, relie un nœud du premier cluster à un nœud du deuxième. Pour que les deux clusters s'accordent sur la passerelle utilisée, seul celui de plus grand identifiant peut choisir une passerelle.

Lorsqu'un nœud i reçoit le jeton de son cluster $Token_x$, la fonction *UpdateStatus* parcourt la variable *gate* du nœud, à la recherche de clusters d'identifiants plus petits. Si un cluster d'identifiant $y = gate[j]$ plus petit que x est trouvé, il est ajouté à $Token_x.N$, et le lien (i, j) est choisi comme passerelle, et enregistré dans $Token_x.gateway[y]$ (ligne 24 à 27, algorithme 14). Cet détection n'est effectuée que si le nœud appartient au cluster C_x.

La fonction *UpdateStatus* conserve son rôle de mise à jour de l'arbre couvrant du cluster (ligne 1 à 13, algorithme 14). Une fois l'arbre mis à jour, s'il est élagué, certaines passerelles $Token.gateway$ peuvent devenir inaccessibles. Dans ce cas, elle doivent être enlevées de $Token.gateway$. Les clusters associés ne sont plus voisins et doivent donc être enlevés de $Token.N$ (ligne 15 à 21 de la fonction *UpdateStatus*). Ceci est accompli en vérifiant, pour chaque passerelle, si elle est encore accessible par un chemin dans l'arbre couvrant. Cette détection n'est effectuée que sur les nœuds appartenant au cluster C_x.

Algorithme 14 *UpdateStatus*() (appelée pendant le traitement d'un message $Token_x$)

pour tout k with $Token.tree[k] = i$ **faire**
  si $k \notin N$ **alors**
    $RemoveFromTree[k, Token.topology]$
  fin si
fin pour
si $cluster \neq x \wedge Token.tree[i] \neq \perp$ **alors**
  $RemoveFromTree[k, Token.topology]$
fin si
si $cluster = x \wedge Token.tree[i] = \perp$ **alors**
  Envoyer $Delete[x]$ à tous les nœuds dans N
  Leave()
  Passer à l'état *endormi*
fin si
si $cluster = x$ **alors**
  pour tout y **avec** $Token.gateway[y] \neq \perp$ **faire**
    $(l, m) \leftarrow Token.gateway[y]$
    si $(Token.tree[l] = \perp) \vee [(l = i) \wedge ((m \notin N) \vee (gate[m] \neq y)]$ **alors**
      $Token.gateway[y] \leftarrow \perp$
      enlever y de $Token.N$
      {Il y a un changement topologique au niveau supérieur.}
      $\mathcal{A}$.*à perte de la connexion vers le nœud* y
    fin si
  fin pour
  pour tout $k \in N$ avec $gate[k] \neq (\perp, 0)$ et $x > gate[k]$ **faire**
    $Token.gateway[gate[k]] \leftarrow (i, k)$
    ajouter $gate[k]$ à $Token.N$
  fin pour
fin si

La détection réciproque, par les clusters de plus petit identifiant, est effectuée lors de la réception du premier message inter-cluster, et sera expliquée dans la section suivante.

Les variables $Token.N$ et $Token.gateway$ sont ainsi complétées en parallèle, un nœud virtuel y étant enregistré dans $Token_x.N$ en même temps qu'une passerelle menant au cluster C_y est ajoutée dans $Token_x.gateway[y]$.

3.3.3 Communication entre clusters

Lorsqu'un cluster C_x a détecté un cluster C_y, la variable $Token_x.N$ contient l'identifiant C_y, et $Token_x.gateway[y]$ contient un lien de communication entre un nœud de C_x et un nœud de C_y.

Les actions d'un nœud virtuel x sont toujours effectuées par le nœud détenant le jeton $Token_x$. Ce nœud a ainsi accès à toutes les informations présentes dans le jeton, en particulier l'arbre $Token_x.tree$, ainsi que les passerelles $Token_x.gateway$. Ces informations permettent l'établissement d'une communication inter-cluster vers un nœud virtuel y présent dans $Token_x.N$. Pour ce faire, le nœud détenant le jeton $Token_x$ calcule un chemin jusqu'à la passerelle $Token_x.gateway[y]$ à l'aide de l'arbre $Token_x.tree$. Ce chemin, complété par la passerelle, mène jusqu'à un nœud m du cluster y. Le chemin est représenté sous la forme d'une liste, manipulée avec les primitives usuelles $Head$ et $Tail$. La procédure $Head$ retourne le premier élément de la liste, la primitive $Tail$ retourne quand à elle la liste privée de son premier élément.

Le message virtuel est donc encapsulé dans un message $Transmit$, accompagné du chemin jusqu'à m, qui est envoyé au premier nœud du chemin. Le message $Transmit$ contient également les identifiants du nœud virtuel émetteur et du nœud virtuel destinataire. La procédure $Encapsulate$ (algorithme 15) construit ce message $Transmit$, qui fournit aux nœuds virtuels la capacité de communiquer avec leurs voisins.

Algorithme 15 $Encapsulate(msg, y)$ (appelée pendant le traitement d'un message $Token_x$)

$(l, m) \leftarrow Token.gateway[y]$
$path \leftarrow$ Nouvelle liste vide
$Push(m, path)$
$path \leftarrow Concat(ComputePath(l), path)$
$lowlevmsg \leftarrow Transmit[msg, Tail(path), x, y]$
Envoyer $lowlevelmsg$ à $Head(path)$)

Algorithme 16 $ComputePath(msg, y)$ (appelée pendant le traitement d'un message $Token_x$)

$path \leftarrow$ Nouvelle liste vide
$NoeudCourant \leftarrow l$
$Push(NoeudCourant, path)$
tant que $NoeudCourant \neq i$ **faire**
  $NoeudCourant \leftarrow Token.tree[NoeudCourant]$
  $Push(NoeudCourant, path)$
fin tant que
Retourner $path$

Ce message $Transmit$ est ensuite relayé le long du chemin. Ainsi, lorsqu'un nœud reçoit un message $Transmit[msg, path, x, y]$, si le chemin est non vide et si le premier nœud du chemin $path$ est son voisin, il lui envoie le message $Transmit$ après l'avoir retiré de la tête de $path$ (ligne 4 de l'algorithme 18). Si le chemin est vide, le nœud doit être le dernier du chemin, et doit donc faire partie du cluster destinataire C_y. Si c'est bien le cas, le message est stocké dans la

variable *listmsg*, avec l'identifiant du cluster émetteur x et l'identifiant du dernier nœud ayant relayé le $Transmit$ (ligne 2 de l'algorithme 18). Dans tous les autres cas, le message $Transmit$ est simplement détruit. En effet, cela signifie que le chemin n'est plus valide. Un changement topologique est en cours. Il est normal que les messages en transit sur ce lien retiré du système virtuel soient détruits.

Algorithme 17 À la réception d'un message $Transmit[msg, path, x, y]$ venant de e

si $(path = \emptyset) \wedge (cluster = y)$ **alors**
  Ajouter (msg, e, x) à *listmsg*
sinon si $((cluster = x) \wedge (Head(path) \in N))$ **alors**
  Envoyer $Transmit[msg, Tail(path), x, y]$ à $Head(path)$
fin si

Les messages virtuels sont reçus lors du passage du jeton. En effet, lorsqu'un nœud reçoit le jeton de son cluster, il lance une procédure appelée $TriggerUpperLevel$, qui simule la réception par le nœud virtuel de tous les messages en attente dans la liste *listmsg*. La procédure *À réception d'un message Token* est donc modifiée pour appeler $TriggerUpperLevel$ dans les cas où le jeton $Token_x$ est reçu par un nœud de son cluster C_x (cas 1 et 2, mise à jour des informations sur un nœud de cœur ou ordinaire).

Avant de lancer la réception d'un message virtuel, la fonction $TriggerUpperLevel$ effectue quelques vérifications sur la provenance du message et sur la connaissance du voisinage que possède le nœud virtuel. En effet, comme indiqué dans la section 3.3.2, la détection du voisinage est faite de manière asymétrique. Seul le cluster de plus grand identifiant peut sélectionner une arête en tant que passerelle, et ainsi ajouter un cluster à la liste de ses voisins. C'est lorsqu'il reçoit un premier message d'un cluster voisin d'identifiant supérieur qu'un nœud virtuel peut décider de l'arête à utiliser pour communiquer avec ce cluster. Ainsi, pour chaque message en attente, le nœud virtuel vérifie s'il a connaissance de l'expéditeur du message. Si l'expéditeur est connu, et si le lien passerelle par lequel le message de niveau supérieur est passé est le même que celui enregistré dans $Token.gateway$, alors le nœud simule la réception de ce message par le nœud virtuel (ligne 4 de $TriggerUpperLevel$). Si l'expéditeur n'est pas connu, le lien ayant amené le message $Transmit$ $((i, e))$ est ajouté comme passerelle avant d'effectuer la réception (ligne 6 à 10 algorithme 18). Si l'expéditeur est connu mais que la passerelle enregistrée dans $Token.gateway$ ne coïncide pas avec celle ayant été utilisée dans cette transmission, alors deux cas de figure sont possibles. On considère que le cluster de plus grand identifiant a raison sur le choix de la passerelle. Ainsi, si l'identifiant de l'expéditeur est supérieur à l'identifiant du cluster $Token.id$, cela veut dire qu'il y a eu un changement topologique, forçant l'expéditeur à changer de passerelle. Par conséquent, $TriggerUpperLevel$ simule une perte de connexion (pour prendre en compte d'éventuels messages de niveau supérieur perdus), puis simule la réception du message (ligne 11 à 17). Au contraire, si l'identifiant de l'expéditeur est inférieur, le message est simplement ignoré. En effet, si le cluster courant a changé de nœud passerelle, c'est qu'il a détecté une perte de connexion avec le cluster expéditeur. Dans ce cas, il y a un changement topologique, qui n'a pas encore été détecté par le cluster expéditeur. Les messages sont ignorés, car ils correspondent à un lien de communication qui a disparu.

3.3.4 Émulation du nœud virtuel

L'émulation du comportement d'un nœud virtuel par le cluster est exécutée par les fonctions $UpdateStatus$ et $TriggerUpperLevel$. La fonction $UpdateStatus$, gérant la détection du voisinage, sera amenée à lancer la procédure *À la perte de la connexion vers* de l'algorithme $\mathcal{A}$. La

Algorithme 18 $TriggerUpperLevel()$ (appelée pendant le traitement d'un message $Token_x$)

tant que $(listmsg \neq \emptyset)$ **faire**
  $(msg, e, y) \leftarrow Pop(listmsg)$
  si $Token.gateway[y] = (i, e)$ **alors**
    {Émuler la réception du message msg par le nœud virtuel}
    $\mathcal{A}$.à la réception de (msg,buffer)
  sinon si $Token.gateway[y] = \perp$ **alors**
    $Token.gateway[y] \leftarrow (i, e)$
    Ajouter C à $Token.N$
    {Émuler la réception du message msg par le nœud virtuel}
    $\mathcal{A}$.à la réception de (msg,buffer)
  sinon si $Token.gateway[y] \neq (i, e)$ et $x < y$ **alors**
    {Exécuter la procédure *à la perte d'une connexion*}
    $\mathcal{A}$.à la perte de la connexion vers y
    $Token.gateway[y] \leftarrow (i, r)$
    Ajouter y à $Token.N$
    {Émuler la réception du message msg par le nœud virtuel}
    $\mathcal{A}$.à la réception du message (msg,buffer)
  sinon si $Token.gateway[y] \neq (i, e)$ et $x > y$ **alors**
    ignorer msg
  fin si
fin tant que
tant que $(buffer \neq \emptyset)$ **faire**
  $(msg, dst) \leftarrow Pop(buffer)$
  $Encapsulate(msg, dst)$
fin tant que
si $(Token.T > 0)$ **alors**
  $Token.T \leftarrow Token.T - 1$
  si $(Token.T = 0)$ **alors**
    $\mathcal{A}.wakeup()$
  fin si
fin si

procédure $TriggerUpperLevel$ va quant à elle amorcer toutes les procédures *À la réception d'un message* de l'algorithme $\mathcal{A}$, ainsi que la procédure *À la perte de la connexion vers* dans le cas de la réception d'un message par une mauvaise passerelle, comme expliqué dans la section précédente. La fonction $TriggerUpperLevel$ va également gérer l'éveil aléatoire des nœuds virtuels, selon l'algorithme $\mathcal{A}$.

Les méthodes de l'algorithme $\mathcal{A}$ utilisent les variables du nœud virtuel, enregistrée dans $Token.status$, et utilisent la primitive *envoyer à* du niveau supérieur. Celle-ci permet à un nœud virtuel d'envoyer un message à un nœud virtuel voisin. Cette primitive $\mathcal{A}.envoyer\ msg\ à\ y(buffer)$ se traduit au niveau inférieur par l'enregistrement du message à envoyer dans une liste d'attente *buffer*. Cette liste est ensuite récupéré par la fonction $TriggerUpperLevel$ (ligne 22 à 24), qui les encapsule et les envoie en utilisant la fonction *Encapsulate* (algorithme 15).

Algorithme 19 $\mathcal{A}.envoyer\ msg\ à\ y(buffer)$

$Push((msg, y), buffer)$

Nous avons fait l'hypothèse que les nœuds sont initialement endormis, et s'éveillent après un temps aléatoire. Ce phénomène est simulé au niveau supérieur en décomptant les sauts du

jeton. Lors de la création du jeton, une variable T est initialisée avec une valeur entière choisie au hasard. Lorsque la fonction *TriggerUpperLevel* a traité tous les messages en attente, elle décrémente cette variable (ligne 27). Quand cette variable atteint zéro, *TriggerUpperLevel* simule le lancement de la procédure *wakeup* par le nœud virtuel (ligne 28 à 30).

Ces deux fonctions vont ainsi déclencher la simulation de toutes les procédures de l'algorithme $\mathcal{A}$ par le système virtuel.

3.4 Application au clustering hiérarchique

Dans cette section, nous proposons un algorithme distribué construisant un clustering hiérarchique par une approche *bottom-up*, et utilisant l'algorithme présenté dans les sections précédentes.

En effet, cet algorithme permet de construire un clustering du système et de former un nouveau système distribué, exécutant un algorithme distribué $\mathcal{A}$, dont les nœuds sont les clusters précédemment calculés. Si cet algorithme $\mathcal{A}$ est également un algorithme de clustering, des clusters de clusters sont formés.

De plus, si l'algorithme $\mathcal{A}$ choisi est l'algorithme décrit dans les sections précédentes, ces clusters de clusters forment un nouveau système distribué.

En choisissant d'exécuter, sur chaque nouveau système distribué engendré, l'algorithme présenté dans les sections précédentes, on obtient un algorithme distribué construisant une hiérarchie de clusters. Dans chaque niveau, les clusters émulent un système distribué, qui sera clusterisé par le même algorithme. Chaque nouveau système ainsi engendré comporte strictement moins de nœuds que le système du niveau inférieur, puisqu'il n'y a pas de cluster de taille 1. Ce processus se poursuit jusqu'à ce qu'un système composé d'un seul nœud soit engendré, représentant toute la structure hiérarchique. Lorsqu'il exécute la procédure d'éveil 2.3, ce nœud n'ayant aucun voisin, il se rendort.

Le nombre de niveaux de la hiérarchie est ainsi fixé automatiquement par l'algorithme, en fonction de la topologie du système, et du paramètre K fixé par l'utilisateur.

Des changements topologiques peuvent survenir dans le système. De plus, une exécution de cet algorithme de clustering peut induire des changements topologiques au niveau supérieur, par exemple lorsqu'un cluster est détruit (ce qui est prévu par l'algorithme du chapitre 2). L'algorithme de clustering utilisé dans chaque niveau hiérarchique étant le même que celui présenté dans le chapitre 2, ces changements topologiques subits ou induits seront gérés. De plus, un tel changement n'affectera qu'une partie du réseau : le cluster où il a lieu, et éventuellement les clusters adjacents dans son niveau hiérarchique.

Enfin, chaque cluster, dans chaque niveau logique, possède un arbre couvrant. Il en résulte l'organisation du réseau en un arbre couvrant modulaire.

3.5 Simulations

Dans cette section, nous évaluons l'efficacité de notre algorithme par des simulations. Nous voulons savoir si l'émulation de nœuds virtuels par les clusters se déroule efficacement. Pour cela, nous exécutons l'algorithme présenté dans ce chapitre, et nous exécutons également cet algorithme sur le système émulé par les clusters du premier niveau. Il en résulte un clustering à deux niveaux du système.

Pour écrire ces simulateurs, nous avons utilisé DASOR ([Rab09]), qui se présente sous la forme d'une bibliothèque C++. DASOR s'appuie sur un modèle a passage de messages proche

du nôtre. Cette bibliothèque définit en effet une classe nœud générique (dont on peut hériter) et définit les primitives d'envoi et de réception de message. DASOR fonctionne par rondes, donnant la priorité à chaque nœud l'un après l'autre, et en déclenchant la réception de tous les messages en attente à destination de ce nœud. Lorsque DASOR simule la réception d'un message par un nœud, il déclenche une fonction générique de réception de message, dont on peut hériter pour implémenter notre algorithme.

Pour implémenter notre algorithme, nous avons procédé aux choix techniques suivants :

- Nous définissons une classe *nœud virtuel*, définissant les variables de l'algorithme distribué du niveau supérieur.
- Les messages *Token* contiennent un objet de type *nœud virtuel*.
- La méthode *envoyer à* de la classe *nœud virtuel* ajoute le message à envoyer à un tampon d'envoi, accompagné de l'identifiant du nœud virtuel destinataire.
- Les méthodes *à la réception d'un message* et *wakeup* de la classe nœud virtuel renvoie le tampon d'envoi.
- La méthode *TriggerUpperLevel* émule la réception des messages en attente en appelant la méthode *à la réception d'un message* sur le nœud virtuel porté par le jeton *Token*, récupère le tampon d'envoi, puis encapsule chacun des messages contenu dans le tampon avant de les envoyer avec la méthode *envoyer à*.

De cette façon, la méthode *TriggerUppelLevel* de la classe des nœuds physiques envoie les messages avec la primitive de communication du système, et la méthode *TriggerUpperLevel* de la classe des nœuds virtuels utilise la méthode *envoyer à* qui remplit le tampon d'envoi. Cette façon de procéder nous permet d'implémenter l'algorithme de clustering hiérarchique présenté à la section 3.4.

Dans cette campagne de simulations, nous nous limitons à un clustering à deux niveaux logiques. Nous exécutons notre algorithme de clustering sur le système initial, et nous exécutons également notre algorithme sur le système distribué émulé.

Nous décidons d'exécuter ces campagnes de simulations sur des graphes *petits mondes* suivant le modèle de Watts et Strogatz, connu pour posséder des propriétés présentes dans de nombreux systèmes réels. Ce modèle possède deux paramètres : la taille et la densité du réseau. Nous exécutons des simulations avec un degré constant, tout en faisant évoluer la taille des réseaux. En effet, faire croître le degré nous amène à une explosion du nombre de messages, lors des envois à tous les voisins (messages *Recruit* et *Cluster*), que DASOR ne peut pas gérer. Un nouveau graphe est engendré pour chaque simulation.

Dans une première campagne de simulation, nous mesurons le temps de convergence de l'algorithme, c'est-à-dire le temps nécessaire pour que les clusters des deux niveaux logiques cessent d'évoluer. Cette étude nous permettra d'évaluer si notre algorithme est ou non capable de clusteriser des réseaux de grande taille en un temps raisonnable.

Dans une deuxième campagne de simulations, nous mesurons la durée des communications au sein du système émulé, c'est-à-dire le temps nécessaire pour qu'un message émis par un nœud du système émulé à destination d'un de ses voisins soit reçu. Pour mesurer cela, nous attendons que le système ait convergé. Ensuite, le nœud virtuel de plus petit identifiant envoie un message particulier à l'un de ses voisins, choisi uniformément au hasard. Lorsque le nœud virtuel reçoit et traite ce message, la durée de la communication est mesurée grâce à l'horloge du simulateur.

3.5.1 Temps de clusterisation

Pour la première campagne, le simulateur a été lancé avec les paramètres suivants :

- Taille du réseau : 100, 200, 500, 1000, 2000 et 10000 ;

– K : 3 ;
– Degré : 22.

L'algorithme de clustering à deux niveaux est exécuté jusqu'à obtenir une configuration où le clustering dans chaque étage n'évoluera plus. Le temps nécessaire pour atteindre une telle configuration est appelé temps de clusterisation.

Les résultats obtenus sont présentés dans la figure 3.9.

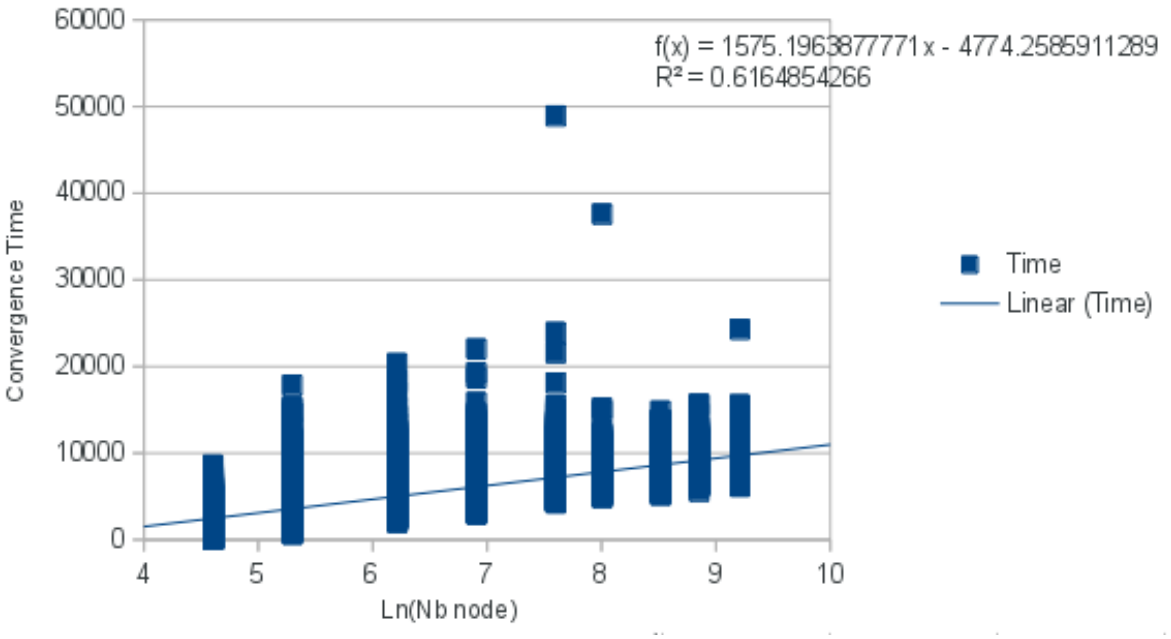

FIGURE 3.9 – Temps de clusterisation en fonction de ln(taille du réseau)

De par la méthode de construction des clusters (par marche aléatoire), le temps de clusterisation est une grandeur aléatoire, et on peut s'attendre à de grandes variations d'une simulation à l'autre pour une taille de système donnée. Ceci étant, le temps de clusterisation a clairement une croissance sous-linéaire en fonction de la taille du système, passant de 2400 étapes en moyenne pour clusteriser un réseau de 100 nœuds à 10000 étapes en moyenne pour clusteriser un réseau de 10000 nœuds.

Une régression logarithmique sur les temps de clusterisation nous confirme cette nette tendance avec un coefficient $R^2 = 0,62$, l'équation obtenue expliquant ainsi 62% de la variance des résultats.

Ces résultats confirment le caractère très local de cet algorithme.

Le temps de clusterisation augmente ainsi très lentement en fonction de la taille du système. En effet, d'après ces observations, multiplier la taille du système par deux augmente le temps de clusterisation de 1091 étapes, quelque soit la taille du système. En suivant cette tendance, on peut espérer clusteriser un système de 20000 nœuds en 11000 étapes, un système de 100000 nœuds en environ 13000 étapes et un système de 1000000 de nœud en environ 17000 étapes.

Cette algorithme de clustering est donc adapté à un passage à l'échelle.

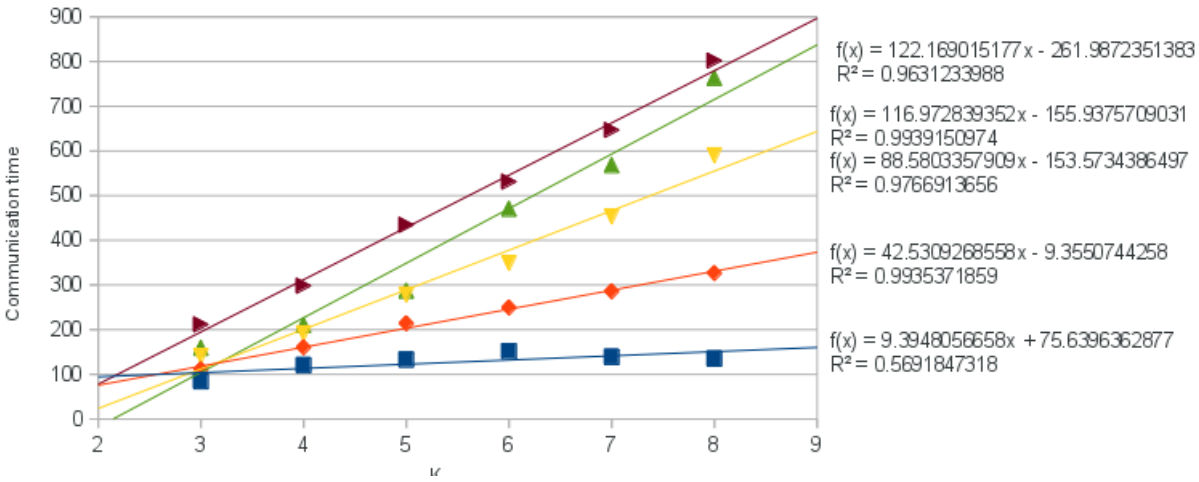

FIGURE 3.10 – Temps moyen de communication en fonction de K

3.5.2 Temps de communication

Lors d'une communication dans le système émulé, le message est relayé au sein du cluster émetteur jusqu'à être mis en attente sur un nœud du cluster destinataire. Le message est ensuite reçu lorsque le jeton du cluster destinataire visite ce nœud. Ce temps de communication peut ainsi se décomposer en deux parties distinctes :

- le temps durant lequel le message est relayé à l'intérieur du cluster émetteur : ce temps est a priori linéaire en la taille du chemin parcouru, et dépend donc de la taille des clusters et du paramètre K ;
- le temps d'attente dans le cluster destinataire avant le traitement au passage du jeton : ce temps dépend de la marche aléatoire du jeton.

Nous cherchons donc à mesurer l'impact de la taille du réseau et du diamètre des clusters sur les temps de communication dans le système virtuel.

Les simulations ont été exécutées avec les paramètres suivants :

- Taille du réseau : 100, 200, 500, 1000 et 2000,
- K : 3 à 10,
- Degré : constante = 22.

Nous obtenons les résultats présentés dans les figures 3.10 et 3.11.

Premièrement, on observe dans ces simulations que le temps d'attente sur sur un nœud du cluster de destination occupe une part prépondérante du temps de communication. En effet, pour une taille de réseau donnée, et pour une taille de cœur K donnée, on observe un écart type relatif de l'ordre de 1,6. Cela signifie qu'en moyenne, les temps de communication sont écartés du temps de communication moyen de 60% de cette valeur. Ce résultat nous confirme le caractère fortement asynchrone du système émulé.

Pour une taille de réseau fixée, nous effectuons une régression linéaire du temps de communication en fonction de la taille des cœurs K. Ces régressions donnent des coefficients R^2 entre 0,005 et 0,05, ce qui est peu surprenant au vu de la variance observée.

Cependant, en travaillant sur les durées moyennes de communication (figure 3.10), on observe une évolution nettement linéaire, avec un coefficient R^2 supérieur à $0,95$. Ainsi, le temps de communication moyen entre deux clusters évolue linéairement avec la taille de ceux-ci.

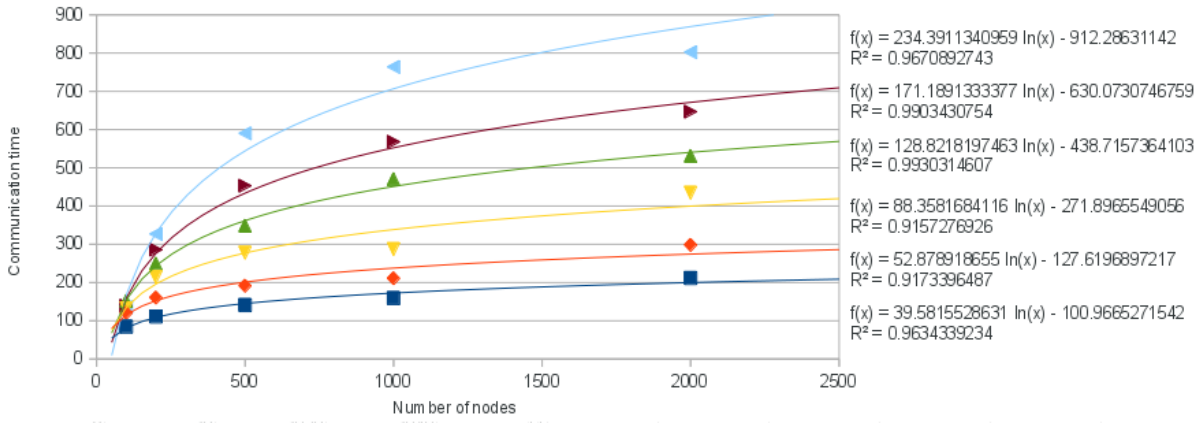

FIGURE 3.11 – Temps moyen de communications en fonction de la taille du réseau

Lorsque l'on observe l'évolution des temps de communication en fonction de la taille du système, on retrouve la même dispersion des valeurs. Les diverses régressions effectués donnent des coefficients R^2 très bas.

De nouveau, nous travaillons sur les valeurs moyennes (figure 3.11). Un régression logarithmique sur les temps moyens nous donne un coefficient R^2 de l'ordre de $0,92$. Les temps de communication augmentent donc lorsqu'on fait croitre la taille du système. Cependant, cette évolution est très lente (sous-linéaire), et n'empêche en rien le passage à l'échelle.

3.6 Conclusion

Dans ce chapitre, nous avons doté les clusters de la capacité d'émuler des nœuds virtuels. Les clusters définissent ainsi un nouveau système distribué exécutant l'algorithme distribué de notre choix.

La mise en place de cette couche d'abstraction nécessite que chaque cluster soit capable d'exécuter atomiquement un algorithme, de détecter dynamiquement les clusters adjacents et de communiquer avec ces derniers. Dans chaque cluster, c'est le jeton *Token* qui incarne le nœud virtuel. Cela permet d'obtenir une réaction unique et atomique des nœuds virtuels en s'appuyant sur le caractère atomique du traitement des messages *Token*.

Par la suite, en utilisant cette couche d'abstraction, nous avons présenté un algorithme distribué calculant un clustering hiérarchique imbriqué du système par une approche bottom-up. Les nœuds du système sont regroupés en clusters. Ces derniers, grâce à notre couche d'abstraction, forment un nouveau système distribué, sur lequel nous exécutons notre algorithme de clustering. Les nœuds du système émulé sont ainsi regroupés en clusters de clusters, ces derniers formant à leur tour un nouveau système distribué. Le processus se poursuit ainsi jusqu'à obtenir un système distribué composé d'un seul nœud représentant l'intégralité du réseau.

Ce procédé permet d'obtenir une structure hiérarchique imbriquée du système. Cette structure s'adapte aux changements topologiques d'une manière locale : lorsqu'un changement topologique survient dans un niveau logique, ce changement affecte uniquement le cluster de ce niveau logique où le changement à lieu, et éventuellement les clusters qui lui sont adjacents.

Chapitre 4

Démonstration

4.1 Introduction

Dans le chapitre 3, nous avons présenté un algorithme de clustering dont les clusters ont la capacité d'émuler un système distribué. Nous utilisons cet algorithme à la section 3.4 pour construire un algorithme de clustering hiérarchique.

Cet algorithme consiste à construire de manière distribuée un clustering du système. Ensuite, chaque cluster se comporte comme un nouveau système distribué qui sera clusterisé à son tour. Ces clusters de niveau supérieur émulent à leur tour des nœuds virtuels, formant un nouveau niveau hiérarchique. Ce processus se poursuit ainsi jusqu'à obtenir un niveau formé d'un seul nœud représentant l'ensemble du système : on obtient alors un clustering hiérarchique, dont chaque cluster de niveau k est composé de clusters de niveau $k-1$, chacun de ces clusters vérifiant la spécification du chapitre 2.

Pour démontrer le fonctionnement de ce processus, nous démontrons que le système abstrait défini par le clustering est bien un système distribué exécutant un algorithme distribué $\mathcal{A}$. Cela signifie en particulier que lors de toute exécution de notre algorithme, les modifications qui surviennent dans le système émulé sont exactement des transitions définies par $\mathcal{A}$ ou des changements topologiques autorisés par le modèle. Toute exécution de notre algorithme induit ainsi une exécution de l'algorithme $\mathcal{A}$ sur le système émulé.

Pour établir un clustering hiérarchique, nous choisissons comme algorithme $\mathcal{A}$ l'algorithme présenté aux chapitres 2 et 3. Cet algorithme nécessite, pour fonctionner correctement que les hypothèses formulées dans notre modèle soient vérifiées. En particulier, tout message émis finit par être reçu, et tout changement topologique finit par être détecté par les nœuds adjacents. Nous montrons donc que toute exécution de notre algorithme induit un comportement du système émulé conforme à notre modèle. Nous présentons une spécification du fait qu'un clustering émule un système distribué à la section 4.4.

Cette démonstration commence par la preuve que l'algorithme de clustering du chapitre 2 fonctionne correctement.

Nous construisons un clustering hiérarchique en exécutant l'algorithme du chapitre 3 sur le système émulé. Cependant, le mécanisme de communication entre les clusters présenté dans le chapitre 3 ne garantit pas que les messages émis seront reçus en un temps borné, ni que les messages seront reçus dans l'ordre d'émission. Par conséquent, l'algorithme de clustering doit être conçu pour un modèle à passage de messages asynchrone, sans hypothèse FIFO. Cela nous a conduit à adapter l'algorithme de [BKS12] en conséquence.

En effet, dans [BKS12], le but est de construire un algorithme de clustering, rendu auto-

stabilisant à l'aide de vagues de réinitialisation. Pour fonctionner, ces vagues de réinitialisation nécessitent que les communications se passent en un temps borné. Le modèle utilisé dans [BKS12] est donc synchrone, et nécessite que les canaux de communication soient FIFO.

L'absence de vagues de réinitialisations nous conduit à introduire un numéro d'ordre dans l'identifiant des clusters. De plus, nous voulons établir des communications inter-clusters. Les nœuds ordinaires doivent donc être présents dans l'arbre couvrant de leur cluster.

Pour ces raisons, et puisque le système émulé ne respecte pas l'hypothèse FIFO, nous commençons par présenter une démonstration originale de l'algorithme de clustering présenté au chapitre 2.

Par la suite, nous présentons une spécification formelle traduisant le fait que les clusters sont les nœuds d'un nouveau système distribué. Ce système étant composé de clusters qui peuvent être détruits, il est dynamique (même si le système initial ne l'était pas).

Nous démontrons ensuite que l'algorithme du chapitre 3 vérifie cette spécification, c'est-à-dire en particulier que les clusters calculés forment bien un nouveau système distribué conforme à notre modèle. Cela nous permet, à la section 4.7, de prouver que l'algorithme présenté à la section 3.4 construit bien un clustering hiérarchique s'adaptant aux changements topologiques.

4.2 Configurations et exécutions

Une configuration représente l'état du système, et est constituée des informations nécessaires pour décrire son évolution.

Définition 11. *Une configuration γ du système exécutant l'algorithme du chapitre 3 est la donnée des éléments suivants :*

- *le graphe de communication du système $G^\gamma = (V^\gamma, E^\gamma)$,*
- *la donnée des valeurs des variables présentes sur les nœuds du système $(var_i^\gamma / \forall i \in V^\gamma)$*
- *le multi-ensemble de tous les messages en transit $\mathcal{M}^\gamma$, constitué d'éléments $(msg, param, i, j)$ avec $msg \in \{Token, Recruit, Delete, ACK, Cluster, Transmit\}$, $i, j \in V^\gamma$.*

Plusieurs messages identiques peuvent être présents dans $\mathcal{M}$. On note P l'ensemble des paramètres pouvant être porté par un message et
$\delta : \{Token, Recruit, Delete, ACK, Cluster, Transmit\} \times P \times V^2 \to \mathbb{N}$ la multiplicité des messages.

Chaque message comporte dans ses paramètres un identifiant de cluster. On appelle identifiant du message cet identifiant de cluster. Ainsi, le message $Recruit[x]$ est d'identifiant x.

À partir d'une configuration, le système peut évoluer de deux façons différentes :

- soit par une modification du graphe de communication, ce que l'on appelle un changement topologique ;
- soit à la suite de la réception d'un message, ou de l'éveil d'un nœud selon les règles décrites par l'algorithme, ce que l'on appelle une *transition*.

Le système étant dynamique, une cause extérieure à l'algorithme peut modifier le graphe de communication à tout moment. Un changement topologique est donc le passage d'une configuration à une autre, par un changement du graphe de communication uniquement. Ainsi, lorsqu'un changement topologique transforme une configuration γ en une configuration γ', toutes les variables sont inchangées entre les deux configurations, c'est-à-dire que pour toute variable var, on a $\forall k \in V^{\gamma'} \cap V^\gamma, var_k^{\gamma'} = var_k^\gamma$.

On suppose que tous les nœuds qui rejoignent le système exécutent la procédure d'initialisation $Init$ (algorithme 19). Ainsi, les variables des nouveaux nœuds du système ont les valeurs sui-

vantes : $\forall i \in V^{\gamma'} \setminus V^{\gamma}, cluster_i^{\gamma'} = (\perp, 0), core_i^{\gamma'} = \text{faux}, father_i^{\gamma'} = \perp, nexto_i^{\gamma'} = 0, complete_i^{\gamma'} =$ faux.

On suppose également que lorsqu'un canal de communication apparaît dans le système, il ne contient aucun message.

Tout changement topologique se décompose en un nombre fini des changements élémentaires suivants :

- ajout d'un nœud,
- suppression d'un nœud,
- ajout d'un lien de communication,
- suppression d'un lien de communication.

En étudiant chacun de ces changements topologiques élémentaires, nous montrons à la section 4.3.3 qu'après tout changement topologique, le système finit par se retrouver dans un état de fonctionnement correct.

Lorsqu'un nœud s'éveille, ou lorsqu'il reçoit un message, il exécute la procédure associée à cet événement. Cette procédure peut modifier les variables présentes sur le nœud, et peut également modifier le multi-ensemble des messages en transit $\mathcal{M}$, via la primitive *envoyer le message à*. Lors d'une transition de l'algorithme, le graphe de communication du système ne change pas.

Un transition déclenchée par la réception d'un message $msg[x]$ sera notée $\gamma \vdash_{msg[x]} \gamma'$ dans la suite.

Définition 12. *Une exécution $\mathcal{E} = (\gamma_1, \ldots, \gamma_n, \ldots)$ est une suite de configurations du système, chacune étant liée à la suivante par un changement topologique ou par une transition : $\forall i \in \mathbb{N}, (\gamma_i \vdash \gamma_{i+1})$ ou $(\gamma_{i+1}$ se déduit de γ_i par un changement topologique).*

Nous détaillons dans la suite quelques transitions de cet algorithme.

Réception d'un message *Recruit*[x]

Lorsqu'un nœud i reçoit un message $Recruit[x]$ en provenance d'un nœud e, il exécute la procédure *JoinOrdinary* (algorithme 5). Le système passe d'une configuration γ à une configuration γ' par la transition $\gamma \vdash \gamma'$. Si le nœud i est déjà dans un cluster, c'est-à-dire si $cluster_i^{\gamma} \neq (\perp, 0)$, le message est ignoré. Dans ce cas, $\delta^{\gamma'}(Recruit, x, e, i) = \delta^{\gamma}(Recruit, x, e, i) - 1$. Aucune variable sur aucun nœud du système ne change de valeur : $\forall k \in V^{\gamma} = V^{\gamma'}, var_k^{\gamma'} = var_k^{\gamma}$.

Si le nœud i n'est pas clusterisé, c'est-à-dire si $cluster_i^{\gamma} = (\perp, 0)$, il rejoint le cluster C_x en tant que nœud ordinaire (algorithme 5). Les variables sur les autres nœuds que i ne changent pas : $\forall k \neq i, var_k^{\gamma'} = var_k^{\gamma}$.

Les variables du nœud i prennent quant à elle les valeurs suivantes : $cluster_i^{\gamma'} = x$, $core_i^{\gamma'} =$ faux, $nexto_i^{\gamma'} = nexto_i^{\gamma}$, $listmsg_i^{\gamma'} = \emptyset$ et $father_i^{\gamma'} = e$. Le nœud i envoie également un message $Cluster[x]$ à tous ses voisins : $\forall k \in V^{\gamma}$ avec $(k, i) \in E^{\gamma}, \delta^{\gamma'}(Cluster, x, i, k) = \delta^{\gamma'}(Cluster, x, i, k) +$ 1.. Le nœud i envoie également un message $ACK[x]$ à l'expéditeur e : $\delta^{\gamma'}(ACK, x, i, e) = \delta^{\gamma'}(ACK, x, i, e) + 1$.

Réception d'un message *Delete*[x]

Lorsqu'un nœud i reçoit un message $Delete[x]$ en provenance d'un nœud e, il exécute la procédure *À la réception d'un message Delete*, algorithme 7.

Si le nœud e n'est pas le père du nœud i (c'est-à-dire si $father_i^{\gamma} \neq e$), ou si le i n'est pas un nœud du cluster C_x (c'est-à-dire si $cluster_i^{\gamma} \neq x$), alors le message est ignoré. Toutes les

variables, sur tous les nœuds du système, restent identiques :

$$\forall var, \forall k \in V^{\gamma} = V^{\gamma'}, var_k^{\gamma'} = var_k^{\gamma}.$$

De plus, le message $Delete[x]$ reçu est enlevé de $\mathcal{M}^{\gamma'}$, c'est-à-dire $\delta^{(\gamma')}(Delete, x, e, i) = \delta^{(\gamma)}(Delete, x, e, i) - 1$.

Si le nœud e est le père du nœud i, et que ce dernier est dans le cluster C_x^{γ} (c'est-à-dire si $father_i^{\gamma} = e \wedge cluster_i^{\gamma} = x$), alors le nœud i exécute la procédure *Leave* (algorithme 6). Seules les variables du nœud i vont changer :

$$\forall k \neq i, \forall var, var_k^{\gamma'} = var_k^{\gamma}.$$

Le nœud i quitte le cluster C_x, et il envoie à chacun de ses voisins un message $Delete[x]$. Ainsi, $cluster_i^{\gamma'} = (\perp, 0)$, $core_i^{\gamma'} =$ faux, $nexto_i^{\gamma'} = nexto_i^{\gamma}$, $listmsg_i^{\gamma'} = \emptyset$ et $father_i^{\gamma'} = \perp$. De plus, le message $Delete[x]$ adressé à i est enlevé de $\mathcal{M}^{\gamma'}$, et des messages $Delete[x]$ émis par le nœud i pour chacun de ses voisins y sont ajoutés : $\delta^{(\gamma')}(Delete, x, e, i) = \delta^{(\gamma)}(Delete, x, e, i) - 1$ et $\forall k \in V^{\gamma'}, (i, k) \in E^{\gamma'} = E^{\gamma} \Rightarrow \delta^{(\gamma')}(Delete, x, i, k) = \delta^{(\gamma)}(Delete, x, i, k) + 1$.

Réception d'un message $Token[x]$

Lors de la réception d'un message $Token[x]$, le récepteur i exécute l'algorithme 10. Les cinq situations suivantes sont possibles :

- Le récepteur est un nœud de cœur dont les variables sont mises à jour.
- Le récepteur est un nœud ordinaire qui n'est pas recruté dans K_x.
- Le récepteur est recruté ou promu dans K_x.
- Le récepteur déclenche la destruction du cluster C_x.
- Le récepteur renvoie le message $Token_x$ à l'expéditeur.

Dans chaque cas, seules les variables du nœud i peuvent changer : $\forall k \neq i, var_k^{\gamma'} = var_k^{\gamma}$.

Dans les trois premiers cas, la procédure *TriggerUpperLevel* (algorithme 18) peut être appelée. S'il y a des messages en attente sur le nœud i, c'est-à-dire si $listmsg_i^{\gamma} \neq \emptyset$, alors *TriggerUpperLevel* émule la réception de ces messages par le nœud virtuel x. Nous reviendrons sur les modifications apportées à cette variable ainsi que sur les messages *Transmit* émis par *TrigerUpperLevel* dans la section 4.5.

Dans les deux derniers cas, aucune variable n'est modifiée : $var_i^{\gamma'} = var_i^{\gamma}$. Le jeton $Token_x$ en transit entre les nœuds e et i est enlevé de $\mathcal{M}^{\gamma}$, et remplacé dans $\mathcal{M}^{\gamma'}$ par le message $Token_x$ renvoyé à l'expéditeur, ou par un message $Delete[x]$ adressé à l'expéditeur : $\delta^{(\gamma')}(Token, x, e, i) = \delta^{(\gamma)}(Token, x, e, i) - 1 = 0$ puisque le seul message $Token_x$ en transit est reçu, et $\delta^{(\gamma')}(Token, x, i, e) = \delta^{(\gamma)}(Token, x, i, e) + 1 = 1$, ou alors $\delta^{(\gamma')}(Delete, x, i, e) = \delta^{(\gamma)}(Delete, x, i, e) + 1$.

Si le récepteur i est dans K_x, c'est-à-dire si $cluster_i^{\gamma} = x = Token_x.cluster$ et $core_i^{\gamma} =$ vrai, alors le nœud i reste un nœud du cœur de C_x. Le nœud exécute la procédure *UpdateStatus* (algorithme 14) pour mettre à jour les informations dans le jeton, dont la représentation de l'arbre couvrant qu'il porte. Puis le jeton est envoyé à un nœud j voisin, choisi uniformément au hasard. Ainsi $cluster_i^{\gamma'} = cluster_i^{\gamma} = x$, et $core_i^{\gamma'} = core_i^{\gamma} =$ vrai. La variable $father$ est mise à jour : $father_i^{\gamma'} = k$.

Le message $(Token, x, e, i)$ est enlevé de $\mathcal{M}^{\gamma}$ et $(Token, x, i, j)$ est ajouté à $\mathcal{M}^{\gamma'}$, ainsi qu'un message $Recruit[x]$ adressé à tous les voisins de i :

$$\forall k \in V, (k, i) \in E \Rightarrow \delta^{(\gamma')}(Recruit, x, i, k) = \delta^{(\gamma)}(Recruit, x, i, k) + 1.$$

Si le récepteur i est un nœud ordinaire qui n'est pas recruté, c'est-à-dire si $cluster_i^\gamma = x = Token_x.cluster$ et $Token_x.size \geq K$, alors le nœud i reste un nœud ordinaire de C_x. L'arbre couvrant est mis à jour avant que le jeton soit renvoyé à l'expéditeur. Ainsi, $cluster_i^{\gamma'} = cluster_i^\gamma = x$ et $core_i^{\gamma'} = core_i^\gamma =$ faux. La variable $father$ est mise à jour : $father_i^{\gamma'} = em$.

Le message $(Token, x, e, i)$ est enlevé de $\mathcal{M}^\gamma$ et $(Token, x, i, e)$ est ajouté à $\mathcal{M}^{\gamma'}$.

Si le récepteur est un nœud ordinaire ou endormi ($core_i =$ faux), et si le cluster C_x n'est pas complet ($Token_x.size < K$), alors le récepteur devient un nœud du cœur de C_x. Ainsi, $cluster_i^{\gamma'} = x$ et $core_i^{\gamma'} =$ vrai. Le jeton est ensuite envoyé à un voisin j choisi uniformément au hasard. Le message $(Token, x, e, i)$ est enlevé de $\mathcal{M}^\gamma$ et $(Token, x, i, j)$ est ajouté à $\mathcal{M}^{\gamma'}$, ainsi qu'un message $Recruit[x]$ adressé à tous les voisins de i :

$$\forall k \in V, (k, i) \in E \Rightarrow \delta^{(\gamma')}(Recruit, x, i, k) = \delta^{(\gamma)}(Recruit, x, i, k) + 1.$$

4.3 Démonstration du clustering à un étage

Cette démonstration est rédigée en trois étapes. Dans un premier temps, on donne une définition formelle des configurations qui peuvent être atteintes par le système en l'absence de changement topologique. Ces configurations sont appelées configurations *valides*. Ensuite on démontre que le système se maintient dans cet ensemble de configurations au cours de toute exécution sans changement topologique.

Dans un deuxième temps, nous démontrons qu'un ensemble de configurations vérifiant la spécification présentée à la section 2.1.3 agit comme un attracteur dans l'ensemble des configurations valides. Cela signifie qu'en l'absence de changement topologique, l'algorithme finit par se stabiliser dans un ensemble de configurations qui vérifient la spécification.

Dans un troisième temps, nous montrons qu'en cas de changement topologique, bien que le système puisse se retrouver dans une configuration invalide, il revient en un temps fini dans une configuration valide. À partir de là, le système peut reprendre sa convergence vers une configuration vérifiant la spécification.

4.3.1 Clôture des configurations valides

L'algorithme calcule un clustering du système qui vérifie la spécification présentée à la section 2.1.3. En particulier, chaque cluster calculé est un cluster correct. Son cœur est un ensemble dominant connexe du cluster, de taille comprise entre 1 et K. De plus, il existe un unique message jeton en circulation dans le réseau avec son identifiant, qui transporte un arbre couvrant du cluster (définition 2).

Une configuration vérifie la spécification du problème (définition 4) si et seulement si tous les clusters qu'elle contient sont corrects, avec un cœur de taille au moins deux, et sans aucun couple de clusters voisins incomplets. Nous démontrerons dans la section 4.3.2 que ces propriétés finissent par être vraies presque sûrement dans toute exécution.

Notre algorithme vise à construire autant que possible des clusters dont le cœur a pour taille un paramètre K de l'algorithme. Pour y parvenir, l'algorithme peut être amené à détruire des clusters incomplets. Cette destruction n'étant pas instantanée, deux types de clusters vont être présents dans le système :

- les clusters cohérents,
- les clusters en destruction.

Les clusters cohérents sont des clusters correctement construits. Un cluster cohérent va grandir jusqu'à devenir complet, ou va finir par devenir un cluster en destruction. Un cluster en destruction est un cluster amené à disparaître.

Commençons par donner une définition de ce qu'est un cluster cohérent :

Définition 13. *Un cluster C_x est dit cohérent si :*

- *Structure du cluster :*
  1. $1 \leq |K_x| \leq K$.
  2. *K_x est un ensemble dominant connexe de C_x (K_x est connexe, et tous les nœuds de C_x ont un voisin dans K_x ; en particulier, cela implique la connexité de C_x).*
- *Messages en circulation :*
  1. *Tous les nœuds ayant émis un message $Recruit[x]$ sont des nœuds de cœur :* $(Recruit, x, e, r) \in \mathcal{M} \Rightarrow s \in K_x$.
  2. *Il n'y a pas de message $Delete[x]$ en circulation dans le réseau :* $(Delete, y, e, r) \in \mathcal{M} \Rightarrow y \neq x$.
  3. *Tout message $ACK[x]$ en circulation dans le réseau est émis à destination d'un nœud de cœur :* $(ACK, x, e, r) \in \mathcal{M} \Rightarrow r \in K_x$.
  4. *Il existe un unique message $Token$ en circulation tel que $Token.id = x$ (noté $Token_x$ dans la suite).*
  5. *Le jeton $Token_x$ est dans un lien de communication adjacent au cœur du cluster :* $(Token, [x, topology], e, r) \in \mathcal{M} \Rightarrow ((e \in K_x) \vee (r \in K_x))$.
- *Arbre couvrant :*
  1. $\mathcal{A}_x = (C_x, \mathcal{R}_x)$ *où*

$$\mathcal{R}_x = \left\{(l,m) \in C_x^2 / father_m = l \text{ et } (l,m) \in E\right\} \setminus \left\{ (j,k) \in E / \begin{array}{l} (Token_x, param, j, k) \in \mathcal{M} \vee \\ (Token_x, param, k, j) \in \mathcal{M} \end{array} \right\}$$

     est un arbre couvrant de C_x (l'arête portant le jeton est exclue).
  2. *$Token_x.tree$ est une représentation de l'arbre couvrant $\mathcal{A}_x$, cohérente avec la représentation distribuée implémentée par la variable $father$:*

$$\forall j \neq k \in C_x,\ Token_x.tree[j] = k \ \Rightarrow father_j = k.$$

  3. *Le nœud racine a les propriétés suivantes :*

$$\begin{aligned}(Token_x.tree[j] = j) \Leftrightarrow & (j \text{ est la racine de } \mathcal{A}_x) \\ \Leftrightarrow & (\exists k \in N_j, (Token_x, param, j, k) \in \mathcal{M} \\ & \vee (Token_x, param, k, j) \in \mathcal{M}).\end{aligned}$$

  4. $(((Token_x, e, r) \in \mathcal{M}) \wedge (e \in K_x)) \Rightarrow ((e \text{ racine de } \mathcal{A}_x) \wedge (father_e = r))$.
  5. $k \in C_x \setminus K_x \Rightarrow \forall l \in C_x, father_l \neq k$.

Pour vérifier la spécification, le système ne doit comporter que des clusters corrects. Cela signifie que les clusters comportent une partie dominante connexe dont la taille est comprise entre 1 et K et possèdent un jeton qui transporte un arbre couvrant du cluster (définition 2). On

retrouve donc ces propriétés dans la définition des clusters cohérents. De plus, pour garantir le bon fonctionnement de l'algorithme, il ne doit pas y avoir de message $Delete[x]$ dans le système, les messages $Recruit[x]$ doivent être émis par des nœuds appartenant au cœur K_x, et les messages $ACK[x]$ doivent être envoyés à des nœuds de K_x.

Notre algorithme utilise deux représentations de l'arbre couvrant d'un cluster. Une représentation complète de l'arbre couvrant est portée par le jeton, $Token.tree$. De plus, une représentation distribuée est mise en place, chaque nœud connaissant l'identifiant de son père, enregistré dans la variable $father$. Ces représentations doivent être cohérentes entre elles. Cependant, il y un décalage dans le temps entre ces deux représentations. En effet, lorsqu'un nœud i reçoit le jeton de son cluster, il devient la racine de l'arbre couvrant. Il met à jour la représentation $Token.tree$, puis envoie le jeton à un de ces voisins j, qui devient son père ($father_i \leftarrow j$). Ainsi le nœud i considère le nœud j comme son père, alors que cette information ne pourra être écrite dans $Token.tree$ que lorsque le nœud j recevra le jeton. De plus, le nœud j peut appartenir à un autre cluster, et il n'entrera alors ni dans le cluster C_x, ni dans son arbre couvrant. Par conséquent, le lien de communication contenant le jeton ne doit pas être considéré comme une arête de l'arbre couvrant du cluster.

Enfin, la racine de l'arbre couvrant d'un cluster est connue : c'est le dernier nœud du cluster ayant eu le jeton, et les nœuds ordinaires ne pouvant recruter, ce sont forcément des feuilles de l'arbre couvrant.

En particulier, un cluster cohérent est un cluster correct.

Pendant la destruction d'un cluster, celui-ci n'est plus cohérent (en particulier, il n'est plus associé à un message jeton). Nous donnons dans la définition 14 les propriétés que doit vérifier un cluster pour que sa destruction se passe correctement.

Définition 14. *Un cluster C_x est dit en destruction si et seulement si :*

- *il n'y a pas de message Token en circulation dans le réseau et tel que $Token.id = x$: $(Token_y, param, e, r) \in \mathcal{M} \Rightarrow y \neq x$;*
- *tous les messages $Recruit[x]$ en transit ont été émis par des nœuds qui ne sont pas des nœuds ordinaires de C_x : $(Recruit, x, e, r) \in \mathcal{M} \Rightarrow (e \in K_x$ ou $e \notin C_x)$;*
- *tous les messages $ACK[x]$ en transit ont été émis à destination d'un nœud qui n'est pas un nœud ordinaire de C_x : $(ACK, x, e, r) \in \mathcal{M} \Rightarrow (r \in K_x$ ou $r \notin C_x)$;*
- *$\mathcal{F}_x = (C_x, \{(l, m) \in E / father_m = l\})$ est une forêt couvrante de C_x ;*
- *il y a un message $Delete[x]$ en transit vers la racine de l'arbre de chaque composante connexe de C_x qui contient au moins un nœud de cœur : $\forall k \in K_x$, $(father_k \notin C_x$ ou $(k, father_k) \notin E) \Leftrightarrow \exists (Delete, x, father_k, k) \in \mathcal{M}$ et k est une racine d'un arbre de la forêt couvrante $\mathcal{F}_x$;*
- *$k \in C_x \setminus K_x \Rightarrow \forall l \in C_x, father_l \neq k$;*
- *tout nœud ordinaire de C_x, si son père n'est pas dans K_x, est émetteur d'un message $ACK[x]$ vers son père, ou destinataire d'un message $Delete[x]$ envoyé par son père : $\forall k \in C_x \setminus K_x, (father_k \notin C_x) \Rightarrow ((Delete, x, father_k, k) \in \mathcal{M}$ ou $(ACK, x, k, father_k) \in \mathcal{M})$.*

La destruction d'un cluster est effectuée par la propagation d'une vague de messages $Delete[x]$ le long de l'arbre couvrant du cluster. Cet arbre devient ainsi une forêt couvrante, un message $Delete[x]$ étant adressé à la racine de chaque arbre de cette forêt.

Chaque nœud racine finit par recevoir ce message $Delete[x]$, il quitte alors le cluster et propage le message $Delete[x]$ à tous ses voisins, et en particulier à ses fils dans la forêt couvrante. De cette façon, tous les nœuds du cluster finiront par recevoir un message $Delete[x]$ et par quitter le cluster. Cependant, des messages $Recruit[x]$ peuvent encore être en circulation dans le système. Les messages ACK sont la pour répondre au cas d'un message $Recruit[x]$ reçu alors que son

expéditeur a déjà quitté le cluster C_x. En effet, chaque message $Recruit[x]$ est émis par un nœud du cœur K_x. Cependant celui-ci peut avoir reçu un message $Delete[x]$ et avoir quitté le cluster C_x depuis.

Puisque les canaux de communication ne vérifient pas l'hypothèse FIFO, le nœud destinataire peut recevoir le message $Delete[x]$ en premier, qui sera ignoré, puis recevoir le message $Recruit[x]$. Le destinataire est alors recruté en tant que nœud ordinaire et envoie un message $ACK[x]$ à son père. Celui-ci n'étant plus dans le cluster C_x, il répondra par un message $Delete[x]$. Ainsi, tout nœud ordinaire d'un cluster en destruction a un père dans K_x, ou un message $ACK[x]$ ou $Delete[x]$ est en transit entre lui et son père.

En l'absence de changement topologique, le système se trouve et se maintient dans une configuration ne contenant que des clusters cohérents ou en destruction. Une telle configuration est appelée une configuration *valide*.

Définition 15. *Une configuration est dite valide si elle ne contient que des clusters cohérents ou en destruction.*

À partir d'une configuration valide, le système finira par se retrouver dans une configuration vérifiant la spécification du problème. Cela est démontré dans la section 4.3.2.

Nous démontrons maintenant que, en l'absence de changement topologique, le système se maintient dans une configuration valide.

Théorème 16. *Si γ est une configuration valide et si l'exécution de l'algorithme amène à une configuration γ', alors γ' est une configuration valide.*

La démonstration de ce théorème consiste à vérifier que pour chaque transition possible de l'algorithme, le système reste dans une configuration valide.

Les transitions possibles sont les suivantes :

- création d'un cluster par la procédure *wakeup*,
- réception d'un message et exécution de l'algorithme associé.

Dans toute la section 4.3, on ignore les messages *Cluster* et *Transmit* qui n'ont pas d'impact sur le clustering.

Lemme 17. *Soit C_x un cluster. La réception d'un message Cluster ou d'un message Transmit, sur n'importe quel nœud du réseau, et quel que soit l'identifiant associé, laisse C_x inchangé.*

Démonstration. La réception d'un message *Cluster* modifie uniquement la variable *gate* du nœud récepteur (algorithme 13). De la même manière, un message *Transmit* déclenche l'exécution de l'algorithme 17. Le message est soit ignoré (ligne 1), soit relayé au suivant sur le chemin qu'il contient (ligne 4), soit le message de niveau supérieur transporté est enregistré dans la variable *listmsg* du nœud récepteur (ligne 2). Dans tous les cas, aucune variable, ni aucun message de la définition d'un cluster cohérent n'est modifié. Les clusters du système sont donc inchangés. □

Pour démontrer le théorème 16, nous montrons donc que les clusters nouvellement créés sont valides (proposition 19). Nous montrons ensuite que lors d'une transition déclenchée par la réception d'un message, un cluster cohérent va soit rester cohérent, soit passer en destruction, et qu'un cluster en destruction va rester en destruction ou devenir vide.

Pour cela, nous considérons un cluster et nous commençons par traiter les transitions liées à une réception de message entre deux nœuds n'appartenant pas à ce cluster (proposition 18). Nous observons ensuite, pour les clusters cohérents, puis pour les clusters en destruction, l'impact des transitions déclenchées par :

– un message sortant du cluster (proposition 20 et 24),
– un message interne au cluster (proposition 21 et 25),
– un message entrant dans le cluster (proposition 22 et 26).

Dans la suite, on notera C_x^γ le cluster C_x tel qu'il apparaît dans la configuration γ, et K_x^γ son cœur.

La majorité des communications se situant en dehors d'un cluster n'affecte pas directement celui-ci. Cependant, puisque les communications sont asynchrones, un message $Recruit[x]$ ou un message $ACK[x]$ obsolète peut se retrouver en transit entre deux nœuds n'appartenant plus à C_x. Nous vérifions que la réception de ces messages ne peut pas rendre la configuration invalide.

Proposition 18. *Soit C_x^γ un cluster cohérent ou en destruction, et soient $e, i \notin C_x^\gamma$ deux nœuds n'appartenant pas à C_x^γ. Si $\gamma \vdash \gamma'$ est une transition déclenchée par la réception par le nœud i d'un message émis par le nœud e, alors $C_x^{\gamma'}$ reste cohérent ou en destruction.*

Démonstration. Soit C_x^γ un cluster cohérent ou en destruction, et soient deux nœuds $e, i \notin C_x$. Un message en transit entre e et i peut soit porter l'identifiant x, soit porter un identifiant $y \neq x$.

Si le message porte un identifiant différent de x, alors $C_x^{\gamma'} = C_x^\gamma$. En effet, la réception d'un message d'identifiant $y \neq x$ ne peut pas modifier le cluster C_x^γ :

– $ACK[y]$: soit il est ignoré, soit il déclenche l'envoi d'un message $Delete[y]$ (algorithme 9) ;
– $Delete[y]$: soit il est ignoré, soit il déclenche le départ du récepteur et l'émission de message $Delete[y]$ (algorithme 7) ;
– $Recruit[y]$: soit il est ignoré, soit le récepteur $i \notin C_x$ rejoint le cluster C_y ;
– $Token_y$: soit le récepteur $i \notin C_x$ rejoint K_y, soit le message est renvoyé à l'expéditeur ou transféré à un voisin, soit i envoie un message $Delete[y]$ à e.

Dans tous les cas, seule la structure du cluster C_y^γ peut être modifiée. Le cluster C_x reste donc cohérent ou en destruction.

Étudions maintenant le cas d'un message d'identifiant x. Si le cluster C_x^γ est cohérent, alors par définition il ne peut pas y avoir de message d'identifiant x en transit entre e et i. Si le cluster C_x^γ est en destruction, alors par définition le message en transit ne peut pas être un message $Token_x$.

Si le message est un $Delete[x]$, puisque i n'est pas dans le cluster C_x, ce message est ignoré (algorithme 7). C_x est alors inchangé.

Si le message est un $ACK[x]$, le nœud i, n'appartenant pas à C_x, envoie un message $Delete[x]$ au nœud e (algorithme 9). Le cluster $C_x^{\gamma'} = C_x^\gamma$ est inchangé et reste un cluster en destruction.

Si le message est $Recruit[x]$, le nœud i peut éventuellement être recruté dans le cluster C_x comme nœud ordinaire (algorithme 5). Le nœud i devient un nœud ordinaire de C_x et envoie un message $ACK[x]$ au nœud e qui est devenu son père. Ainsi, on a $e = father_i \notin K_x$ et $\exists (ACK, x, i, father_i) \in \mathcal{M}^{\gamma'}$. Le cluster $C_x^{\gamma'}$ reste en destruction.

Ainsi, dans ces trois cas, le cluster $C_x^{\gamma'}$ reste un cluster en destruction.

Par conséquent, si C_x^γ est un cluster cohérent ou en destruction, alors pour toute transition $\gamma \vdash \gamma'$ déclenchée par la réception d'un message entre deux nœuds n'appartenant pas à C_x, $C_x^{\gamma'}$ reste un cluster cohérent ou un cluster en destruction. □

Les communications entre deux nœuds n'appartenant pas au cluster C_x laissent donc celui-ci cohérent, ou en destruction. Montrons maintenant que les clusters nouvellement créés sont cohérents :

Proposition 19. *Lorsqu'un nœud créé un cluster C_x, C_x est cohérent.*

Démonstration. Lorsque le nœud i s'éveille à la fin de sa période de sommeil, il exécute la procédure *wakeup*() (algorithme 2.3) pour créer un cluster. Ce cluster a pour identifiant $x = (i, nexto_i)$. Cet identifiant de cluster ne peut être déjà utilisé, car seul le nœud i peut créer un cluster ayant un tel identifiant, et car la variable $nexto_i$ est incrémentée après chaque création de cluster (procédure *wakeup*(), algorithme 2.3). Ainsi C_x ne contient qu'un nœud de cœur, émetteur de $Token_x$ et de tous les messages $Recruit[x]$ existant. Il n'y a aucun message $ACK[x]$ ni $Delete[x]$ en circulation dans le réseau. De plus, $\mathcal{A}_x$ est un arbre couvrant constitué du seul nœud i, vérifiant les propriétés de la définition de cluster cohérent (définition 13).

Le cluster C_x est donc cohérent. □

Nous montrons maintenant que lorsque $\gamma \vdash \gamma'$ par la réception d'un message, un cluster cohérent reste cohérent ou passe en destruction (proposition 23).

Considérons un cluster C_x^γ. Les propositions suivantes nous permettent d'affirmer qu'un cluster cohérent C_x reste cohérent ou passe en destruction lors de la réception d'un message d'identifiant x.

Proposition 20. *Si C_x^γ est un cluster cohérent et si $\gamma \vdash \gamma'$ est déclenchée par la réception, sur un nœud n'appartenant pas à C_x^γ, d'un message venant de C_x^γ, alors $C_x^{\gamma'}$ reste cohérent ou passe en destruction.*

Proposition 21. *Si C_x^γ est un cluster cohérent et si $\gamma \vdash \gamma'$ est déclenchée par la réception, sur un nœud de C_x^γ, d'un message émis par un nœud également dans C_x^γ, alors $C_x^{\gamma'}$ reste cohérent.*

Proposition 22. *Si C_x^γ est un cluster cohérent et si $\gamma \vdash \gamma'$ est déclenchée par la réception, sur un nœud du cluster C_x^γ, d'un message venant d'un nœud n'appartenant pas au cluster C_x^γ, alors $C_x^{\gamma'}$ reste cohérent.*

Pour démontrer ces propositions, on liste l'ensemble des transitions possibles lors de la réception d'un message. Pour chaque cas possible, on montre que le cluster C_x reste un cluster cohérent, ou qu'il devient un cluster en destruction. Ces démonstrations sont présentées en annexe de ce chapitre (lemme 72 à 91).

Ces trois propositions nous permettent de conclure qu'un cluster cohérent restera un cluster cohérent ou passera en destruction lors de n'importe quelle transition :

Proposition 23. *Si C_x^γ est un cluster cohérent et si $\gamma \vdash \gamma'$ est déclenchée par la réception d'un message, alors $C_x^{\gamma'}$ reste cohérent ou devient en destruction.*

D'une manière similaire, nous démontrons maintenant qu'un cluster en destruction reste en destruction, ou devient vide, quelle que soit la transition suivie par le système.

Proposition 24. *Si C_x^γ est en destruction et si $\gamma \vdash \gamma'$ est déclenchée par la réception, sur un nœud externe à C_x^γ, d'un message émis par un nœud de C_x^γ, alors $C_x^{\gamma'}$ reste en destruction.*

Proposition 25. *Si C_x^γ est en destruction et si $\gamma \vdash \gamma'$ est déclenchée par la réception, sur un nœud du cluster C_x^γ, d'un message émis par un nœud de C_x^γ, alors $C_x^{\gamma'}$ reste en destruction.*

Proposition 26. *Si C_x^γ est en destruction et si $\gamma \vdash \gamma'$ est déclenchée par la réception, sur un nœud de C_x^γ, d'un message émis par un nœud externe à C_x^γ, alors $C_x^{\gamma'}$ reste en destruction.*

Ces trois propositions nous permettent de conclure que pour toute transition de l'algorithme, un cluster en destruction reste en destruction. Le lecteur trouvera la démonstration de chacune de ces trois propositions en annexe.

Proposition 27. *Si C_x^γ est en destruction et si $\gamma \vdash \gamma'$, alors $C_x^{\gamma'}$ reste en destruction.*

Cela nous permet de démontrer la clôture des configurations valides sous les transitions de l'algorithme :

Preuve du théorème 16. En l'absence de changement topologique, si γ est une configuration valide, alors tous les clusters sont cohérents ou en destruction dans γ. Grâce aux propositions 19, 23 et 27, si $\gamma \vdash \gamma'$, alors tous les clusters sont encore cohérents ou en destruction dans γ'.

Ainsi γ' est une configuration valide. □

Chaque exécution commence dans une configuration ne comprenant que des nœuds endormis, qui est une configuration valide. Ainsi, le théorème 16 nous permet d'affirmer qu'en l'absence de changement topologique, le système se maintient dans l'ensemble des configurations valides.

4.3.2 Convergence vers la spécification

En l'absence de changement topologique, le clustering finit par ne plus changer. Cependant, notre algorithme ne devient jamais silencieux. En particulier, les messages jetons continuent de circuler dans le réseau, et les arbres couvrants qu'ils transportent continuent d'évoluer.

Nous montrons ici que toute exécution sans changement topologique possède un suffixe infini de configurations vérifiant la spécification, et dans lequel les clusters n'évoluent plus. Lorsque c'est le cas, ont dit que le système a *convergé*.

Pour cela, nous montrons dans un premier temps que le clustering converge presque sûrement. Ensuite, en supposant que le clustering a convergé, nous démontrons par l'absurde que le système est dans une configuration qui vérifie la spécification.

Proposition 28. *En l'absence de changement topologique, l'algorithme converge presque sûrement.*

Démonstration. Considérons une exécution infinie $\mathcal{E} = (\gamma_1, \dots, \gamma_k, \dots)$ sans changement topologique. Le graphe de communication, identique dans chaque configuration (puisque l'exécution ne comporte pas de changement topologique), est noté $G = (V, E)$ dans la suite.

Lorsqu'un nœud s'éveille, il crée un nouveau cluster C_x, et envoie le jeton associé à l'un de ses voisins. Si ce voisin est endormi, ou s'il est un nœud ordinaire, il est recruté dans le cœur du cluster C_x, qui comporte alors deux nœuds. Puisque les nœuds s'éveillent à des temps aléatoires, on finit nécessairement par obtenir un cluster dont le cœur est de taille supérieure ou égale à deux. Ainsi, $\{i \in V / \exists p \in \mathbb{N}, \exists \gamma \in \mathcal{E}, |K_{(i,p)}^\gamma| \geq 2\} \neq \emptyset$ est non vide.

Soit k le plus grand identifiant de cet ensemble. Le nœud k initie donc la construction d'un cluster dont le cœur atteint une taille supérieure à deux. Notons $x = (k, p)$ l'identifiant de ce cluster.

En l'absence de changement topologique, le nœud k de peut pas quitter le cluster C_x tant que ce cluster est cohérent. En effet, un nœud ne quitte un cluster que lors de la réception d'un message $Delete[x]$ (algorithme 7), qui par définition n'existe pas tant que le cluster C_x est cohérent.

De plus, le cluster C_x ne peut pas passer en destruction. En effet, un cluster ne peut être détruit que lorsque son jeton est reçu par un nœud de cœur d'un cluster ayant un identifiant $y > x$, et ayant au moins deux nœuds (ligne 30 à 33 de l'algorithme 10). Or, si $y = (l, p) > x$ est l'identifiant d'un cluster ayant un cœur de taille au moins deux, cela signifie que $l > k$ ou que $l = k$ et $p > n$, ce qui est impossible par construction de k. De plus, puisque k reste dans C_x, aucun cluster d'identifiant (k, p) avec $p > n$ ne peut être créé.

Le cluster C_x ne peut donc pas être détruit, et finira par atteindre sa taille maximale (éventuellement inférieure à K).

Lorsqu'il a atteint sa taille maximale, soit C_x est complet ($|K_x| = K$), soit il finit par être entouré de clusters qui le sont. En effet, si ce n'était pas le cas, un jeton de l'un de ses voisins C_y finirait par visiter un nœud de K_x d'après la propriété de percussion des marches aléatoires. Dans ce cas, le cluster C_y deviendrait un cluster en destruction (lignes 30 à 33, algorithme 10). Presque sûrement, le jeton $Token_x$ finirait par recruter un des nœuds anciennement dans C_y, ce qui contredit le fait que C_x a atteint sa taille maximale. Le cluster C_x finit donc bien par être complet, ou entouré par des clusters complets.

Ainsi, il finit par exister un cluster C_y complet. La suite de l'exécution peut être vu comme une exécution sur le système privé des nœuds appartenant au cœur de ce cluster K_y. En effet, puisque C_y est complet, le message $Token_y$ sera systématiquement renvoyé à l'expéditeur. Les nœuds ordinaires de C_y peuvent quant à eux être vus comme des nœuds non clusterisés ne créant pas de nouveau cluster.

Ce nouveau système n'est plus forcément connexe. Cependant, les mêmes arguments permettent d'affirmer que chaque composante connexe de taille supérieure à K finit par comporter un cluster complet.

Ainsi, en poursuivant ce raisonnement, on obtient un système composé uniquement de composantes connexes de tailles inférieures à K. Dans ces composantes connexes, un cluster unique constitué de tous les nœuds finira par être formé.

Le système a alors convergé et ne peut plus évoluer. □

Le système converge donc presque sûrement. Montrons maintenant que lorsque le système a convergé, il se trouve dans une configuration valide. Pour cela, nous commençons par montrer que les clusters en destruction finissent par disparaître.

Un cluster est détruit par la propagation d'une vague de messages $Delete[x]$ sur l'arbre couvrant. Durant cette propagation, le cœur du cluster ne peut plus grandir, puisqu'il n'y a plus de message $Token_x$. Tous les nœuds de K_x vont donc recevoir un message $Delete[x]$ et quitter le cluster.

De même, les nœuds ordinaires vont finir par quitter le cluster, soit un recevant la vague de messages $Delete[x]$, soit via le mécanisme des messages ACK présenté précédemment.

Nous montrons dans un premier temps que tous les nœuds du cœur d'un cluster en destruction finissent par quitter ce cluster.

Lemme 29. *Si C_x est un cluster en destruction, alors dans toute exécution, il existe une configuration γ' telle que $\gamma \vdash^* \gamma'$ et $K_x^{\gamma'} = \emptyset$.*

Démonstration. Premièrement, lorsqu'un cluster est en destruction, aucun nœud ne peut rejoindre son cœur. En effet, la fonction $JoinCore$ n'est exécutée qu'au passage du jeton $Token_x$, qui n'existe pas puisque C_x est en destruction.

Considérons une configuration γ et un cluster en destruction C_x^γ. Par définition des clusters en destruction, $\mathcal{F}_x = (C_x, \{(l,m) \in E / father_m = l\})$ est une forêt couvrante de C_x, et donc acyclique. Ainsi, l'ensemble $\{k \in K_x / father_k \notin C_x\} \neq \emptyset$ est non-vide.

Par définition des clusters en destruction, chaque nœud dans cet ensemble est le destinataire d'un message $Delete[x]$ envoyé par son père, c'est-à-dire que l'ensemble $\{k \in K_x / (Delete, x, father_k, k) \in \mathcal{M}\} \neq \emptyset$ est non vide.

Puisque, en l'absence de changement topologique, tout message finit par être reçu, un nœud k de cet ensemble finit par recevoir un message $Delete[x]$ provenant de son père. Ce message déclenche l'exécution de l'algorithme 7, et le nœud k quitte le cluster C_x. Cette transition est

détaillée à la section 4.2. Le cœur du cluster K_x finit donc par diminuer, et le cluster C_x reste en destruction d'après la propriété 27.

Par une récurrence sur la taille du cœur des clusters en destruction, on démontre que les clusters en destruction finissent par avoir un cœur vide. □

Grâce à ce lemme, nous montrons que tout cluster en destruction finit par être vide, et qu'aucun nouveau nœud ne peut le rejoindre.

Proposition 30. *Si C_x^γ est en destruction, alors dans toute exécution, il existe une configuration γ' telle que $\gamma \vdash^* \gamma'$ et $C_x^{\gamma'} = \emptyset$.*

Démonstration. Soit une configuration γ, et un cluster en destruction C_x^γ. D'après le lemme 29, le cœur de C_x finit par être vide, c'est-à-dire il existe une configuration γ' telle que $K_x^{\gamma'} = \emptyset$. De plus, d'après la propriété 27, $C_x^{\gamma'}$ est toujours un cluster en destruction.

Puisque C_x est un cluster en destruction, il n'y a aucune message $Token_x$ en transit dans le réseau, et aucun nouveau message $Recruit[x]$ ne peut être émis. D'après notre modèle, tous ces messages finiront par être reçus. Ainsi, le système se retrouve dans une configuration γ'' ne contenant plus aucun message $Recruit[x]$, et dans laquelle C_x est toujours un cluster en destruction par la propriété 27.

À ce stade, le nombre de nœuds de C_x ne peut plus augmenter, puisque aucun message $Token_x$ ou $Recruit[x]$ n'est présent dans $\mathcal{M}^{\gamma''}$, ni ne peut être émis.

Par définition des clusters en destruction, aucun nœud ordinaire ne peut être le père d'un autre nœud du cluster. Ainsi, $\forall k \in C_x^{\gamma''}, father_k \notin C_x^{\gamma''}$.

De plus, par définition des clusters en destruction, chacun de ces nœuds est le destinataire d'un message $Delete[x]$ venant de son père, ou est l'émetteur d'un message $ACK[x]$ à destination de son père : $\forall k \in C_x^{\gamma''}, (Delete, x, father_k^{\gamma''}, k) \in \mathcal{M}^{\gamma''}$ ou $(ACK, x, k, father_k^{\gamma''}) \in \mathcal{M}^{\gamma''}$.

Tout ces messages $ACK[x]$ finiront par être reçus. À la réception d'un de ces messages, puisque le destinataire est dans un autre cluster, un message $Delete[x]$ est envoyé à l'émetteur (algorithme 9).

Ainsi, tous les nœuds ordinaires finiront par recevoir un message $Delete[x]$ de leur père, et quitteront le cluster C_x.

Ainsi, si C_x est en destruction, alors tôt ou tard $C_x = \emptyset$. □

Corollaire 31. *Lorsque le clustering a convergé, tous les clusters non vides sont cohérents.*

Ainsi, lorsque le système a convergé, il ne comporte que des clusters cohérents, qui sont en outre des clusters corrects. Nous vérifions maintenant qu'après convergence, le système est dans une configuration vérifiant la spécification :

Théorème 32. *Après convergence, le système est dans une configuration qui respecte la spécification, c'est-à-dire :*

- *Il n'y a aucun cluster en destruction ;*
- *Pour tout cluster C_x, $2 \leq |K_x| \leq K$;*
- *Si deux clusters sont voisins, au moins l'un d'entre eux est complet ;*
- *Chaque nœud est dans un cluster.*

Démonstration. Supposons que le système a convergé.

1. Il n'y a pas de cluster en destruction : Si un cluster C_x est en destruction, par le corollaire 31, tous les nœuds de C_x vont tôt ou tard le quitter, ce qui contredit le fait que le système ait convergé.

2. Pour tout cluster C_x, $2 \leq |K_x| \leq K$

   Nous savons, par définition des clusters cohérents, que tout cluster cohérent C_x possède un coeur K_x tel que $1 \leq |K_x| \leq K$. Nous supposons que $|K_x| = 1$, et nommons j son nœud de cœur. C_x est cohérent, donc il existe un message $(Token, x, j, l) \in \mathcal{M}$. Puisque chaque message émis finit par être reçu, l finit par recevoir le message $Token$. À ce moment :
   - Si l est un nœud ordinaire ou non clusterisé, il rejoint le cluster C_x (Algorithme 10 qui appelle la fonction $JoinCore$), ce qui contredit le fait que le système a convergé ;
   - Si l est un nœud de cœur d'un autre cluster, il envoie un message $Delete[x]$ au nœud j (fonction $Token$, ligne 34 à 36). C_x devient un cluster en destruction, ce qui contredit le fait que le système ait convergé.

   Ainsi, $|K_x| \neq 1$.

3. Soit deux clusters voisins C_x et C_y. Supposons qu'ils sont tous deux incomplets, et sans perte de généralité que $x < y$. Par la propriété de percussion des marches aléatoires, le message $Token_x$ finis par visiter un nœud l dans C_y (section 2.1.2) :
   - Si l est un nœud ordinaire, l entre dans K_x (fonction $Token()$), ce qui contredit le fait que le système ait convergé ;
   - Si i est un nœud de cœur, l envoi un message $Delete[x]$ au nœud j (ligne 34 36 de la fonction $Token()$). C_x devient un cluster en destruction, ce qui contredit le fait que le système ait convergé.

   Ainsi, au moins l'un des clusters est complet après convergence.

4. Supposons que le nœud j n'est dans aucun cluster. Il finira tôt ou tard par rejoindre un cluster :
   - Si j reçoit un message $Recruit[x]$, il rejoint C_x ;
   - Si j reçoit un message $Token_x$ venant d'un cluster incomplet C_x, il rejoint K_x ;
   - Sinon, j crée un cluster $C_{j,nexto_j}$ à son éveil (fonction $wakeup()$).

   Tous ces cas contredisent le fait que le système ait convergé. Ainsi, chaque nœud fait partie d'un cluster après convergence.

Après convergence, le système vérifie donc la spécification. □

Ainsi, en l'absence de changement topologique, le système converge en un temps fini vers une configuration vérifiant la spécification.

4.3.3 Réaction aux changements topologiques

Lorsqu'un changement topologique survient dans le réseau, le système peut se retrouver dans une configuration invalide. En particulier, lors de la disparition d'un lien de communication à l'intérieur d'un cluster, celui-ci peut cesser d'être cohérent. Lorsque c'est le cas, une partie des nœuds quitte ce cluster. La circulation du jeton permet ensuite de mettre à jour l'arbre couvrant du cluster, ainsi que sa taille. Ces mécanismes permettent de ramener le système dans une configuration valide. Le système peut alors démarrer une nouvelle phase de convergence vers la spécification.

Théorème 33. *Après un changement topologique, le système finit par se trouver dans une configuration valide.*

Pour démontrer ce théorème, nous nous appuyons sur une série de lemmes, séparant les différents cas possibles. Dans un premier temps, l'ajout d'un nœud ou d'un lien de communication dans le réseau ne peut en aucun cas rendre invalide la configuration du système.

Lemme 34. *Soit une configuration γ valide. Si la configuration γ' dérive de γ par l'ajout d'un lien de communication, ou par l'ajout d'un nœud, alors γ' est une configuration valide.*

Démonstration. La configuration γ est valide. Par conséquent, tous les clusters qu'elle comporte sont cohérents ou en destruction.

Si γ' se déduit de γ par l'ajout d'un lien de communication, celui-ci ne comporte aucun message. Les propriétés sur les messages des clusters cohérents ou en destruction restent donc vérifiées, et la configuration γ' est valide.

Si γ' se déduit de γ par l'ajout d'un nœud, celui-ci exécute la procédure d'initialisation (algorithme 19). En particulier, ce nœud est non clusterisé ($cluster = (\perp, 0)$). Par conséquent, les clusters cohérents de γ restent cohérents dans γ', et les clusters en destruction restent en destruction.

La configuration γ' est donc valide. □

La disparition d'un lien de communication peut rendre un cluster incohérent. Cependant, si le lien n'est pas une arête d'un arbre couvrant d'un cluster, sa disparition ne peut pas rendre la configuration invalide.

Lemme 35. *Soit une configuration γ valide. Si la configuration γ' dérive de γ par la disparition d'un lien de communication (e, i) tel que $father_e^\gamma \neq i$ et $father_i^\gamma \neq e$, alors la configuration γ' est valide.*

Démonstration. Soit C_x^γ un cluster. Puisque γ est une configuration valide, C_x^γ est soit cohérent, soit en destruction.

Si C_x^γ est cohérent, puisque $father_i \neq e$ et $father_e \neq i$, $\mathcal{A}_x^{\gamma'}$ reste un arbre couvrant de $C_x^{\gamma'}$. Par conséquent, $C_x^{\gamma'}$ reste connexe et $K_x^{\gamma'}$ reste dominant connexe.

Puisque $\mathcal{M}^{\gamma'} = \mathcal{M}^\gamma \setminus \left\{ \begin{array}{l} \left\{ (msg, param, e, i)/msg \in \left\{ \begin{array}{c} Cluster, Recruit, Delete, \\ Token, ACK, Transmit \end{array} \right\} \right\} \\ \cup \left\{ (msg, param, i, e)/ \left\{ \begin{array}{c} Cluster, Recruit, Delete, \\ Token, ACK, Transmit \end{array} \right\} \right\} \end{array} \right\}$, et puisque le lien (e, i) ne peut pas transporter le message $Token_x$ (par définition des clusters cohérents, $(Token, x, e, i) \in \mathcal{M} \Rightarrow father_i = e$ ou $father_e = i$), les propriétés que vérifient les messages d'un cluster cohérent restent vraies. Le cluster $C_x^{\gamma'}$ est donc cohérent.

Si C_x^γ est en destruction, de la même façon $F_x^{\gamma'}$ reste une forêt couvrante de C_x. Puisque C_x^γ est un cluster en destruction, pour chaque nœud $k \in K_x$ avec $father_k \notin C_x$, il existe un message $(Delete, x, father_k, k) \in \mathcal{M}^\gamma$. Comme $father_e \neq i$ et $father_i \neq e$, ces messages sont encore présents dans $\mathcal{M}^{\gamma'}$. Pour les mêmes raisons, il existe un message $Delete[x]$ ou un message $ACK[x]$ en transit sur les liens entre $(k, father_k)$ de tout nœud ordinaire ayant un père $father_k \notin C_x$. Le cluster $C_x^{\gamma'}$ est donc en destruction.

Par conséquent, la configuration γ' est valide. □

Ainsi, seule la disparition d'un lien appartenant à un arbre couvrant peut amener le système dans une configuration invalide. Cependant, lorsque c'est le cas, toute exécution de l'algorithme ramène le système dans une configuration valide. Si un nœud i possède une valeur pour sa variable $father_i$, cela signifie qu'il est clusterisé (cette variable est affectée à $\perp$ par les procédures *Init* et *Leave*, et n'est modifiée que par les procédures *JoinCore* et *JoinOrdinary*). Nous séparons les cas selon que le nœud i est membre d'un cluster cohérent, ou d'un cluster en destruction.

Lemme 36. *Soit γ une configuration valide. Si γ' dérive de γ par la suppression d'un lien (e, i) avec $father_i = e$ et $cluster_i$ un cluster en destruction, alors dans toute exécution il existe une configuration γ'' telle que $\gamma' \vdash^* \gamma''$ et telle que γ'' soit une configuration valide.*

Démonstration. Soit γ une configuration valide. La configuration γ' se déduit de γ par la suppression d'un lien de communication (e, i) tel que $father_i = e$. Puisque la variable $father_i$ a une valeur, le nœud i est forcément dans un cluster. Par hypothèse, le nœud i est dans un cluster en destruction. D'après notre modèle, le nœud i finit par exécuter la procédure *À la perte de la connexion vers e*, c'est-à-dire il existe deux configurations γ_1, γ_2 telles que $\gamma' \vdash^* \gamma_1$ et $\gamma_1 \vdash \gamma_2$ par l'exécution de la procédure *À la perte de la connexion vers e* par le nœud i.

Si le nœud i est un nœud de cœur $i \in K_x^\gamma$, alors $C_x^{\gamma'}$ n'est plus en destruction. Cependant, la seule propriété qui n'est plus vraie est le fait qu'un message $Delete[x]$ soit adressé à la racine de chaque arbre de $F_x^{\gamma'}$ qui contient au moins un nœud de cœur.

Durant $\gamma' \vdash^* \gamma_1$, le nœud i reste dans le cœur du cluster K_x. En effet, s'il reçoit un message
- *Recruit* : un tel message est ignoré (algorithme 5),
- *Delete* : ce message ne pouvant venir de son père, il est ignoré (algorithme 7),
- *ACK* : ce message ne peut pas déclencher le départ du nœud i (algorithme 9),
- *Token* : ce message ne peut déclencher le départ du nœud i (algorithme 10).

Lorsque le nœud exécute la procédure *À la perte de la connexion vers e* (algorithme 7), le nœud i, racine d'un arbre de $\mathcal{F}_x^{\gamma_1}$, quitte le cluster C_x et envoie un message $Delete[x]$ à tous ses voisins. Ainsi il y a bien un message $Delete[x]$ adressé à la racine de chaque arbre de $\mathcal{F}_x^{\gamma_2}$, et le cluster $C_x^{\gamma_2}$ redevient un cluster en destruction.

Si le nœud i est un nœud ordinaire, la seule propriété qui n'est plus vérifiée est le fait qu'il existe un message $ACK[x]$ ou un message $Delete[x]$ entre tout nœud ordinaire et son père. Les évènements suivants peuvent survenir durant $\gamma' \vdash^* \gamma_1$. Le nœud i peut recevoir un message :
- *Recruit* : étant déjà dans un cluster, il ignore ce message (algorithme 5),
- *Delete* : ce message ne pouvant provenir de son père $e = father_i$, il l'ignore (algorithme 7),
- *ACK* : ce message ne peut provoquer le départ du nœud i de C_x
- *Token* : ce message ne peut pas être le jeton $Token_x$, puisque C_x^γ est en destruction. Ainsi, si le nœud i est recruté par ce jeton, il quitte le cluster C_x qui redevient un cluster en destruction. De plus, le nœud i change de père (la variable $father_i$ est égale à l'identifiant du nœud auquel i envoie le jeton).

Lors de l'exécution de la procédure *À la perte de connexion*, si le nœud i est encore dans le cluster C_x, il quitte ce dernier et envoie un message $Delete[x]$ à tous ses voisins. Par conséquent, dans tous les cas, le nœud j finit par quitter le cluster C_x. Lorsque c'est le cas, ce cluster redevient un cluster en destruction. □

Lemme 37. *Soit une exécution $\gamma_0 \vdash^* \gamma_k \vdash^* \ldots$ telle que γ_0 soit une configuration valide et telle que γ_1 dérive de γ_0 par la suppression d'un lien (e, i) avec $father_i = e$.*

Soit $p \in \mathbb{N}$ tel que $\gamma_p \vdash \gamma_{p+1}$ par exécution de la procédure À la perte de la connexion vers *e par le nœud i.*

Alors, presque surement, il existe $q > p$ telle que $\forall j, k,\ \delta(Delete, x, j, k) = 0$.

Démonstration. On sait que p existe puisque chaque message finit par être reçu.

Soit $\mathcal{F}_x^\gamma = (C_x^\gamma,\ \{(l,m) \in E^\gamma / father_m = l\} \setminus \left\{(j,k) / \begin{array}{l}(Token_x, param, j, k) \in \mathcal{M}^\gamma \vee \\ (Token_x, param, k, j) \in \mathcal{M}^\gamma\end{array}\right\}$ la forêt couvrante du cluster C_x dans la configuration γ (par les mêmes arguments que pour la proposition 16, $\forall r > p+1$, $\mathcal{F}_x^{\gamma_r}$ est bien une forêt couvrante de C_x).

Soit $\mathcal{F}_D^{\gamma_r}$ la foret constituée de tous les arbres de $\mathcal{F}_x^{\gamma_r}$ tels qu'un message $Delete[x]$ est adressé à la racine. On appelle $\mathcal{A}^{\gamma_r}$ l'arbre de $\mathcal{F}_x^{\gamma_r}$ dont la racine a émis ou est destinataire du jeton $Token_x$.

Pour chaque configuration γ_r avec $r > p+1$, il n'y a qu'un nombre fini de nœuds de cœur dans $\mathcal{F}_D^{\gamma_r}$. Par les mêmes arguments que pour la proposition 30, tous les nœuds de $\mathcal{F}_D^{\gamma_r}$ vont finir par recevoir un message $Delete[x]$ et par quitter le cluster.

Nous décrivons maintenant une exécution possible, afin de montrer que la propagation de message $Delete[x]$ à une probabilité non nulle de s'arrêter.

À partir de γ_r, on construit une exécution dans laquelle le jeton $Token_x$ est bloqué aussi longtemps qu'on le souhaite entre deux nœuds. En effet, dans γ_r, le jeton $Token_x$ est en transit sur un lien $(j,k) \in E^{\gamma_r}$. Lorsque le nœud k reçoit le jeton, il exécute la procédure *À la réception d'un message Token*, puis il choisit un voisin uniformément au hasard et lui envoie le jeton. Il peut choisir j avec une probabilité $\frac{1}{deg_k}$. Le nœud j finira ensuite par recevoir le jeton, et par exécuter la procédure *À la réception d'un message Token*. Il choisit alors un voisin uniformément au hasard et lui envoie le jeton. Il peut choisir le nœud k avec une probabilité $\frac{1}{deg_j}$.

Le message jeton peut alors transiter l fois entre les nœuds j et k. Ce scénario a une probabilité de se produire supérieure à $\left(\frac{1}{d}\right)^l$ de se produire, où $d = max(deg_j, deg_k)$.

Le temps de destruction est le temps nécessaire pour que les messages $Delete[x]$ soient propagés le long de l'arbre $\mathcal{F}_D^{\gamma_r}$. Ce temps correspond au temps nécessaire pour recevoir h messages $Delete[x]$, où h est la profondeur de l'arbre $\mathcal{F}_D^{\gamma_r}$.

On sait, d'après notre modèle, que les durées de transit des messages $Token$ et $Delete$ sont de même ordre en probabilité : $\forall K, \exists \alpha > 0, P(t_{Delete[x]} > K \times t_{Token_x}) < \alpha$. En prenant, $K = \lfloor \frac{l}{h} \rfloor$, il existe donc α tel $P(t_{Delete[x]} > \lfloor \frac{l}{h} \rfloor \times t_{Token_x}) < \alpha$. Ainsi, durant les $\lfloor \frac{l}{h} \rfloor$ premières transmissions du jeton $Token_x$, le message $Delete[x]$ adressé à la racine de $\mathcal{F}_D^{\gamma_r}$ a une probabilité supérieure à $(1-\alpha)$ d'être reçu.

La propagation des messages $Delete[x]$ sur l'arbre $\mathcal{F}_D^{\gamma_r}$ nécessite de faire transiter h messages $Delete[x]$. Par conséquent, durant les l transmissions du jeton, la vague de $Delete[x]$ a une probabilité supérieure à $(1-\alpha)^h$ d'être propagée sur l'arbre $\mathcal{F}_D^{\gamma_r}$.

Puisque les tirages aléatoires des nœuds sont indépendants des temps de communication, la propagation des messages $Delete[x]$ à partir de γ_r s'arrête avec une probabilité supérieure à $\frac{1}{d}^l \times (1-\alpha)^{\lfloor \frac{l}{h} \rfloor} > \varepsilon > 0$.

Ce scénario a donc une probabilité supérieure à $\varepsilon > 0$ de se produire à partir de toute configuration γ_r avec $r > p+1$. Par la loi des grands nombres, ce scénario finira par se réaliser, et la propagation des messages $Delete[x]$ se termine presque surement.

□

Remarque 38. *Le scénario décrit dans la preuve précédente n'est pas le seul permettant la fin de la propagation des messages Delete[x]. En effet, un grand nombre de scénarios plus probables permettent également à cette propagation de prendre fin. Cela suffit pour conclure que la propagation prendra fin presque surement, mais serait très pessimiste si nous voulions évaluer la complexité de l'algorithme ou le temps nécessaire pour corriger le clustering suite à un changement topologique.*

En effet, pour toute configuration γ_r, il n'y a qu'un nombre fini de nœuds de cœur dans $\mathcal{F}_D^{\gamma_r}$.

La propagation de messages Delete[x] sur cet arbre cesse, sauf si le nombre de nœuds de cœur dans $\mathcal{F}_D$ a augmenté durant cette propagation. Or, la seule possibilité pour que le nombre de nœuds de cœur de $\mathcal{F}_D$ augmente est qu'un nœud de $\mathcal{F}_D$ reçoive le jeton $Token_x$.

Si un nœud k de $\mathcal{F}_D$ reçoit le jeton $Token_x$, ce nœud devient la racine de $\mathcal{A}_D$. Le jeton est envoyé à un voisin de k choisi uniformément au hasard. Si ce voisin n'est pas $father_k^{\gamma_r}$, alors k sort de $\mathcal{F}_D$.

Ainsi, pour que le nombre de nœuds de cœur dans $\mathcal{F}_D$ augmente, il faut que le jeton soit reçu par un nœud de $\mathcal{F}_D$ et "remonte" dans l'arbre couvrant jusqu'à ce que le jeton et un message

Delete$[x]$ *se retrouvent dans la même arête, dans des sens opposés :* $\exists(j,k),\ \delta(Token_x, param, j, k) = 1$ *et* $\delta(Delete, x, k, j) > 0$. *Cette chaîne d'évènement est peu probable.*

Lemme 39. *Soit* γ *une configuration valide. Si* γ' *dérive de* γ *par la suppression d'un lien* (e,i) *avec* $father_i = e$ *et* $cluster_i$ *un cluster cohérent, alors il existe* γ'' *telle que* $\gamma' \vdash^* \gamma''$ *et telle que* γ'' *soit une configuration valide.*

Démonstration. Soit γ une configuration valide et C_x^γ un cluster cohérent, et γ' une configuration déduite de γ par la suppression d'un lien de communication (i,e) tel que le nœud $i \in C_x^\gamma$ et $father_i^\gamma = e$.

Toutes les variables, sur tous les nœuds, conservent la même valeur entre γ et γ'. De plus, tous les messages en transit dans γ reste en transit dans γ', sauf les messages en transit sur le lien (i,e) qui sont retirés de $\mathcal{M}^{\gamma'}$: $\forall msg, \delta^{(\gamma')}(msg, y, i, e) = \delta^{(\gamma')}(msg, y, e, i) = 0$ et $\forall msg, \forall (p,q) \neq (i,j), \delta^{(\gamma')}(msg, y, p, q) = \delta^{(\gamma)}(msg, y, p, q)$.

On sait également, d'après le modèle, que le nœud i finit par exécuter la procédure *À la perte de la connexion vers e* (algorithme 6).

Par définition des clusters cohérents, soit l'arête enlevée est une arête de l'arbre couvrant $\mathcal{A}_x^\gamma$, soit le lien (i,e) porte le message $Token_x$.

1. Si le message $Token_x$ est en transit dans le lien (i,e), il est enlevé de $\mathcal{M}^{\gamma'}$. Le cluster $C_x^{\gamma'}$ finira par devenir un cluster en destruction. En effet, puisque C_x^γ est cohérent, $C_x^{\gamma'}$ vérifie toutes les propriétés des clusters en destruction, sauf le fait qu'il y ait un message *Delete*$[x]$ à destination du nœud i en provenance de son père e.

   Lorsque le nœud i exécute la procédure *À la perte de la connexion vers e*, il quitte le cluster C_x et envoie un message *Delete*$[x]$ à tous ses voisins. Le cluster C_x devient alors un cluster en destruction, un message *Delete*$[x]$ étant adressé à chaque racine de la forêt couvrant $\mathcal{F}_x$.
2. Si le lien (i,e) est une arête de l'arbre $\mathcal{A}_x^\gamma$, alors $C_x^{\gamma'}$ n'est plus cohérent. En effet, $\mathcal{A}_x^{\gamma'}$ n'est plus un arbre couvrant, mais une forêt couvrant comportant deux arbre. L'un de ces arbres, que l'on nomme $\mathcal{A}_1^{\gamma'}$, est enraciné sur le dernier nœud du cluster ayant eu le jeton. Le deuxième arbre, $\mathcal{A}_2^{\gamma'}$, est enraciné sur le nœud i. Le jeton $Token_x$ poursuit sa marche aléatoire dans le réseau. La partie du cluster dans l'arbre $\mathcal{A}_1$ vérifie toutes les propriétés des clusters cohérents, sauf que l'arbre couvrant n'est plus valide. Par les mêmes arguments que dans la démonstration du théorème 16, la partie de C_x dans l'arbre $\mathcal{A}_1$ continue de vérifier toutes les propriétés des clusters cohérents, ou vérifie celle des clusters en destruction si le jeton rencontre un nœud appartenant à un cluster incomplet d'identifiant plus grand.

Montrons que dans ce second cas le cluster C_x finit par être un cluster cohérent ou un cluster en destruction.

Si le nœud i est un nœud ordinaire, il peut être recruté en tant que nœud de cœur d'un autre cluster, être promu nœud de cœur de C_x, ou il finira par quitter C_x lorsqu'il exécute la procédure *À la perte de la connexion vers e*.

1. S'il quitte le cluster en exécutant la procédure *À la perte de la connexion vers e* ou en entrant dans un autre cluster, le cluster C_x finit par redevenir un cluster cohérent. En effet, toutes les propriétés des clusters cohérents sont vraies, et par propriété de percussion des marches aléatoires, le jeton $Token_x$ finira par visiter le nœud e, et l'arbre $\mathcal{A}$ sera alors corrigé en enlevant i (algorithme 14).
2. S'il est promu nœud de cœur de C_x, i devient alors la racine de $\mathcal{A}_x$, qui redevient alors un arbre couvrant de tout le cluster. C_x redevient alors un cluster cohérent.

Si le nœud i est un nœud de cœur de C_x, alors il ne peut pas changer de cluster. En exécutant la procédure *À la perte de la connexion vers* e, i quitte le cluster C_x et il initie la propagation d'une vague de messages $Delete[x]$ sur l'arbre $\mathcal{A}_2$. Nous démontrons au lemme 37 que la propagation des messages $Delete[x]$ termine presque surement.

Les évènements suivants sont alors possibles :

- le jeton $Token_x$ rencontre un cluster d'identifiant plus grand et est supprimé : C_x passe en destruction ;
- le jeton $Token_x$ rencontre un nœud de $\mathcal{A}_2$ qui n'a pas quitté C_x avant de visiter e ;
- le jeton $Token_x$ rencontre un nœud de $\mathcal{A}_2$ qui a quitté C_x avant de visiter e ;
- le jeton $Token_x$ ne rencontre aucun nœud de $\mathcal{A}_2$ avant de visiter le nœud e.

En effet, la propriété de percussion des marches aléatoires nous garantit que si le jeton $Token_x$ n'est pas détruit, il finira par visiter le nœud e.

1. Si le jeton $Token_x$ visite un nœud de cœur d'un cluster incomplet d'identifiant plus grand, ce nœud initie la destruction du cluster C_x. Comme dans le lemme précédent, le cluster C_x devient donc un cluster en destruction lorsque le nœud i exécute la procédure *À la perte de connexion.*

2. Si le jeton $Token_x$ ne visite aucun nœud de $\mathcal{A}_2^{\gamma'}$, alors tous les nœuds de $\mathcal{A}_2^{\gamma'}$ vont finir par recevoir un message $Delete[x]$, et par quitter le cluster C_x. Par la suite, lorsque le jeton $Token_x$ visite le nœud e, l'arbre couvrant $Token_x.tree$ est mis à jour par la procédure *UpdateStatus* (algorithme 14). Cette procédure enlève tout le sous arbre enraciné en i de $Token_x.tree$. Le cluster C_x redevient alors un cluster cohérent.

3. Si le jeton $Token_x$ visite un nœud k de $\mathcal{A}_2$ qui n'a pas quitté le cluster et qui est dans l'arbre $Token_x.tree$, ce nœud devient la racine de l'arbre $\mathcal{A}_1$. Tout le sous arbre enraciné en k fait alors partie de $\mathcal{A}_1$. Le nœud k envoie ensuite le jeton $Token_x$ à un voisin sélectionné au hasard, qui devient son père (algorithme 10). Ainsi, lorsque k reçoit un message $Delete[x]$ provenant de son père dans l'arbre $\mathcal{A}_2^{\gamma'}$, ce nœud n'est plus son père $father_k$. Par conséquent, comme le message ne vient pas de son père $father_k$, il est ignoré (algorithme 7). Tous ses descendants ignoreront d'éventuels messages $Delete[x]$, puisque ceux-ci ne peuvent venir de leur père. Lorsque le jeton visite le nœud e, le sous arbre enraciné en k ne fait plus partie de la descendance du nœud i. Par conséquent, ces nœuds encore présents dans le cluster ne sont pas enlevés de $Token_x.tree$. L'arbre $\mathcal{A}_x$ redevient un arbre couvrant de C_x, et le cluster C_x redevient cohérent.

4. Si le jeton $Token_x$ visite un nœud k de $\mathcal{A}_2$ qui a quitté le cluster, il exécute la procédure *UpdateStatus* qui enlève tout le sous-arbre enraciné en k de $Token_x.tree$. Puisque ce nœud a quitté C_x, c'est qu'il a déjà reçu un message $Delete[x]$ et qu'il l'a propagé sur ces descendants (algorithme 7). Nous montrons plus loin que tous ces descendants vont finir par quitter le cluster C_x.

5. Si le jeton $Token_x$ visite un nœud k qui n'a pas quitté le cluster, et qui n'est pas dans l'arbre $Token_x.tree$, c'est que le jeton a déjà visité un des ancêtres de k dans $\mathcal{A}_2$ qui a quitté le cluster. En effet, c'est le seul cas où il y a une mise à jour de l'arbre $Token_x.tree$. Cet ancêtre a initié la propagation d'une vague de messages $Delete[x]$ sur sa descendance. Dans ce cas, la procédure *UpdateStatus* (algorithme 14) force le nœud k à quitter le cluster et à envoyer un message $Delete[x]$ à ses voisins. Le nœud k peut ensuite être recruté. Le système se retrouve alors dans le même état que si le nœud k avait reçu un message $Delete[x]$ avant de recevoir le jeton $Token_x$. Cela nous garantit que tous les nœuds retirés de l'arbre $Token_x.tree$ par la procédure *UpdateStatus* vont bien quitter le cluster C_x.

L'arbre $Token_x.tree$ finit donc par redevenir un arbre couvrant de C_x, qui redevient un cluster cohérent.

Ainsi dans tous les cas, le cluster C_x finit par être un cluster cohérent ou en destruction, et le système se retrouve dans une configuration valide. □

(Preuve du théorème 33). D'après les lemmes 34 à 39, le système revient à une configuration valide après toutes modifications des liens de communication, ou après l'ajout d'un nœud au système.

La suppression d'un nœud du système peut être vue comme la suppression de tous les liens de communication le liant au système. Ainsi, après la suppression d'un nœud, le système finit par se retrouver dans une configuration valide. □

Ainsi, le système finit presque sûrement, en un temps fini après le dernier changement topologique, par rejoindre une configuration valide.

4.3.4 Conformité à la spécification

Le système commence toute exécution dans une configuration ne comportant que des nœuds endormis. À partir de cette configuration valide, le système se maintient dans une configuration valide, ou y retourne après un changement topologique. Lorsqu'il n'y a plus de changement topologique, le système converge vers la spécification.

Théorème 40. *Toute exécution $\mathcal{E} = (\gamma_1, \dots, \gamma_n, \dots)$ comportant un nombre fini de changements topologiques possède un suffixe infini de configurations vérifiant la spécification.*

Démonstration. Toute exécution commence dans une configuration valide. Après le dernier changement topologique, le système retourne dans une configuration valide d'après le théorème 33. Ensuite, le système converge vers une configuration vérifiant la spécification d'après le théorème 32, et s'y maintient d'après le théorème 16. □

4.4 Spécification : système virtuel

Nous donnons maintenant une définition formelle du fait qu'un clustering émule un système distribué exécutant un algorithme distribué $\mathcal{A}$.

Pour cela, chaque cluster doit se comporter comme un nœud. Les clusters doivent donc disposer de variables en tant que nœuds virtuels, et être capables de s'envoyer des messages entre eux. Ces variables, ainsi que les types de messages que les clusters peuvent s'échanger, sont définis par l'algorithme $\mathcal{A}$.

Si un algorithme émule un système distribué, il définit des variables qui représentent les variables des nœuds virtuels. Ces variables peuvent donc être extraites d'une configuration de l'algorithme.

De même, les messages en transit entre les nœuds virtuels apparaissent dans toute configuration γ du système de départ.

On peut donc construire, à partir de toute configuration γ du système de départ, une nouvelle configuration $f^{(1)}(\gamma)$ du système émulé. Cela définit une application extrayant une configuration du système émulé à partir d'une configuration du système initial.

Le clustering émule ainsi un système distribué lorsqu'une telle application existe, et lorsque le graphe de communication présent dans les configurations extraites est le graphe d'adjacence des

clusters $G^{(1)} = (V^{(1)}, E^{(1)})$ où $V^{(1)} = \{cluster_i / i \in V\}$ et $E^{(1)} = \left\{(x,y) \in {V^{(1)}}^2 / \exists (i,j) \in E, cluster_i = x, cluster_j = y\right\}$.

Au cours d'une exécution, la configuration du système de départ va changer, entrainant des modifications sur les configurations extraites. Puisque le clustering émule correctement un système distribué exécutant l'algorithme $\mathcal{A}$, les modifications apportées aux configurations extraites sont des transitions légales, définies par l'algorithme $\mathcal{A}$, ou des changements topologiques autorisés par le modèle. On note $\vdash_{\mathcal{A}}$ toute transition définie par l'algorithme distribué $\mathcal{A}$, et $\vdash^*_{\mathcal{A}}$ toute suite de transition de l'algorithme $\mathcal{A}$.

L'algorithme $\mathcal{A}$ est un algorithme distribué écrit pour des systèmes se comportant conformément au modèle exposé à la section 1.4. Cela signifie que toute exécution de l'algorithme sur un tel système, si elle comporte un nombre fini de changements topologiques, amène le système dans une configuration conforme à la spécification de $\mathcal{A}$.

Pour garantir le fonctionnement de l'algorithme $\mathcal{A}$, il faut donc vérifier que le système émulé se comporte conformément au modèle. Cela signifie que pour toute exécution sur le système de départ, les hypothèses du modèle doivent être vraies dans le système émulé :

- tout message émis finit par être reçu ;
- tout changement topologique finit par être détecté ;
- tous les nœuds finissent éveillés.

On dira donc qu'un clustering émule correctement un système distribué exécutant un algorithme $\mathcal{A}$ si la spécification suivante est vérifiée :

Définition 41 (spécification)**.** *Soit une exécution $\mathcal{E} = (\gamma_1, \gamma_2, \dots)$ contenant un nombre fini de changements topologiques. Cette exécution vérifie la spécification si et seulement si il existe une application $f^{(1)} : \mathcal{C} \mapsto \mathcal{C}^{(1)}$ telle que :*

Correspondance au clustering :

– il existe i tel que le graphe de communication de $f^{(1)}(\gamma_i)$ soit le graphe des clusters $G^{(1)}$.

Exécution induite :

- *Pour tout i,*
  - *soit $f^{(1)}(\gamma_i) = f^{(1)}(\gamma_{i+1})$;*
  - *soit $f^{(1)}(\gamma_i) \vdash^*_{\mathcal{A}} f^{(1)}(\gamma_{i+1})$;*
  - *soit $\exists \gamma_1, \gamma_2 \in \mathcal{C}^{(1)}, f^{(1)}(\gamma_i) \vdash^*_{\mathcal{A}} \gamma_1, \gamma_2 \vdash^*_{\mathcal{A}} f^{(1)}(\gamma_{i+1})$ et γ_1 résulte de γ_2 par un changement topologique.*
- *Il y a un nombre fini de changements topologiques dans cette exécution.*

Conformité au modèle :

- *Si $m \in \mathcal{M}^{\gamma_i(1)}$, il existe $j > i$ tel que entre $\mathcal{M}^{\gamma_j(1)}$ et $\mathcal{M}^{\gamma_{j+1}(1)}$, la multiplicité de m soit réduite de 1, et $f^{(1)}(\gamma_j) \vdash^*_{\mathcal{A}} f^{(1)}(\gamma_{j+1})$ avec une transition correspondant à la réception du message m.*
- *si deux clusters x et y voisin dans γ_i ne sont plus voisins dans la configuration γ_{i+1}, ils finissent par exécuter la procédure* à la perte de la connexion vers *de l'algorithme $\mathcal{A}$.*
- *si un nœud virtuel x est endormi dans $f^{(1)}(\gamma_i)$, il existe $j > i$ tel que x soit éveillé dans $f^{(1)}(\gamma_j)$, ou qu'il ait disparu.*

4.5 Transition et émulation

Dans notre algorithme, les actions des clusters en tant que nœud virtuel sont effectuées durant le traitement d'un message *Token*. Nous examinons dans cette section les transitions déclenchées par la réception d'un jeton *Token*, et leur impact sur le système émulé.

Lorsqu'un nœud i reçoit un message *jeton*, le système suit l'une des transitions $\gamma \vdash \gamma'$ définie par l'algorithme *À la réception d'un message Token* (algorithme 10). Les cinq cas suivants sont possibles :

- $i \in K_x^\gamma$: les variables sont mises à jour ;
- $i \in C_x^\gamma \setminus K_x^\gamma$ et $Token_x.size^\gamma \geq K$: les variables sont mises à jour ;
- $i \notin K_x^\gamma$ et $Token_x.size^\gamma < K$: le nœud entre dans $K_x^{\gamma'}$;
- $Token_x.size < K$, $\neg complete_i$ et $cluster_i > x$, ou alors $Token_x.size = 1$: le nœud initie la destruction du cluster C_x ;
- sinon : le jeton est renvoyé à l'expéditeur.

Ces transitions peuvent impacter le système distribué émulé (sauf le dernier cas).

Dans les trois autres cas, le nœud exécute la fonction *UpdateStatus* (algorithme 14). Cette fonction est utilisée dans l'algorithme de clustering pour mettre à jour l'arbre couvrant du cluster $Token_x.tree$.

La communication entre deux clusters C_x et C_y s'appuie sur la sélection d'un lien de communication privilégié, enregistré dans la variable $Token_x.gateway[y]$.

Après avoir mis à jour l'arbre couvrant $Token_x.tree$, la fonction *UpdateStatus* vérifie si les liens $Token_x.gateway$ sont encore accessibles. Cette fonction émule également l'exécution de la procédure *À la perte de la connexion vers* de l'algorithme $\mathcal{A}$ si c'est nécessaire.

La fonction *UpdateStatus* cherche enfin à ajouter de nouveaux liens de communication au système émulé. Si un nœud j voisin de i est dans un autre cluster, $y = gate_i[j] \neq x$. Dans ce cas, le lien (i, j) peut être utilisé comme lien passerelle entre les clusters C_x et C_y. Si $y > x$, *UpdateStatus* enregistre ce lien dans le tableau $Token_x.gateway$.

Ensuite, le nœud i exécute la procédure *TriggerUpperLevel* (algorithme 18). Cette procédure émule la réception des messages en attente dans la variable $listmsg_i$. Ces messages sont enregistrés avec l'identifiant du cluster émetteur, et l'identifiant du dernier nœud du cluster émetteur ayant relayé ce message. Si ce cluster est connu (c'est-à-dire dans $Token_x.gateway$), le nœud émule sa réception à l'aide de l'algorithme *À la réception d'un message* de l'algorithme $\mathcal{A}$ (ligne 9). Si l'émetteur du message est inconnu, il est d'abord ajouté à $Token_x.gateway$ (ligne 7). Si le message provient d'un voisin connu, mais par un lien différent de celui enregistré dans $Token_x.gateway$, le nœud i ignore ce message ou émule une perte de connexion entre le nœud virtuel et ce cluster (ligne 11 à 20).

Le système émulé peut donc subir un changement topologique, et peut suivre une ou plusieurs transitions définies par l'algorithme $\mathcal{A}$.

4.6 Démonstration émulation

Dans cette section, nous prouvons que le clustering émule le comportement d'un système distribué exécutant un algorithme distribué $\mathcal{A}$, c'est-à-dire que ce système distribué respecte la spécification présentée dans la section 4.4.

Cet algorithme $\mathcal{A}$ est un algorithme distribué écrit pour un système se comportant conformément à notre modèle. La détection du voisinage dans le système émulé s'appuie sur les communications entre les nœuds virtuels. Nous supposons donc qu'en exécutant l'algorithme $\mathcal{A}$, tous les nœuds du système envoient régulièrement des messages à chacun de leurs voisins. En particulier, cette hypothèse est vérifiée par notre algorithme de clustering. En effet, chaque nœud envoie un message *Cluster* à tous ses voisins à chaque fois qu'il reçoit le jeton de son cluster, c'est-à-dire infiniment souvent. Cela nous permettra de démontrer l'algorithme de clustering hiérarchique présenté à la section 3.4.

Dans un premier temps, nous définissons un nouveau système distribué, ainsi qu'une application f qui extrait une configuration de ce système à partir d'une configuration du système de départ.

Nous démontrons ensuite que ce système a un comportement conforme à notre modèle (section 1.4), et que son graphe de communication correspond bien au clustering calculé sur le système de départ.

Ensuite, nous montrons qu'au cours de toute exécution, les modifications apportées au système émulé correspondent à des transitions ou à des changements topologiques dans le système émulé. Toute exécution sur le système distribué initial induit ainsi une exécution sur le système émulé.

Cela permet de conclure que le système distribué initial finit presque sûrement par vérifier la spécification présentée à la section 4.4.

4.6.1 Système distribué virtuel

Dans le chapitre 3, nous avons présenté des mécanismes permettant à un cluster de se comporter comme un nœud virtuel. Les variables que possède un cluster C_x en tant que nœud virtuel sont contenues dans son jeton $Token_x$. Nous construisons donc un système distribué ne comportant que les clusters possédant un jeton.

Lorsqu'un cluster passe en destruction, celui-ci disparaîtra de ce système distribué, ce qui apparaîtra comme la déconnexion de ce nœud.

Cela nous amène à considérer le système distribué suivant :

Définition 42. *Soit une configuration γ. Nous considérons le système distribué composé des éléments suivants :*

- $\widetilde{G^{(1)}} = \left(\widetilde{V^{(1)}}, \widetilde{E^{(1)}}\right)$ *est un graphe non-orienté tel que* $\widetilde{V^{(1)}} = \{x / C_x^\gamma \text{ non en destruction}\}$

$$et\ \widetilde{E^{(1)}} = \left\{ (x,y) \in \left(\widetilde{V^{(1)}}\right)^2 / \begin{array}{c} C_x^\gamma \text{ et } C_y^\gamma \text{ voisins tel que } x \in Token_y.N, \\ y \in Token_x.N, \text{ et } \exists (i,j) \in V^\gamma \text{ tel que} \\ Token_x.gatway[y] = (i,j) \text{ et } Token_y.gateway[x] = (j,i) \end{array} \right\};$$

- *un algorithme distribué $\mathcal{A}$.*

L'ensemble d'arête $\widetilde{E^{(1)}}$ correspond à la relation de voisinage entre clusters, en ne considérant que les liens détectés par le système.

Une configuration de ce système distribué peut ainsi être obtenue à partir d'une configuration du système de départ en conservant uniquement les informations relatives au système émulé.

Définition 43. *Soit une configuration γ.*

On note $\gamma^{(1)} = f^{(1)}(\gamma)$ la configuration sur ce système distribué construite de la façon suivante :

- $\widetilde{G^{(1)}}$;
- $var_{C_x}^{\gamma^{(1)}} = Token_x.var^\gamma$;
- $\mathcal{M}^{\gamma^{(1)}} = \left\{ \begin{array}{c} (msg, param, x, y) / \exists (Transmit, (msg[param], path, x, y), i, j) \in \mathcal{M}^\gamma \\ tel\ que\ i \in C_x^\gamma,\ j \in path,\ et\ end(path) \in C_y^\gamma \end{array} \right\}$
$\bigcup \left\{ \begin{array}{c} (msg, param, x, y) / \exists i \in C_x^\gamma,\ j \in C_y^\gamma \\ tel\ que\ (msg, i, x) \in listmsg_j^\gamma\ et\ (i,j) \in E^\gamma \end{array} \right\}$

Les variables du système émulé sont portées par les jetons des clusters. Lorsqu'un nœud virtuel envoie un message dans le système émulé, celui-ci est encapsulé dans un message $Transmit$, et relayé le long d'un chemin jusqu'à un nœud du cluster destinataire. Le message est ensuite enregistré dans la variable *listmsg* de ce nœud. Lorsque le jeton du cluster destinataire visite ce nœud, il émule la réception par le nœud virtuel destinataire de tous les messages dans *listmsg*. Les messages en transit dans le système émulé sont donc soit encapsulés dans un message $Transmit$, soit en attente de traitement dans la variable *listmsg* d'un nœud.

Nous ne considérons, dans cet ensemble de messages, que les messages transportés sur un lien présent dans $\widetilde{G^{(1)}}$. Nous démontrerons plus loin que si d'autres messages sont encapsulés dans un message $Transmit$, ceux-ci n'arriveront pas à destination ou seront ignorés.

Ce système est dynamique, puisque le clustering évolue au cours du temps.

Nous avons désormais un système distribué et une fonction d'extraction des configurations. Montrons que ce système se comporte conformément à la spécification présentée en section, 4.4.

4.6.2 Compatibilité avec le modèle et correspondance au clustering

Nous montrons dans un premier temps que chaque nœud de ce système finit par s'éveiller. Cette propriété sera en effet nécessaire pour montrer que le graphe de communication du système émulé finit par être le graphe d'adjacence des clusters.

Proposition 44. *Chaque nœud x du système distribué émulé finit par s'éveiller, ou par quitter le système.*

Démonstration. Soit γ une configuration du système de départ.

Soit x un nœud endormi du système distribué émulé, présent dans la configuration $f^{(1)}(\gamma)$. Par définition du système émulé, le nœud x correspond à un cluster qui n'est pas en destruction, donc possédant un jeton $Token_x$, en transit dans la configuration γ. D'après les hypothèses de notre modèle, ce message jeton finira par être reçu, ou par disparaître pendant un changement topologique.

Si le jeton est reçu par un nœud de son cluster, ou par un nœud qu'il recrute, alors la fonction $TriggerUpperLevel$ (algorithme 18) est appelée. Soit cette fonction émule la réception d'un message par le nœud x, entraînant son éveil, soit cette fonction décrémente la variable T contenue dans le jeton. Si $Token_x.T = 0$, le nœud émule l'exécution de la procédure $wakeup()$ de l'algorithme $\mathcal{A}$ par le nœud virtuel x.

Si le jeton est reçu par un nœud extérieur à son cluster, qu'il ne peut recruter, alors il est simplement renvoyé à l'expéditeur.

Si le jeton est perdu pendant un changement topologique, le nœud x disparait du système émulé, ce qui apparait comme une déconnexion dans le système émulé.

Ainsi, par une récurrence sur la valeur de $Token_x.T$, un nœud finit par émuler l'exécution de la procédure $wakeup()$ de l'algorithme $\mathcal{A}$ par le nœud virtuel x. □

Nous démontrons maintenant ce système est cohérent avec le clustering.

Lemme 45. *Chaque nœud j finit par connaître les clusters qui lui sont adjacents.*

Démonstration. Dans un cluster C_x, d'après la propriété de percussion des marches aléatoires, chaque nœud i reçoit le jeton $Token_x$ et exécute la fonction *À la réception d'un message Token* (algorithme 10) infiniment souvent. À chaque fois, le nœud i envoie un message $Cluster[cluster_i]$ à tous ses voisins.

D'après la propriété 32, on sait qu'en l'absence de changement topologique, le nœud i finira par ne plus changer de cluster : dans toute exécution dépourvue de changement topologique $\mathcal{E} = (\gamma_1, \dots, \gamma_n, \dots)$, il existe une configuration γ_p telle que pour tout $q > p$, $cluster_i^{\gamma_q} = cluster_i^{\gamma_p}$.

À partir de la configuration γ_p, les messages $Cluster$ envoyés par le nœud i sont forcément d'identifiant $cluster_i$.

Le nœud i ne peut donc envoyer qu'un nombre fini de $Cluster$ ayant un identifiant différent de x. De plus, d'après notre modèle, tous ces messages $Cluster$ finiront par être reçus.

Ainsi, au bout d'un moment, les voisins du nœud i ne reçoivent plus que des messages $Cluster[cluster_i^{\gamma_p}]$, et enregistrent $gate[i] = cluster_i^{\gamma_p}$ (algorithme 13).

Chaque nœud finit donc par connaître le cluster auquel appartient chacun de ses voisins. □

Lemme 46. *Un message $Transmit$ bien formé, tant que les informations qu'il contient restent valides, finit par être reçu par un nœud appartenant au cluster de destination, c'est-à-dire si $(Transmit, (msg[param], path, x, y), e, i) \in \mathcal{M}^\gamma$ avec $path = i_1, \cdots, i_l$, tel que $i_1, \cdots, i_{l-1} \in C_x$, $i_l \in C_y$ et $\forall p \in \{1, \dots, l-1\}, (i_p, i_{p+1}) \in E^\gamma$, alors il existe une configuration γ' tel que $(msg[param], i_{l-1}, x) \in listmsg_{i_l}$.*

Démonstration. Soit une configuration γ et soit un message $(Transmit, (msg[param], path, x, y), e, i) \in \mathcal{M}^\gamma$ avec $path = i_1, \cdots, i_l$, $i_1, \cdots, i_{l-1} \in C_x^\gamma$, $i_l \in C_y^\gamma$ et $\forall p \in \{1, l-1\}, (i_p, i_{p+1}) \in E^\gamma$. Supposons que les nœuds $i_1, \cdots, i_l$ ne changent pas de cluster.

D'après notre modèle, le message $(Transmit, (msg[param], path, x, y), e, i)$ finit par être reçu. Lorsque c'est le cas :

- si $path = \emptyset$ (on a alors $i \in C_y$), alors i finit par recevoir le message $Transmit$. Il exécute alors l'algorithme 17 ligne 2 et conserve $(msg[param], e, x)$ dans $listmsg_j$;
- si $path \neq \emptyset$ (on a alors $i \in C_x$), i finit par recevoir le message $Transmit$. Il exécute l'algorithme 17 ligne 5. Ainsi $(Transmit, (msg[param], tail(path), x, y), i, head(path)) \in \mathcal{M}^{\gamma'}$.

Ainsi, par récurrence sur la longueur du chemin, et tant qu'il n'y a pas de changement topologique et qu'aucun nœud du chemin $path$ ne change de cluster, on montre que $msg[param]$ finit par être stocké sur le dernier nœud du chemin, dans la variable $listmsg$. □

Nous montrons maintenant que deux clusters adjacents finissent par être voisins dans le système émulé. Pour cela, nous commençons par montrer que si un lien invalide est enregistré dans la variable $Token_x.gateway$ d'un cluster, cette valeur finit par être effacée.

Lemme 47. *Soit C_x^γ un cluster tel que $Token_x^\gamma.gateway[y] = (j, k)$. Si $(j, k) \notin E^\gamma$, ou si $j \notin C_x^\gamma$, ou si $k \notin C_y^\gamma$, alors il existe alors une configuration γ' telle que $Token_x^{\gamma'}.gateway[y] = \perp$ et telle que le nœud virtuel x vient d'exécuter la procédure* À la perte de la connexion vers *y de l'algorithme $\mathcal{A}$.*

Démonstration. D'après le théorème 33, si C_x reste cohérent, $Token_x.tree$ finira par être un arbre couvrant à jour de C_x. Ainsi, dans le deuxième cas, j finira par être enlevé de $Token_x.tree$ par la fonction $UpdateStatus()$, algorithme 14 ligne 4, et alors $Token_x.gateway[y] \leftarrow \perp$. La procédure $UpdateStatus$ émule alors l'exécution de la procédure *À la perte de la connexion vers* y de l'algorithme $\mathcal{A}$ par le nœud virtuel x (ligne 21).

Si k n'est plus dans le cluster C_y, le nœud j finit par le savoir d'après la propriété 45. Par la propriété de percussion des marches aléatoires, le nœud j reçoit par la suite le jeton $Token_x$. Le nœud j exécute alors la procédure $UpdateStatus$, et que le nœud k soit dans un autre cluster, ou que le lien (j, k) ne soit plus dans E, cette fonction émule l'exécution de l'algorithme *À la perte de la connexion vers* y par le nœud virtuel x. De plus, $Token_x.gateway[y] \leftarrow \perp$.

Ainsi, dans tous les cas, le cluster x finit par exécuter la procédure *À la perte de la connexion vers* y, et la variable $Token_x.gateway[y]$ est affectée à $\perp$. □

Proposition 48. *Soit C_x et C_y deux clusters voisins. On finit par avoir $Token_x.gateway[y] = (j,k)$ et $Token_y.gateway[x] = (k,j)$ avec $(j,k) \in E$, $j \in C_x$ et $k \in C_y$, $x \in Token_y.N$ et $y \in Token_x.N$.*

Démonstration. Considérons deux clusters voisins C_x et C_y. Nous supposons que le système est dans une configuration où le clustering a convergé, et qu'il n'y a pas de changement topologique.

Supposons sans perte de généralité que $x > y$:

1. On finit par avoir $Token_x.gateway[y] \in E$:

   Si $Token_x.gateway[y]$ contient un lien non valide, alors d'après le lemme 47, on finit par avoir $Token_x.gateway[y] = \perp$.

   Si $Token_x.gateway[y] = \perp$, alors $\{i \in C_x / \exists j \in C_y, (i,j) \in E\} \neq \emptyset$. D'après la propriété de percussion des marches aléatoires, $Token_x$ finit par visiter un nœud α qui possède un voisin $\beta \in C_y$. D'après le lemme 45, $gate_\alpha[\beta] = y$. La fonction $UpdateStatus()$ (algorithme 14 ligne 15) ajoute y à $Token_x.N$, et modifie $Token_x.gateway[y] \leftarrow (\alpha, \beta) \in E$.

2. Ensuite, on a finalement $Token_y.gateway[x] = (\beta, \alpha)$:

   D'après la propriété 44, le nœud virtuel x finit par s'éveiller. Par hypothèse sur l'algorithme $\mathcal{A}$, ce nœud envoie régulièrement des messages à ses voisins une fois éveillé.

   Ainsi le nœud virtuel x finira par envoyer un message au nœud virtuel y puisque $y \in Token_x.N$. Lorsque x envoie un message à y, il utilise la fonction $\mathcal{A}.Envoyer\ msg\ à\ y(buffer)$ (algorithme 19) qui place le message à envoyer dans une file d'attente *buffer*. Ensuite, la fonction $TriggerUpperLevel$ récupère ce message, et l'encapsule dans un message $Transmit$ à l'aide de la fonction $Encapsulate$ (algorithme 15). Puisque $Token_x.gateway[y] = (\alpha, \beta)$ et que $\alpha \in Token_x.tree$, un chemin valide jusqu'à β est calculé et un message $Transmit$ bien formé est construit et envoyé. D'après le lemme 46, $(msg[param], \alpha, x)$ est finalement stocké dans $listmsg_\beta$.

   Ensuite, d'après la propriété de percussion des marches aléatoires, $Token_y$ finit par rendre visite au nœud β, et $TriggerUpperLevel()$ émule la réception de ce message par le nœud virtuel y (algorithme 18).

   Si le nœud x est inconnu du nœud y, ce dernier ajoute x à $Token_y.N$ et modifie $Token_y.gateway[x] \leftarrow (\beta, \alpha)$. Si le nœud x est connu, mais avec que $Token_y.gateway[x] \neq (\beta, \alpha)$, alors le nœud j émule l'exécution de l'algorithme *À la perte de la connexion vers* x de l'algorithme $\mathcal{A}$ par le nœud virtuel y, et $Token_y.gateway[x]$ est mis à jour (ligne 11 à 17).

Ainsi, tôt ou tard, un lien (α, β) est choisi comme lien de communication privilégié, tel que $Token_x.gateway[y] = (\alpha, \beta)$ et $Token_y.gateway[x] = (\beta, \alpha)$, $C_y \in Token_x.N$ et $C_x \in Token_y.N$. □

Corollaire 49. *Si deux clusters C_x et C_y sont adjacents, alors $\exists \gamma$ tel que $(x,y) \in E^{\gamma^{(1)}}$. En particulier, cela signifie que $\widetilde{G^{(1)}}$ finit par être égal à $G^{(1)}$.*

Ainsi, le système distribué définit à la section 4.6.1 finit bien par correspondre au clustering.

Pour s'assurer que ce système se comporte bien conformément au modèle, il nous reste à vérifier que tout message émis, en l'absence de changement topologique, finit par être reçu, et que tout changement topologique finit par être détecté par les nœuds adjacents.

Proposition 50. *Quand un lien de communication du système virtuel disparaît, les deux nœuds qu'il relie finissent par exécuter la fonction* À la perte de la connexion vers *de l'algorithme* $\mathcal{A}$.

Démonstration. Soit deux nœuds x et y du système émulé, voisins, et tels que $x > y$. Par définition, il existe $(i, j) \in V$ tel que $Token_x.gateway[y] = (i, j)$ et $Token_y.gateway[x] = (j, i)$. Si un changement topologique survient dans le système émulé, supprimant le lien de communication entre C_x et C_y, cela signifie que au moins l'un des clusters a retiré le lien (i, j) de sa variable $Token.gateway$.

Si $C_x^{\gamma'}$ et $C_y^{\gamma'}$ ne sont plus reliés par le lien (i, j) dans une configuration γ', cela signifie que le lien (i, j) n'apparaît pas dans $E^{\gamma'}$, que $i \notin C_x^{\gamma'}$ ou que $j \notin C_y^{\gamma'}$.

D'après le lemme 47, alors les deux nœuds virtuels vont finir par exécuter la procédure *À la perte de la connexion vers* de l'algorithme $\mathcal{A}$.

Les seuls procédures modifiant la variable $Token.gateway$ sont la procédure $UpdateStatus$ (algorithme 14) et la procédure $TriggerUpperLevel$ (algorithme 18). Si $Token_x.gateway[y] = (i, j)$, $UpdateStatus$ ne peut changer cette valeur que si le nœud i n'est plus dans l'arbre couvrant $Token_x.tree$, ou lors de la réception du jeton $Token_x$ par le nœud i, si j n'est plus voisin (ligne 17). Dans les deux cas, les clusters exécuteront la procédure *À la perte de la connexion vers* d'après ce qui précède.

La procédure $TriggerUpperLevel$ ne peut modifier $Token_x.gateway[y]$, puisque $x > y$ (ligne 11). La procédure $TriggerUpperLevel$ ne modifie la variable $Token_y.gateway[x]$ que lors de la réception d'un message venant de x par un autre lien que (i, j). Cela signifie que la valeur $Token_x.gateway[y] \neq (i, j)$, et par conséquent, puisque $Token_x.gateway[y] \neq Token_y.gateway[x]$, les nœuds x et y ne sont pas voisins dans le système émulé, c'est-à-dire $(x, y) \notin E^{f^{(1)}(\gamma)}$.

Ainsi, si un lien est enlevée du système distribué émulé, les deux nœuds adjacents finissent par exécuter la procédure *À la perte de la connexion vers.*

De la même façon, la disparition d'un nœud dans le système distribué virtuel finira par être détectée par les nœuds virtuels adjacents. □

Les changements topologiques finissent donc par être détectés par les nœuds du système virtuel. Nous montrons maintenant que les messages émis finissent par être reçus. Pour cela, nous avons besoin du lemme suivant.

Lemme 51. *Soit* C_x *et* C_y *deux clusters voisins dans le système. Soit* $(i, j) = Token_x.gateway[y]$. *Si le nœud virtuel* x *envoie un message au nœud virtuel* y *et que ces nœuds restent voisins, alors le message finit par être enregistré dans la variable* $listmsg_j$.

Démonstration. Soient deux clusters C_x^γ et C_y^γ voisins dans le système distribué, c'est-à-dire $(x, y) \in E^{f^{(1)}(\gamma)}$. On suppose que le nœud virtuel x envoie un message au nœud virtuel y.

L'envoi d'un message par le nœud virtuel x se produit durant l'exécution de la procédure $TriggerUpperLevel$, durant le traitement du message $Token_x$ par un nœud $i \in C_x^\gamma$. La procédure $Encapsulate$ (algorithme 15) calcule, à l'aide des informations présentes dans le jeton, un chemin $path$ jusqu'au lien de communication $(j, k) = Token_x.gateway[y]$. Le message émis est ensuite encapsulé dans un message $Transmit$, ainsi que ce chemin.

Soit le message $Transmit$ est bien formé, c'est-à-dire $(Transmit, msg[param], path, x, y)$ avec $path = i_1, \ldots, i_{l-1} \in C_x^\gamma, i_l \in C_y^\gamma$, et $\forall p \in \{1, \ldots, l-1\}(i_p, i_{p+1}) \in E^\gamma$. Dans ce cas, d'après la proposition 46, et sous réserve que le chemin reste valide, il existe une configuration γ' dans l'exécution telle que $(msg[param], i_{l-1}, x) \in listmsg_{i_l}$.

Si le chemin est ou devient invalide, alors cela signifie que deux nœuds du chemin ne sont plus voisins, ou qu'un nœud du chemin a changé de cluster. Ce chemin étant un chemin dans

l'arbre $Token_x.tree^\gamma$, cet arbre n'est plus un arbre couvrant du cluster. Or, d'après le théorème 33, le système revient à une configuration valide. Par conséquent, l'arbre $Token_x.tree$ finit par être mis à jour, et la passerelle $Token_x.gateway[y]$ est alors effacée (procédure *UpdateStatus*, algorithme 14). Par définition du système distribué émulé, les nœuds virtuels x et y ne sont alors plus voisins.

Ainsi, si les nœuds restent voisins dans le système émulé, le message finit par être enregistré dans la variable $listmsg_{i_l}$ du nœud passerelle appartenant à C_y. □

Ce lemme nous permet de montrer que tout message émis au niveau supérieur finit par être reçu en l'absence de changement topologique.

Proposition 52. *Tant que deux clusters restent voisins dans le système virtuel, tout message envoyé de l'un à l'autre finira par être reçu.*

Démonstration. Soient deux clusters C_x^γ et C_y^γ voisins dans le système distribué, c'est-à-dire $(x, y) \in E^{f^{(1)}(\gamma)}$. On suppose que le nœud virtuel x envoie un message au nœud virtuel y.

D'après le lemme 51, si les nœuds restent voisins, le message émis finit par être enregistré dans la variable *listmsg* du nœud passerelle appartenant à C_y.

Par la suite, par propriété de percussion des marches aléatoires, si j reste dans le cluster C_y, il finit par recevoir le jeton $Token_y$. Le nœud j exécute alors la procédure $TriggerUpperLevel$ (algorithme 18) qui émule la réception des messages en attente dans la variable $listmsg_j$ par le nœud virtuel y. De plus, si le nœud j quitte le cluster C_y, alors par définition du système émulé, les nœuds x et y ne sont plus voisins. Par conséquent, tout message émis finit par être reçu, sauf en cas de changement topologique dans le système émulé. □

Nous avons donc montré que le système virtuel définit dans la section 4.6.1 est un système distribué se comportant de manière conforme au modèle présenté dans la section 1.4.

4.6.3 Exécution sur le système distribué

Toute exécution de l'algorithme sur le système distribué induit des transformations du système distribué émulé. Nous démontrons ici que chacune de ces transformations est soit une transition de l'algorithme distribué $\mathcal{A}$, soit un changement topologique autorisé par le modèle, formant ainsi une exécution de l'algorithme $\mathcal{A}$ sur le système distribué émulé.

Montrons dans un premier temps que toute transition de notre algorithme induit un segment d'exécution (éventuellement vide) de l'algorithme $\mathcal{A}$ sur le système émulé (théorème 60).

La réception d'un message *Recruit* ne modifie pas le système émulé. Lorsqu'un nœud est recruté dans un cluster C_x, ce cluster peut avoir un nouveau cluster voisin C_y. Cependant, tant que ce voisin n'est pas détecté par le cluster C_x (c'est-à-dire $y \in Token_x.N$), le lien de communication n'apparaît pas dans le système émulé. Ce lien finira par être détecté d'après la propriété 48.

Lemme 53. *Si* $\gamma \vdash_{Recruit[x]} \gamma'$, *alors* $f^{(1)}(\gamma) = f^{(1)}(\gamma')$.

Démonstration. Les variables des nœuds virtuels sont portées par les messages $Token$. Par conséquent, la réception d'un message $Recruit[x]$ laisse les variables du système émulé inchangées : $\forall var, \forall x \in \tilde{V}^{f^{(1)}(\gamma')} = \tilde{V}^{f^{(1)}(\gamma)}, var_x^{f^{(1)}(\gamma')} = var_x^{f^{(1)}(\gamma)}$

Lorsqu'un nœud i reçoit un message $Recruit[x]$, il exécute la procédure *JoinOrdinary* (algorithme 5).

De plus, pour tout cluster C_x, $Token_x.N^{\gamma'} = Token_x.N^{\gamma}$ et $Token_x.gateway^{\gamma'} = Token_x.gateway^{\gamma}$. Par conséquent, $\tilde{E}^{f^{(1)}(\gamma')} = \tilde{E}^{f^{(1)}(\gamma)}$ et $\tilde{G}^{f^{(1)}(\gamma')} = \tilde{G}^{f^{(1)}(\gamma)}$

Puisque la procédure *JoinOrdinary* ne modifie pas la variable $listmsg_i$, et puisqu'aucun message $Transmit$ n'est émis, $\mathcal{M}^{\gamma'^{(1)}} = \mathcal{M}^{\gamma^{(1)}}$.

On en déduit que $f^{(1)}(\gamma') = f^{(1)}(\gamma)$. □

De la même façon, la réception d'un message $Cluster[x]$ n'a aucun impact sur la configuration du système émulé. Si ce message révèle un changement dans le graphe d'adjacence des clusters, celui-ci n'apparaîtra dans le système émulé que lorsque le jeton $Token_x$ sera modifié. La propriété 48 nous garantit que ce sera bien le cas.

Lemme 54. *Si* $\gamma \vdash_{Cluster[x]} \gamma'$, *alors* $f^{(1)}(\gamma') = f^{(1)}(\gamma)$.

Démonstration. Lors de la réception d'un message $Cluster[x]$ sur un nœud i, ce nœud modifie uniquement sa variable $gate_i$ et n'envoie aucun message (algorithme 13). Par conséquent, l'ensemble des messages en transit dans le système émulé ne change pas : $\mathcal{M}^{\gamma'^{(1)}} = \mathcal{M}^{\gamma^{(1)}}$.

De même, les variables des nœuds du système émulé ne changent pas : $\forall var, \forall x \in \tilde{V}^{f^{(1)}(\gamma')} = \tilde{V}^{f^{(1)}(\gamma)}, var_x^{f^{(1)}(\gamma')} = var_x^{f^{(1)}(\gamma)}$.

Enfin, pour tout cluster C_x, $N_x^{f^{(1)}(\gamma')} = Token_x.N^{\gamma'} = Token_x.N^{\gamma} = N_x^{f^{(1)}(\gamma)}$ et $Token_x.gateway^{\gamma'} = Token_x.gateway^{\gamma}$.

Par conséquent, puisque $G^{\gamma'} = G^{\gamma}$, le graphe de communication du système émulé ne change pas : $G^{f^{(1)}(\gamma')} = G^{f^{(1)}(\gamma)}$, et ainsi $f^{(1)}(\gamma') = f^{(1)}(\gamma)$.

□

La réception d'un message $ACK[x]$ n'a également aucun impact sur la configuration du système distribué émulé.

Lemme 55. *Si* $\gamma \vdash_{ACK[x]} \gamma'$, *alors* $f^{(1)}(\gamma') = f^{(1)}(\gamma)$.

Démonstration. Lorsqu'un nœud i reçoit un message $ACK[x]$, il exécute l'algorithme 9. Soit il l'ignore, soit il envoie un message $Delete[x]$ à l'expéditeur.

Dans tous les cas, les variables des nœuds virtuels sont inchangées : $\forall var, \forall x \in \tilde{V}^{f^{(1)}(\gamma')} = \tilde{V}^{f^{(1)}(\gamma)}, var_x^{f^{(1)}(\gamma')} = var_x^{f^{(1)}(\gamma)}$.

De même, les messages en transit dans le système émulé, ainsi que son graphe de communication, sont inchangés.

Par conséquent, $f^{(1)}(\gamma') = f^{(1)}(\gamma)$. □

La réception d'un message $Transmit$ laisse le système émulé inchangé. En effet, même si le message est enregistré sur un nœud du cluster destinataire C_y, il ne sera reçu que lorsque le jeton $Token_y$ visite ce nœud.

Lemme 56. *Si* $\gamma \vdash_{Transmit} \gamma'$, *alors* $f^{(1)}(\gamma') = f^{(1)}(\gamma)$.

Démonstration. Lorsque un nœud i reçoit un message $Transmit[msg, path, x, y]$, il exécute la procédure *À la réception d'un message Transmit*, algorithme 17.

Si le message encapsulé est relayé au suivant sur le chemin (ligne 2), ou s'il est enregistré dans la variable $listmsg_i$, alors par définition du système distribué émulé, le message reste dans l'ensemble des messages en transit $\mathcal{M}^{f^{(1)}(\gamma')} = \mathcal{M}^{f^{(1)}(\gamma)}$.

Dans les autres cas, le message est ignoré. Or, d'après la propriété 46, si le lien de communication est encore présent dans le système émulé ($(x,y) \in E^{f^{(1)}(\gamma)}$), le message encapsulé finit par être enregistré dans la variable *listmsg* d'un nœud du cluster C_y. Par conséquent, cela signifie que le $(x,y) \notin E^{f^{(1)}(\gamma)}$. Par conséquent, par définition du système émulé, ce message n'est pas dans le multi-ensemble $\mathcal{M}^{f^{(1)}(\gamma)} = \mathcal{M}^{f^{(1)}(\gamma)}$.

Ainsi, dans tous les cas, $f^{(1)}(\gamma') = f^{(1)}(\gamma)$. □

La réception d'un message $Delete[x]$ peut modifier le cluster C_x. Ainsi, si un nœud adjacent à l'arête $Token_x.gateway[y]$ quitte son cluster, le lien (x,y) est enlevé du système émulé. Dans tous les autres cas, le système émulé est inchangé.

Pour démontrer cela, nous avons besoin du lemme suivant.

Lemme 57. *Soit un nœud $i \in V^\gamma$. Les messages en attente dans la liste $listmsg_i^\gamma$ sont à destination du nœud virtuel x.*

Démonstration. La variable $listmsg_i$ d'un nœud i ne peut être modifiée que par les procédures *Leave* (algorithme 6), *À la réception d'un message Transmit* (algorithme 17) et *TriggerUpperLevel* (algorithme 18).

Lorsqu'un nœud i reçoit un message $Transmit[msg, path, x, y]$ venant d'un nœud e, il enregistre (msg, e, x) dans $listmsg_i$ (ligne 2) seulement si $i \in C_y$, c'est à dire si $cluster_i = y$.

Le nœud i ne peut changer de cluster que lorsqu'il reçoit un message $Delete[y]$ (algorithme 7), lorsqu'il détecte une perte de connexion avec son père (algorithme *À la perte de la connexion vers*), ou s'il est un nœud ordinaire recruté dans le cœur d'un autre cluster (ligne 20 à 29 algorithme 10). Dans tous les cas, le nœud exécute la procédure *Leave*, qui vide la liste $listmsg_i$.

Ainsi, les messages en attente dans la liste $listmsg_i$ sont tous adressés au cluster $cluster_i$. □

Lemme 58. *Si $\gamma \vdash_{Delete[x]} \gamma'$, alors soit $f^{(1)}(\gamma') = f^{(1)}(\gamma)$, soit $f^{(1)}(\gamma')$ résulte de $f^{(1)}(\gamma)$ par la perte d'un lien de communication.*

Démonstration. Par définition, il n'y a pas de message $Delete[x]$ en circulation dans un cluster cohérent. Par conséquent, si un message $Delete[x]$ est en transit, c'est soit que le cluster C_x est en destruction, soit qu'un changement topologique a eu lieu au sein de C_x. Lorsqu'un nœud i reçoit un message $Delete[x]$, il exécute la procédure *À la réception d'un message Delete* (algorithme 7).

Si ce message est ignoré, alors $f^{(1)}(\gamma') = f^{(1)}(\gamma)$.

Si le message à été envoyé par le nœud $e = father_i$, le nœud i quittent le cluster C_x et vide la liste des messages en attente $listmsg_i$.

Si le cluster C_x est en destruction, alors par définition ce cluster ne fait pas partie du système émulé. Or, d'après le lemme 57, les messages enregistrés dans la variable *listmsg* sont adressés au cluster C_x. Par conséquent, ces messages ne sont pas dans le multi-ensemble $\mathcal{M}^{f^{(1)}(\gamma)} = \mathcal{M}^{f^{(1)}(\gamma)}$, et $f^{(1)}(\gamma') = f^{(1)}(\gamma)$.

S'il existe y et l tel que $Token_x.gateway[y] = (i,l)$, alors par définition du système émulé, les nœuds x et y ne sont plus voisins dans le système émulé, et $f^{(1)}(\gamma')$ résulte de $f^{(1)}(\gamma)$ par la disparition du lien de communication (x,y).

Dans les autres cas, par définition, le graphe de communication du système émulé est inchangé, c'est-à-dire $G^{f^{(1)}(\gamma')} = G^{f^{(1)}(\gamma)}$. De plus, si un message est enlevé de $listmsg_i$, d'après le lemme 51 cela signifie que i est utilisé comme passerelle vers un cluster voisin, c'est-à-dire qu'il existe y et l tel que $Token_y.gateway[x] = (l,i)$, ce qui nous ramène au cas précédent. Les messages enlevés sont donc dans un canal de communication qui disparait de $\widetilde{E}$. □

C'est lors de la réception d'un jeton $Token$ que l'exécution sur le système va avoir lieu. En effet, lorsque le jeton est reçu par un nœud i de son cluster, la fonction $UpdateStatus$ est appelée et vérifie s'il y a eu des changements topologiques. Si ces changements ont un impact sur le système émulé, le nœud i émule l'exécution de la procédure *À la perte de la connexion vers* de l'algorithme $\mathcal{A}$ par le nœud virtuel x. Ensuite, les messages en attente dans la variable $listmsg_i$ sont traités l'un après l'autre, la fonction $TriggerUpperLevel$ émulant leur réception par le nœud virtuel x. Ainsi, selon s'il y a ou non des messages en attente, et selon les éventuels changements topologiques, le système émulé reste dans la même configuration ou la nouvelle configuration résulte de la précédente par l'exécution d'une séquence de transition de l'algorithme $\mathcal{A}$ sur le système émulé.

Lemme 59. *Si $\gamma \vdash_{Token_x} \gamma'$, alors $f^{(1)}(\gamma') = f^{(1)}(\gamma)$, $f^{(1)}(\gamma) \vdash^*_{\mathcal{A}} f^{(1)}(\gamma')$ ou il existe ω, ω' telle que $f^{(1)}(\gamma) \vdash^*_{\mathcal{A}} \omega$, $\omega' \vdash^*_{\mathcal{A}} f^{(1)}(\gamma)$ et ω' se déduit de ω par un changement topologique.*

Démonstration. Lorsqu'un noeud i reçoit un jeton $Token_x$, il exécute la fonction *À la réception d'un message Token*, (algorithme 10). Les cas suivants sont possibles :

- le jeton est renvoyé à l'expéditeur,
- le nœud i initie la destruction du cluster C_x,
- le nœud i est dans K_x,
- le nœud i est un nœud ordinaire non recruté,
- le nœud i est recruté dans le cœur K_x.

Si le jeton est renvoyé à l'expéditeur, la configuration du système distribué émulé ne change pas, c'est-à-dire $f^{(1)}(\gamma') = f^{(1)}(\gamma)$.

Si le nœud i initie la destruction du cluster C_x, celui-ci devient un cluster en destruction et disparaît de $G^{f^{(1)}(\gamma')}$. Par conséquent, $f^{(1)}(\gamma')$ se déduit de $f^{(1)}(\gamma)$ par la déconnexion d'un nœud.

Si le nœud i est dans K_x, ou s'il est un nœud ordinaire non recruté (ligne 2 à 12 et ligne 14 à 19), alors le nœud i exécute la fonction $UpdateStatus$, puis la fonction $TriggerUpperLevel$. La fonction $UpdateStatus$ (algorithme 14) vérifie si les clusters voisins sont encore accessibles via l'arbre $Token_x.tree$ (ligne 16 à 21), puis émule éventuellement l'exécution de la fonction *À la perte de la connexion vers* de l'algorithme $\mathcal{A}$ par le nœud virtuel x. Ensuite, cette fonction détecte les éventuels nouveaux voisins en parcourant la variable $gate_i$ (ligne 23 à 26). Ces étapes peuvent modifier le graphe de communication du système distribué émulé.

La fonction $TriggerUpperLevel$ émule ensuite la réception des messages en attente dans la variable $listmsg_i$ (ligne 4 ou 9, algorithme 18). Durant cette étape, toutes les modifications apportées au système émulé correspondent donc à des transitions de l'algorithme $\mathcal{A}$, exécutées de manière atomique par le nœud $cluster_i$ du système émulé. Par conséquent, $f^{(1)}(\gamma') = f^{(1)}(\gamma)$, $f^{(1)}(\gamma) \vdash^*_{\mathcal{A}} f^{(1)}(\gamma')$ ou il existe ω, ω' telle que $f^{(1)}(\gamma) \vdash^*_{\mathcal{A}} \omega$, $\omega' \vdash^*_{\mathcal{A}} f^{(1)}(\gamma)$ et ω' se déduit de ω par un changement topologique.

Si le nœud i est recruté dans K_x (ligne 21 à 29, algorithme 10) et ne change pas de cluster, l'impact sur le système distribué émulé est le même que dans le cas précédent. Si le nœud change de cluster, les messages en attente dans la liste $listmsg_i$ sont supprimés. Ensuite, comme précédemment, les fonctions $UpdateStatus$ et $TriggerUpperLevel$ sont appelées, et soit $f^{(1)}(\gamma') = f^{(1)}(\gamma)$, $f^{(1)}(\gamma) \vdash^*_{\mathcal{A}} f^{(1)}(\gamma')$, soit il existe ω, ω' telle que $f^{(1)}(\gamma) \vdash^*_{\mathcal{A}} \omega$, $\omega' \vdash^*_{\mathcal{A}} f^{(1)}(\gamma)$ et ω' se déduit de ω par un changement topologique.

Par conséquent, dans tous les cas, $f^{(1)}(\gamma') = f^{(1)}(\gamma)$, $f^{(1)}(\gamma) \vdash^*_{\mathcal{A}} f^{(1)}(\gamma')$ ou il existe ω, ω' telle que $f^{(1)}(\gamma) \vdash^*_{\mathcal{A}} \omega$, $\omega' \vdash^*_{\mathcal{A}} f^{(1)}(\gamma)$ et ω' se déduit de ω par un changement topologique. □

Ces lemmes nous permettent de démontrer que lors de toute transition, les modifications apportées au système émulé forment une exécution de l'algorithme $\mathcal{A}$.

Proposition 60. *Si $\gamma \vdash \gamma'$, alors $f^{(1)}(\gamma) = f^{(1)}(\gamma')$ ou $f^{(1)}(\gamma) \vdash_{\mathcal{A}}^{*} f^{(1)}(\gamma')$ ou $\exists \omega, \omega'$ deux configurations du système virtuel, telles que $f^{(1)}(\gamma) \vdash_{\mathcal{A}}^{*} \omega$, $\omega' \vdash_{\mathcal{A}}^{*} f^{(1)}(\gamma')$ et où ω' dérive de ω par un changement topologique.*

Il nous reste à vérifier que lors d'un changement topologique, le système émulé se comporte conformément à l'algorithme $\mathcal{A}$.

L'ajout d'un lien de communication peut éventuellement faire apparaître de nouveaux liens de communication dans le système émulé. Cependant, cela ne se produira que lorsque ces liens seront détectés par la visite d'un jeton sur l'un des nœuds qui leur sont adjacents. Ainsi, $f^{(1)}(\gamma') = f^{(1)}(\gamma)$.

Lorsqu'un nœud est ajouté au système, il exécute la procédure *Init* (algorithme 19) et est non clusterisé. Par conséquent, $f^{(1)}(\gamma') = f^{(1)}(\gamma)$.

Lemme 61. *Si γ' se déduit de γ par l'ajout d'un lien de communication, ou par l'ajout d'un nœud, alors $f^{(1)}(\gamma) = f^{(1)}(\gamma')$.*

Démonstration. L'ajout d'un nœud ou d'un lien de communication ne modifie pas la valeur des variables dans les jetons. Ainsi, pour tout cluster C_x, $Token_x^{f^{(1)}(\gamma')}.gateway = Token_x^{f^{(1)}(\gamma)}.gateway$, et le graphe de communication du système émulé est inchangé.

Par conséquent, $f^{(1)}(\gamma') = f^{(1)}(\gamma)$.

Par contre, $\widetilde{G^{(1)}} \neq G^{(1)}$. On finira par avoir $\widetilde{G^{(1)}} = G^{(1)}$ d'après la propriété 49, donc il existe γ' et γ'' tels que $\gamma \vdash^{*} \gamma' \vdash \gamma''$ avec $G^{f^{(1)}(\gamma')} \neq G^{f^{(1)}(\gamma'')} = G^{(1)}$. □

Lorsqu'un lien (i, j) de communication disparaît dans le système de départ, cela peut entraîner la disparition d'un lien de communication dans le système émulé si ce lien est une arête de l'arbre couvrant d'un cluster, ou si ce lien est la passerelle choisie entre deux clusters. Cependant, tant que cette perte n'est pas détectée, le graphe de communication du système distribué est inchangé. La proposition 48 affirme qu'un tel changement topologique finira par être détecté et traité.

Lemme 62. *Si γ' se déduit de γ par la suppression d'un lien de communication (i, j) tel que $Token_x.gateway[y] \neq (i, j)$ et $Token_y.gateway[x] \neq (i, j)$, alors $f^{(1)}(\gamma') = f^{(1)}(\gamma)$.*

Démonstration. Lors d'un changement topologique, les variables sur les nœuds ne changent pas. Par conséquent, i et j restent dans leur cluster respectif. Ainsi, par définition du système émulé, $E^{f^{(1)}(\gamma')} = E^{f^{(1)}(\gamma)}$.

Par conséquent, $f^{(1)}(\gamma') = f^{(1)}(\gamma)$. □

Lemme 63. *Si γ' se déduit de γ par la suppression d'un lien de communication (i, j) tel que $Token_x.gateway[y] = (i, j)$ et $Token_y.gateway[x] = (j, i)$, alors $f^{(1)}(\gamma')$ se déduit de $f^{(1)}(\gamma)$ par la perte du lien de communication (x, y).*

Démonstration. Par définition du système émulé, puisque le lien $Token_x.gateway[y] \notin E^{\gamma'}$, les nœuds x et y ne sont plus voisins dans $f^{(1)}(\gamma')$. □

En considérant la disparition d'un nœud comme la disparition de toutes les arêtes le reliant au reste du système, on peut démontrer que tout changement topologique laisse le système émulé inchangé, ou induit sur celui-ci un changement topologique.

Proposition 64. *Si γ' dérive de γ par un changement topologique, alors $f^{(1)}(\gamma) = f^{(1)}(\gamma')$ ou $f^{(1)}(\gamma')$ dérive de $f^{(1)}(\gamma)$ par un changement topologique sur le système virtuel.*

Démonstration. D'après les lemmes 61 à 63, et en considérant la disparition d'un nœud comme la disparition de tous les liens le reliant au réseau. □

Les propositions 60 et 64 nous permettent de conclure que toute exécution induit des modifications sur le système émulé qui forment une exécution de l'algorithme $\mathcal{A}$.

Théorème 65. *Toute exécution de notre algorithme sur le système distribué induit une exécution de l'algorithme $\mathcal{A}$ sur le système distribué émulé.*

Les changements topologiques d'un système distribué sont une cause d'instabilité. En particulier dans notre algorithme de clustering, un changement topologique peut détruire un cluster. Nous montrons donc qu'en l'absence de changement topologique dans le système distribué initial, le système émulé ne subira qu'un nombre fini de changements topologiques.

Proposition 66. *Pour toute exécution ayant un nombre fini de changements topologiques, l'exécution induite sur le système émulé comporte un nombre fini de changements topologiques.*

Démonstration. En un temps fini après le dernier changement topologique, le clustering cesse d'évoluer d'après la proposition 28. Ensuite, le graphe de communication finit par correspondre au clustering, d'après le corollaire 49 de la proposition 48. Le graphe de communication du système émulé ne change plus, et l'exécution induite ne peut comporter qu'un nombre fini de changements topologiques. □

4.6.4 Conformité à la spécification

Les sections précédentes nous permettent de montrer que toute exécution de notre algorithme distribué vérifie la spécification présentée à la section 4.4.

Théorème 67. *Toute exécution de l'algorithme du chapitre 3 vérifie la spécification présentée à la section 4.4.*

Démonstration. Soit une exécution $\mathcal{E} = (\gamma_1, \gamma_2, \dots)$ contenant un nombre fini de changements topologiques. Cette exécution vérifie bien tout les points de la spécification :

1. Correspondance au clustering :
   - il existe un entier i tel que le graphe de communication dans $f^{(1)}(\gamma_i)$ soit le graphe des clusters $G^{(1)}$ d'après la proposition 44.
2. Exécution induite :
   - Pour tout i,
     - soit $f^{(1)}(\gamma_i) = f^{(1)}(\gamma_{i+1})$;
     - soit $f^{(1)}(\gamma_i) \vdash^*_{\mathcal{A}} f^{(1)}(\gamma_{i+1})$;
     - soit il existe deux configurations $\gamma_1, \gamma_2 \in \mathcal{C}^{(1)}, f^{(1)}(\gamma_i) \vdash^*_{\mathcal{A}} \gamma_1, \gamma_2 \vdash^*_{\mathcal{A}} f^{(1)}(\gamma_{i+1})$ et γ_1 résulte de γ_2 par un changement topologique.

     Ceci est vrai d'après le théorème 65.
   - Il y a un nombre fini de changements topologiques dans cette exécution d'après la proposition 66.
3. Conformité au modèle :

- Si $m \in \mathcal{M}^{\gamma_j(1)}$, il existe $i > j$ tel que entre $\mathcal{M}^{\gamma_i(1)}$ et $\mathcal{M}^{\gamma_{i+1}(1)}$, la multiplicité de m soit réduite de 1, et $f^{(1)}(\gamma_i) \vdash^*_{\mathcal{A}} f^{(1)}(\gamma_{i+1})$ avec une transition correspondant à la réception du message m d'après la proposition 52.
- si deux clusters x et y ne sont plus voisins dans une configuration I, ils finissent par exécuter la procédure *À la perte de la connexion vers* d'après la proposition 50.
- si un nœud virtuel x est endormi dans $f^{(1)}(\gamma_i)$, il existe $j > i$ tel que x soit éveillé dans $f^{(1)}(\gamma_j)$, ou qu'il ait disparu, d'après la proposition 44.

□

Nous avons donc montré que toute exécution de l'algorithme du chapitre 3 vérifie la spécification présentée dans la section 4.4, induisant une exécution légale de l'algorithme $\mathcal{A}$ sur le système distribué composé des clusters.

4.7 Clustering hiérarchique

Dans cette section, on démontre le bon fonctionnement de l'algorithme de clustering hiérarchique présenté à la section 3.4.

Lorsqu'un système distribué exécute l'algorithme du chapitre 3, un clustering de celui-ci est calculé. De plus, ce clustering est un nouveau système distribué. Si ce nouveau système exécute lui aussi l'algorithme distribué du chapitre 3, un clustering de celui-ci est également calculé, formant des clusters de clusters. En répétant cette démarche, on obtient un clustering hiérarchique imbriqué du système.

Cette démonstration est composée d'une récurrence montrant que ce processus fonctionne correctement, et créé bien un nombre fini de niveaux hiérarchiques.

Lemme 68. *Soit un système distribué vérifiant les hypothèses de notre modèle et exécutant l'algorithme du chapitre 3. Toute exécution comporte un suffixe infini de configuration vérifiant la spécification présenté à la section 2.1.3, dans lequel le clustering est constant.*

Démonstration. Ce lemme est une simple reformulation du théorème 40. □

Lemme 69. *Soit un système distribué vérifiant les hypothèses de notre modèle, possédant au moins deux nœuds, et exécutant l'algorithme du chapitre 3. Le système extrait est de taille strictement inférieure au système de départ, et de taille au moins 1.*

Démonstration. Au bout d'un temps fini après le dernier changement topologique, le système distribué vérifie la spécification de l'algorithme du chapitre 2 d'après le lemme 68. En particulier, les clusters calculés sur ce système sont tous de taille au moins 2. Par conséquent, le nombre de clusters est forcément strictement inférieur au nombre de nœuds du système, c'est-à-dire $|V| > |V^{(1)}|$. De plus, cet algorithme calcule au moins un cluster, et $|V^{(1)}| \geq 1$. □

Lemme 70. *Soit un système distribué comportant au moins deux noeuds, vérifiant les hypothèses de notre modèle, et exécutant l'algorithme du chapitre 3. Le système extrait est un système distribué, vérifiant les hypothèses de notre modèle, et exécutant l'algorithme du chapitre 3.*

Démonstration. Reformulation du théorème 67 indiquant que le système émulé vérifie la spécification 4.4. □

Théorème 71. *L'algorithme de clustering présenté au chapitre 3 construit bien un clustering hiérarchique imbriqué, et un arbre couvrant modulaire du réseau.*

Démonstration. Les deux points à vérifier sont :

- le processus construit bien un nombre fini de niveaux ;
- dans chaque niveau, le clustering est bien construit, c'est-à-dire le système associé vérifie la spécification présentée à la section 2.1.3.

Montrons ces points par récurrence sur les niveaux hiérarchiques. Premièrement, le système de départ se comporte conformément à notre modèle par hypothèse, et il exécute l'algorithme présenté dans le chapitre 3. Ce système finit donc par vérifier la spécification présentée à la section 2.1.3 par le lemme 68. Par application du lemme 70 puis du lemme 68, toute exécution de notre algorithme induit une exécution de notre algorithme sur le niveau hiérarchique supérieur. Ainsi, dans chaque niveau hiérarchique, le système finit donc par vérifier la spécification présentée à la section 2.1.3.

Le lemme 69 nous permet quand à lui de montrer que le nombre de niveaux est fini. □

4.8 Conclusion

Dans ce chapitre, nous avons introduit un cadre formel nous permettant de démontrer le bon fonctionnement de l'algorithme de clustering hiérarchique présenté à la section 3.4.

Cet algorithme s'appuie sur le fait que nos clusters émulent les nœuds virtuels d'un nouveau système distribué, exécutant l'algorithme distribué de notre choix.

De notre point de vue, un système distribué est donc la donnée d'un graphe de communication et d'un algorithme distribué. Le graphe d'adjacence des clusters peut ainsi être associé à un algorithme distribué $\mathcal{A}$ pour former un nouveau système distribué.

Nous avons ainsi introduit une spécification traduisant, en terme d'exécution, le fait qu'un clustering émule un système distribué. Pour cela, un nouveau système distribué est défini de manière formelle à partir du clustering. Les informations concernant ce système distribué étant présentes dans l'algorithme d'émulation, les configurations du système émulé peuvent être extraites des configurations du système initial.

Cette fonction d'extraction transforme ainsi toute exécution sur le système en une exécution sur le système émulé : les configurations extraites au cours d'une exécution sont liées entre elles par des séquences de transitions (et éventuellement des changements topologiques).

Ainsi, lorsqu'une modification survient dans le système émulé, elle correspond exactement à une séquence de transitions de l'algorithme du système émulé, accompagnée ou non de changements topologiques autorisés par le modèle. De plus, pour toute exécution de notre algorithme sur le système initial, les propriétés du modèle sont vraies dans l'exécution induite sur le système émulé : tous les messages émis finissent par être reçus, et tous les changements topologiques finissent par être détectés. Le système émulé est donc un système distribué conforme à notre modèle.

Par la suite, nous utilisons cette propriété pour démontrer le bon fonctionnement de l'algorithme de clustering hiérarchique présenté à la section 3.4. En effet, chaque clustering, dans chaque niveau logique, est un nouveau système distribué exécutant notre algorithme. Par conséquent, dans chaque niveau logique, le système finit par vérifier la spécification du clustering présentée à la section 2.1.3. Il en résulte le calcul d'un clustering hiérarchique imbriqué du système.

4.9 Annexes

4.9.1 Évolution des clusters cohérents

Clusters cohérents : communications sortantes

Lemme 72. *Si C_x^γ est un cluster cohérent et si $\gamma \vdash \gamma'$ est déclenchée par la réception d'un message Recruit[x] par un nœud n'appartenant pas à C_x^γ, alors $C_x^{\gamma'}$ reste cohérent.*

Démonstration. Soit C_x^γ un cluster cohérent. Soit i un nœud n'appartenant pas à C_x^γ recevant un message $Recruit[x]$ venant d'un nœud e dans K_x^γ. Les deux cas suivants sont possibles :

- Si i est dans un cluster ($cluster_i \neq (\perp, 0)$), alors aucune variable ne change (ligne 1 algorithme 5 fonction $JoinOrdinary$). Ainsi $\forall j \in V^{\gamma'} = V^\gamma, var_j^{\gamma'} = var_j^\gamma$. Par conséquent, $C_x^{\gamma'} = C_x^\gamma$, $K_x^{\gamma'} = K_x^\gamma$ et $\mathcal{A}_x^{\gamma'} = \mathcal{A}_x^\gamma$. De plus $\mathcal{M}^{\gamma'} = \mathcal{M}^\gamma \setminus \{(Recruit, x, e, i)\}$. Toutes les propriétés des clusters cohérents restent vraies et $C_x^{\gamma'}$ reste donc cohérent.
- Si i n'est pas dans un cluster, il rejoint $C_x^{\gamma'}$ sans entrer dans $K_x^{\gamma'}$ (ligne 2, 3 et 4 de la fonction $JoinOrdinary$). Le nœud i envoie un message $ACK[x]$ au nœud e, $cluster_i \leftarrow x$ et $father_i$ devient e. D'où $K_x^{\gamma'} = K_x^\gamma$, $C_x^{\gamma'} = C_x^\gamma \cup \{i\}$. Puisque $(Recruit, x, e, i) \in \mathcal{M}^\gamma$, par le premier point de la définition des clusters cohérents, $e \in K_x^\gamma = K_x^{\gamma'}$. Ainsi $i \in C_x^{\gamma'}$ possède un voisin dans $K_x^{\gamma'}$. De plus, K_x^γ est un ensemble dominant connexe de C_x^γ par définition des clusters cohérents. Par conséquent :
  - $|K_x^{\gamma'}| = |K_x^\gamma|$
  - $K_x^{\gamma'}$ est un ensemble dominant connexe de $C_x^{\gamma'}$

  De plus, $\begin{array}{ll} \mathcal{M}^{\gamma'} = \mathcal{M}^\gamma \setminus \{(Recruit, x, e, i)\} & \cup\{(ACK, x, i, e)\} \\ & \cup \left\{(cluster, x, i, m)/(i, m) \in E^{\gamma'}\right\} \end{array}$. Puisque $e \in K_x$, les points 1 à 5 de la définition des clusters cohérents restent donc vrais.
  $father_i \leftarrow e$ et $e \in K_x^\gamma$ donc e est dans $\mathcal{A}_x^\gamma$. Puisque $i \notin C_x^\gamma$, la seule possibilité pour que le nœud $m \in C_x^\gamma$ avec $father_m = i$ est que $(Token_x, param, m, i) \in \mathcal{M}^\gamma$, et dans ce cas $(m, i) \notin \mathcal{R}_x^\gamma$ par définition de $\mathcal{A}_x^\gamma$. Le nœud i est donc ajouté à $\mathcal{A}_x^{\gamma'}$ en tant que feuille, et $\mathcal{A}_x^{\gamma'}$ reste connexe sans cycle.
  Ainsi $C_x^{\gamma'}$ reste cohérent.

□

Lemme 73. *Soit un nœud i non clusterisé, ou ordinaire et n'appartenant pas à C_x^γ. Si C_x^γ est cohérent et si i reçoit un jeton Token avec $Token.id = x$, alors $C_x^{\gamma'}$ reste cohérent.*

Démonstration.

- Si $Token_x.size \geq K$, le nœud i n'est pas recruté (ligne 38 à 40 de la fonction $Token$). Le jeton $Token_x$ est renvoyé à son expéditeur e. Ainsi, aucune variable n'est modifiée, d'où $C_x^{\gamma'} = C_x^\gamma$, $K_x^{\gamma'} = K_x^\gamma$ et $\mathcal{M}^{\gamma'} = \mathcal{M}^\gamma \setminus \{(Token_x, param, e, i)\} \cup \{(Token_x, param, i, e)\}$. La structure du cluster reste inchangée, les propriétés sur les messages restent vraies, $\mathcal{A}_x^{\gamma'} = \mathcal{A}_x^\gamma$ et $Token_x.tree$ est inchangé.
  Ainsi $C_x^{\gamma'}$ reste cohérent.
- Si ($Token_x.size < K$), i est recruté en tant que nœud de cœur, c'est-à-dire qu'il entre dans $K_x^{\gamma'}$ (ligne 28 à 33 de la fonction $Token$). Ensuite i envoie le message $Token_x$ à un voisin choisi uniformément au hasard, j, et envoie un message $Recruit[x]$ à tous ses voisins. Par

conséquent $K_x^{\gamma'} = K_x^{\gamma} \cup \{i\}$, $\forall var, \forall k \neq i, var_k^{\gamma'} = var_k^{\gamma}$, $C_x^{\gamma'} = C_x^{\gamma} \cup \{i\}$ et

$$\mathcal{M}^{\gamma'} = \mathcal{M}^{\gamma} \setminus \{(Token_x, param, e, i)\} \begin{array}{l} \cup (Token_x, param, i, j)\} \\ \cup \{(Recruit, x, i, m)/(i, m) \in E^{\gamma'}\} \\ \cup \{(Cluster, x, i, m)/(i, m) \in E^{\gamma'}\}. \end{array}$$

- $|K_x^{\gamma'}| = |K_x^{\gamma}| + 1 \leq K$.
- $K_x^{\gamma'}$ reste un ensemble dominant connexe de $C_x^{\gamma'}$.

Les propriétés sur la structure et sur les messages des clusters cohérents restent vraies.
De plus, $(Token_x, param, e, i) \in \mathcal{M}^{\gamma}$ *et* $e \in K_x$ donc e est la racine de l'arbre $\mathcal{A}_x^{\gamma}$ et $father_e^{\gamma} = i$. Le nœud i devient la racine de $\mathcal{A}_x^{\gamma'}$ et $father_i^{\gamma'} = j$. Puisque $Token_x.tree[e] = i$, $tree$ reste cohérent avec $\mathcal{A}_x^{\gamma'}$, et $Token_x.tree[i] = i$. Les propriétés sur l'arbre couvrant restent donc vraies.
Par conséquent, $C_x^{\gamma'}$ reste cohérent.

□

Lemme 74. *Soit un nœud de cœur i dans K_y^{γ}, avec $y < x$ par l'ordre lexicographique sur $\mathbb{N}^2$, et tel que C_y soit complet. Si C_x^{γ} est cohérent et si i reçoit le jeton $Token_x$, alors $C_x^{\gamma'}$ reste cohérent.*

Démonstration. Dans ce cas, le nœud i renvoie le message $Token_x$ à son expéditeur (ligne 39 de la fonction $Token$). Ainsi $K_x^{\gamma'} = K_x^{\gamma}$, $C_x^{\gamma'} = C_x^{\gamma}$. La structure du cluster est inchangée. $\mathcal{M}^{\gamma'} = \mathcal{M}^{\gamma} \setminus \{(Token_x, param, e, i)\} \cup \{(Token_x, param, i, e)\}$. Les propriétés sur les messages dans les clusters cohérents restent donc vérifiées.

$\mathcal{A}_x^{\gamma'} = \mathcal{A}_x^{\gamma}$ and $Token_x.tree$ est inchangé.

Par conséquent $C_x^{\gamma'}$ reste cohérent. □

Lemme 75. *Soit un nœud de cœur i dans K_y^{γ}, avec $y < x$ en considérant l'ordre lexicographique. Si C_x^{γ} est cohérent et si i reçoit le jeton $Token_x$, alors $C_x^{\gamma'}$ passe en destruction.*

Démonstration. Dans ce cas, la fonction $Token$ est exécutée, ligne 34 à 37. Aucune variable n'est modifiée, d'où $C_x^{\gamma'} = C_x^{\gamma}$ and $K_x^{\gamma'} = K_x^{\gamma}$. $\mathcal{M}^{\gamma'} = \mathcal{M}^{\gamma} \setminus \{(Token_x, param, e, i)\} \cup \{(Delete, x, i, e)\} \cup \{(Recruit, y, i, e)\}$. Par conséquent, il n'y a plus de jeton $Token_x$ dans le réseau.

Par définition d'un cluster cohérent, tout les nœuds ordinaires de $C_x^{\gamma'} = C_x^{\gamma}$ ont donc pour père un nœud de $K_x^{\gamma'}$, et tous les messages $Recruit[x]$ ont été émis par un nœud dans $K_x^{\gamma'}$

$\mathcal{F}_x^{\gamma'} = \mathcal{A}_x^{\gamma}$ est une forêt couvrante de $C_x^{\gamma'}$ constituée d'un seul arbre.

Puisque $e \in K_x^{\gamma}$ et $(Token_x, param, e, i) \in \mathcal{M}^{\gamma}$, e est la racine de $\mathcal{A}_x^{\gamma}$. Le nœud e est ainsi le seul nœud de $C_x^{\gamma'}$ tel que $father_e = i$ est un nœud n'appartenant pas à $C_x^{\gamma} = C_x^{\gamma'}$, et $(Delete, x, i, e) \in \mathcal{M}^{\gamma'}$.

Ainsi $C_x^{\gamma'}$ devient un cluster en destruction. □

Ces lemmes permettent de démontrer la proposition 20.

Proposition (Rappel de la proposition 20)**.** *Si C_x^{γ} est un cluster cohérent et si $\gamma \vdash \gamma'$ est déclenchée par la réception, sur un nœud n'appartenant pas à C_x^{γ}, d'un message venant de C_x^{γ}, alors $C_x^{\gamma'}$ reste cohérent ou passe en destruction.*

Démonstration. (Preuve de la proposition 20). D'après les lemmes 72, 73, 74, 75, et puisque par définition des clusters cohérents, aucun message $ACK[x]$ ne peut provenir de l'extérieur du cluster, et il n'y a pas de message $Delete[x]$ circulant dans le réseau. □

Clusters cohérent : communications internes

Lemme 76. *Si C_x^γ est cohérent et si un noeud de C_x^γ reçoit un message $Recruit[x]$, alors $C_x^{\gamma'}$ reste cohérent.*

Démonstration. Soit i un nœud de C_x^γ. Si i reçoit un message $Recruit[x]$ venant d'un nœud e, e est dans K_x^γ par définition des clusters cohérents. Le nœud i exécute la fonction *JoinOrdinary*.

Puisque i est dans C_x^γ, aucune variable ne change (ligne 1, fonction *JoinOrdinary*). Ainsi $C_x^{\gamma'} = C_x^\gamma$, $K_x^{\gamma'} = K_x^\gamma$, $\mathcal{A}_x^{\gamma'} = \mathcal{A}_x^\gamma$ et $Token_x.tree^{\gamma'} = Token_x.tree^\gamma$.

Par conséquent $C_x^{\gamma'}$ reste cohérent. □

Lemme 77. *Si C_x^γ est cohérent et si un nœud ordinaire i dans C_x^γ reçoit le jeton $Token_x$, alors $C_x^{\gamma'}$ reste cohérent.*

Démonstration.

- Si $Token_x.size \geq K$, (ligne 38 à 40, fonction $Token$), $Token_x.tree[i] \neq i$: i renvoie le jeton $Token$ à l'expéditeur. $C_x^{\gamma'} = C_x^\gamma$, $K_x^{\gamma'} = K_x^\gamma$, $\mathcal{A}_x^{\gamma'} = \mathcal{A}_x^\gamma$ et $\mathcal{M}^{\gamma'} = \mathcal{M}^\gamma \setminus \{(Token_x, param, e, i)\} \cup \{(Token_x, param, i, e)\}$
  Ainsi, $C_x^{\gamma'}$ reste cohérent.
- Si $Token_x.size < K$, i devient un nœud de cœur (ligne 28 à 33 de la fonction $Token$), envoie un message $Recruit[x]$ à chacun de ses voisins et envoie le jeton $Token_x$ à un voisin j choisi uniformément au hasard. Ainsi, $K_x^{\gamma'} = K_x^\gamma \cup \{i\}$, $C_x^{\gamma'} = C_x^\gamma$, $|K_x^{\gamma'}| = |K_x^\gamma| + 1 \leq K$. Par conséquent $K_x^{\gamma'}$ reste un ensemble dominant connexe de $C_x^{\gamma'}$.
  $$\begin{aligned}\mathcal{M}^{\gamma'} = \mathcal{M}^\gamma \setminus \{(Token_x, param, e, i)\} \quad & \cup\{(Token_x, param, i, e)\} \\ & \cup\{(Recruit, x, i, m)/(m, i) \in E^{\gamma'}\} \\ & \cup\{(cluster, x, i, m)/(m, i) \in E^{\gamma'}\}.\end{aligned}$$
  Toutes les propriétés sur les messages dans un cluster cohérent restent donc vraies, puisque $i \in K_x^{\gamma'}$. $i \in C_x^\gamma \setminus K_x^\gamma$ donc i est une feuille de $\mathcal{A}_x^\gamma$ (propriété 4 de l'arbre couvrant, définition d'un cluster cohérent). i est retiré de l'arbre couvrant et ajouté en tant que racine de $\mathcal{A}_x^{\gamma'}$ puisque $father_e = i$ et e était la racine de $\mathcal{A}_x^\gamma$ par la propriété 4 de l'arbre couvrant dans la définition d'un cluster cohérent.
  Par conséquent $C_x^{\gamma'}$ reste cohérent.

□

Lemme 78. *Si C_x^γ est cohérent et si un nœud i dans K_x^γ reçoit le jeton $Token_x$ en provenance d'un nœud e, $C_x^{\gamma'}$ reste cohérent.*

Démonstration. Dans ce cas, le nœud i exécute les ligne 1 à 17 de la fonction $Token$. Les variables du jeton $Token_x$ sont mises à jour avec les informations locales, puis i envoie un message $Recruit[x]$ et un message $Cluster[x]$ à chacun de ses voisins, et envoie le jeton $Token_x$ à un voisin j choisi uniformément au hasard. Ainsi, $C_x^{\gamma'} = C_x^\gamma$, $K_x^{\gamma'} = K_x^\gamma$. La structure du cluster est inchangée.

$$\begin{aligned}\mathcal{M}^{\gamma'} = \mathcal{M}^\gamma \setminus \{(Token_x, param, e, i)\} \quad & \cup\{(Token_x, param, i, j)\} \\ & \cup\{(Recruit, x, i, m)/(i, m) \in E\} \\ & \cup\{(cluster, x, i, m)/(i, m) \in E\}.\end{aligned}$$

Les propriétés sur les messages dans la définition d'un cluster cohérent restent donc vraies, puisque $i \in K_x^{\gamma'}$.

Nous vérifions maintenant que $\mathcal{A}_x^{\gamma'}$ reste un arbre couvrant vérifiant les propriétés de la définition de cluster cohérent (définition 13). Le lien $i - father_i^\gamma$ est enlevé de $\mathcal{A}_x^\gamma$ et le lien $e - i$ est ajouté à $\mathcal{A}_x^{\gamma'}$.

- connexe : On montre qu'il existe un chemin reliant chaque nœud à la racine i. Puisque $e \in K_x^\gamma$ et $(Token_x, param, e, i) \in \mathcal{M}^\gamma$, e était la racine de $\mathcal{A}_x^\gamma$. Puisque A_x^γ était un arbre couvrant de C_x^γ, il est connexe. Ainsi, pour chaque nœud p, il existe un chemin depuis p jusqu'à e dans $\mathcal{A}_x^\gamma$. A partir de ce chemin, nous construisons un chemin reliant le nœud p à la nouvelle racine i de $\mathcal{A}_x^{\gamma'}$:
  - Si le chemin n'emprunte pas le lien $(i, father_i^\gamma)$, qui est le seul lien enlevé de $\mathcal{A}_x^\gamma$ dans $\mathcal{A}_x^{\gamma'}$, alors ce chemin est entièrement dans $\mathcal{A}_x^{\gamma'}$ et puisque i est le père de e dans $\mathcal{A}_x^{\gamma'}$, p et i sont reliés ;
  - Si le chemin passe par le lien $(i, father_i^\gamma)$, alors p est relié à i dans $\mathcal{A}_x^\gamma$ et dans $\mathcal{A}_x^{\gamma'}$ puisque ces liens ne changent pas.

  Ainsi, dans tous les cas, chaque nœud p est relié à i, la racine de $\mathcal{A}_x^{\gamma'}$
- acyclique : Nous supposons que $\mathcal{A}_x^{\gamma'}$ possède un cycle : il existe p et q dans $C_x^{\gamma'}$ reliés par deux chemins P_1 et P_2 sans aucun nœud commun.
  - Si aucun de ces chemins ne passe par le lien (e, i), alors les deux chemins étaient dans $\mathcal{A}_x^\gamma$, ce qui est impossible puisque $\mathcal{A}_x^\gamma$ est acyclique ;
  - Si un des chemin passe par (e, i), puisque i est le père de e dans $\mathcal{A}_x^{\gamma'}$, il existe un nœud $k \neq e$, fils de i, tel que e et k ont un descendant commun ; puisque $e - i$ est le seul lien dans $\mathcal{A}_x^{\gamma'}$ qui n'est pas dans $\mathcal{A}_x^\gamma$, e et k ont un descendant commun dans $\mathcal{A}_x^\gamma$ ce qui n'est pas possible.

  Ainsi, dans tous les cas, $\mathcal{A}_x^{\gamma'}$ est acyclique.

Par conséquent $C_x^{\gamma'}$ reste cohérent. □

Lemme 79. *Si C_x^γ est cohérent et si un nœud i dans C_x^γ reçoit un message $ACK[x]$, $C_x^{\gamma'}$ reste cohérent.*

Démonstration. Par définition des clusters cohérents, si un message $ACK[x]$ est adressé à un nœud i, ce nœud est un nœud de cœur $i \in K_x$. Par conséquent, le message est simplement ignoré (algorithme 9), et C_x reste un cluster cohérent. □

Ces lemmes nous permettent de démontrer la proposition 21.

Proposition (Rappel de la propriété 21)**.** *Si C_x^γ est un cluster cohérent et si $\gamma \vdash \gamma'$ est déclenchée par la réception, sur un nœud de C_x^γ, d'un message émis par un nœud également dans C_x^γ, alors $C_x^{\gamma'}$ reste cohérent.*

Démonstration. (Preuve de la proposition 21)
D'après les lemmes 76, 77, 78, 79 et puisqu'il n'y a aucun message $Delete[x]$ en circulation dans le réseau, vu que C_x est un cluster cohérent. □

Clusters cohérent : communication en provenance de l'extérieur

Proposition (Rappel de la proposition 22)**.** *Si C_x^γ est un cluster cohérent et si $\gamma \vdash \gamma'$ est déclenchée par la réception, sur un nœud du cluster C_x^γ, d'un message venant d'un nœud n'appartenant pas au cluster C_x^γ, alors $C_x^{\gamma'}$ reste cohérent.*

Démonstration. (Preuve de la proposition 22).
- Si un nœud i dans C_x^γ reçoit un message $Delete[y]$ ou un message $Recruit[y]$ en provenance d'un nœud e dans C_y, alors i l'ignore. $C_x^{\gamma'} = C_x^\gamma$, $K_x^{\gamma'} = K_x^\gamma$, $\mathcal{A}_x^{\gamma'} = \mathcal{A}_x^\gamma$ et $\mathcal{M}^{\gamma'} = \mathcal{M}^\gamma \setminus \{(Recruit, y, e, i)\}$ ou $\mathcal{M}^{\gamma'} = \mathcal{M}^\gamma \setminus \{(Delete, y, e, i)\}$.

Les propriétés sur les messages de la définition des clusters cohérents restent vraies, et $C_x^{\gamma'}$ reste cohérent.

- Si un nœud i dans K_x^γ reçoit le jeton $Token_y$ d'un cluster C_y^γ, alors il exécute l'algorithme 10. Puisque i est dans $K_x^{\gamma'}$, i envoie le jeton $Token_y$ à l'expéditeur, ou il lui envoie un message $Delete[y]$. Dans les deux cas, $C_x^{\gamma'} = C_x^\gamma$, $K_x^{\gamma'} = K_x^\gamma$, $\mathcal{A}_x^{\gamma'} = \mathcal{A}_x^\gamma$ et $\mathcal{M}^{\gamma'} = \mathcal{M}^\gamma \setminus \{(Token_y, param, e, i)\} \cup \{(Token_y, param, i, e)\}$ ou $\mathcal{M}^{\gamma'} = \mathcal{M}^\gamma \setminus \{(Token_y, param, e, i)\} \cup \{(Delete, y, i, e)\}$
$\cup \{(Recruit, x, i, e)\}$.
Dans ces cas, $C_x^{\gamma'}$ reste cohérent.
- Si $i \in C_x^\gamma$ est un nœud ordinaire et si C_y^γ est complet, i envoie le jeton $Token_y$ à l'expéditeur. $C_x^{\gamma'} = C_x^\gamma$, $K_x^{\gamma'} = K_x^{\gamma'}$, $\mathcal{A}_x^{\gamma'} = \mathcal{A}_x^\gamma$ et $\mathcal{M}^{\gamma'} = \mathcal{M}^\gamma \setminus \{(Token_y, param, e, i)\} \cup \{(Token_y, param, i, e)\}$. Ainsi $C_x^{\gamma'}$ reste cohérent.
- Si $i \in C_x^\gamma$ est un nœud ordinaire et si C_y^γ n'est pas complet, alors i quitte le cluster C_x et entre dans K_y. Puisque $i \in C_x^\gamma \setminus K_x^\gamma$, i est une feuille de l'arbre couvrant $\mathcal{A}_x^\gamma$. $C_x^{\gamma'} = C_x^\gamma \setminus \{i\}$, $K_x^{\gamma'} = K_x^\gamma$, et $\mathcal{M}^{\gamma'} = \mathcal{M}^\gamma \setminus \{(Token_y, param, e, i)\} \cup \{(Token_y, param, i, j)\} \cup \{(Recruit, y, i, m) \setminus (i, m) \in E\}$. Les propriétés sur la structure et sur les messages de C_x, dans la définition d'un cluster cohérent, restent vraies.
i était une feuille de $\mathcal{A}_x^\gamma$ et il n'est pas $\mathcal{A}_x^{\gamma'}$ qui reste donc un arbre couvrant C_x vérifiant les propriétés de la définition des clusters cohérents.
Par conséquent, $C_x^{\gamma'}$ reste cohérent.

□

4.9.2 Clusters en destruction

Clusters en destruction : communication vers l'extérieur

Lemme 80. *Si C_x^γ est en destruction et si un nœud $i \notin C_x^\gamma$ reçoit un message $Recruit[x]$ en provenance d'un nœud $e \in K_x$, $C_x^{\gamma'}$ reste en destruction.*

Démonstration. Le nœud i exécute la fonction $JoinOrdinary$. Soit le message est ignoré, et $C_x^{\gamma'}$ reste un cluster en destruction. Soit le nœud i rejoint le cluster $C_x^{\gamma'}$ en tant que nœud ordinaire et envoie des messages $Cluster[x]$ à tous ses voisins et un message $ACK[x]$ à son père e. $C_x^{\gamma'} = C_x^\gamma \cup \{i\}$, $K_x^{\gamma'} = K_x^\gamma$,

$$\mathcal{M}^{\gamma'} = \mathcal{M}^\gamma \setminus \{(Recruit, x, e, i)\} \quad \cup \{(Cluster, x, i, m)/(i, m) \in E^{\gamma'}\} \cup \{(ACK, x, i, e\}$$

et $\mathcal{F}_x^{\gamma'} = \mathcal{F}_x^\gamma \cup \{i\}$ ajouté en tant que feuille. Par conséquent, $\mathcal{F}_x^{\gamma'}$ reste une forêt couvrante valide de $C_x^{\gamma'}$, qui reste donc un cluster en destruction. □

Remarque 81. *Le cas où l'émetteur e n'est plus dans C_x est traité à la propriété 18.*

Lemme 82. *Si C_x^γ est en destruction, et si un nœud $i \notin C_x^\gamma$ reçoit un message $Delete[x]$ en provenance d'un nœud e, alors $C_x^{\gamma'}$ reste en destruction.*

Démonstration. Si un nœud à l'extérieur de C_x reçoit un message $Delete[x]$, il l'ignore (première ligne de la fonction *leave*). $C_x^{\gamma'} = C_x^\gamma$, $K_x^{\gamma'} = k_x^\gamma\ \gamma'$, $\mathcal{M}^{\gamma'} = \mathcal{M}^\gamma \setminus \{(Delete, x, e, i)\}$ et $\mathcal{F}_x^{\gamma'} = \mathcal{F}_x^\gamma$. Ainsi $C_x^{\gamma'}$ reste un cluster en destruction. □

Lemme 83. *Si C_x^γ est en destruction et si un nœud i extérieur à C_x^γ reçoit un message $ACK[x]$ en provenance d'un nœud $e \in C_x$, $C_x^{\gamma'}$ reste en destruction.*

Démonstration. Lors de la réception de ce message, le nœud i exécute l'algorithme 9. Puisque le nœud $i \notin C_x$, il envoie un message $Delete[x]$ à l'expéditeur. Toutes les propriétés sur les messages dans un cluster en destruction restent donc vraies. □

Proposition 84 (Rappel de la proposition 24)**.** *Si C_x^γ est en destruction et si $\gamma \vdash \gamma'$ est déclenchée par la réception, sur un nœud externe à C_x^γ, d'un message émis par un nœud de C_x^γ, alors $C_x^{\gamma'}$ reste en destruction.*

Démonstration. (Preuve de la proposition 24)
Par application des lemmes 80, 82 et 83, et puisqu'il n'y a aucun jeton $Token$ en circulation associé à un cluster en destruction (def 14). □

Clusters en destruction : communications internes

Lemme 85. *Si C_x^γ est en destruction et si un nœud i dans C_x^γ reçoit un message $ACK[x]$ en provenance d'un nœud $e \in C_x^\gamma$, alors $C_x^{\gamma'}$ reste en destruction.*

Démonstration. Par définition des clusters en destruction, puisque le nœud i est le destinataire d'un message $ACK[x]$, c'est un nœud de K_x. Par conséquent, le message est ignoré (algorithme 9), et C_x reste en destruction. □

Lemme 86. *Si C_x^γ est en destruction et si un nœud i dans C_x^γ reçoit un message $Recruit[x]$ en provenance d'un nœud $e \in C_x^\gamma$, alors $C_x^{\gamma'}$ reste en destruction.*

Démonstration. Dans ce cas, comme pour le lemme 76 (Si de la ligne 1, fonction $JoinOrdinary$), le nœud i ignore ce message. $C_x^{\gamma'} = C_x^\gamma$, $K_x^{\gamma'} = K_x^\gamma$, $\mathcal{F}_x^{\gamma'} = \mathcal{F}_x^\gamma$ et $\mathcal{M}^{\gamma'} = \mathcal{M}^\gamma \setminus \{(Recruit, x, e, i)\}$
Par conséquent, $C_x^{\gamma'}$ reste en destruction. □

Lemme 87. *Si C_x^γ est en destruction et si un nœud i de C_x^γ reçoit un message $Delete[x]$ venant d'un nœud e, alors $C_x^{\gamma'}$ reste en destruction.*

Démonstration.

- Si e n'est pas le père de i dans l'arbre couvrant du cluster C_x ($father_i \neq e$, ligne 1 de la fonction $Leave$), alors le message $Delete[x]$ est ignoré, $C_x^{\gamma'} = C_x^\gamma$, $\mathcal{F}_x^{\gamma'} = \mathcal{F}_x^\gamma$ et $\mathcal{M}^{\gamma'} = \mathcal{M}^\gamma \setminus \{(Delete, x, e, i)\}$. Dans ce cas, $C_x^{\gamma'}$ reste en destruction.
- Si e est le père de i dans l'arbre couvrant, i quitte le cluster et envoie un message $Delete[x]$ à tous ses voisins (fonction $Leave$). Ainsi, tous les nœuds ordinaires de C_x ayant i pour père se voient adresser un message $Delete[x]$ en provenance de leur père. De plus, i était la racine d'un arbre de la forêt couvrante $\mathcal{F}_x^\gamma$. Par conséquent $\mathcal{F}_x^{\gamma'}$ reste une forêt couvrante de $C_x^{\gamma'}$. $\mathcal{M}^{\gamma'} = \mathcal{M}^\gamma \setminus \{(Delete, (x), e, i)\} \cup \{(Delete, (x), i, l)/(i, l) \in E\}$. Si m est un nœud tel que $father_m^{\gamma'} \notin C_x^{\gamma'}$, alors $father_m \notin C_x^\gamma$ ou $father_m = i$. Dans tous les cas, $\exists (Delete, (x), father_m, m) \in \mathcal{M}^{\gamma'}$
  Par conséquent $C_x^{\gamma'}$ reste en destruction.

□

Proposition (Rappel de la proposition 25)**.** *Si C_x^γ est en destruction et si $\gamma \vdash \gamma'$ est déclenchée par la réception, sur un nœud du cluster C_x^γ, d'un message émis par un nœud de C_x^γ, alors $C_x^{\gamma'}$ reste en destruction.*

Démonstration. (Preuve de la proposition 25)
Par application des lemmes 86, 87 et 85, et puisqu'il n'y a pas de jeton $Token$ en circulation associé à un message en destruction (def 14). □

Clusters en destruction : communication venant de l'extérieur

Lemme 88. *Si C_x^γ est en destruction et si un nœud i dans C_x^γ reçoit un message $ACK[y]$ tel que $x \neq y$ en provenance d'un nœud $e \notin C_x$, alors $C_x^{\gamma'}$ reste en destruction.*

Démonstration. Par définition des clusters en destruction, puisque i est le destinataire d'un message $ACK[x]$, il est dans K_x. Par conséquent, ce nœud ignore le message $ACK[x]$ (algorithme 9). C_x reste un cluster en destruction. □

Lemme 89. *Si C_x^γ est en destruction et si un nœud i dans C_x^γ reçoit un message $ACK[y]$ tel que $x \neq y$ en provenance d'un nœud $e \notin C_x$, alors $C_x^{\gamma'}$ reste en destruction.*

Démonstration. Lors de la réception de ce message, le nœud i exécute la procédure 9 et envoie un message $Delete[y]$ à l'expéditeur. L'identifiant de ce message n'étant pas x, le cluster C_x reste en destruction. □

Lemme 90. *Si C_x^γ est en destruction et si un nœud i dans C_x^γ reçoit un message $Recruit[y]$ ou un message $Delete[y]$ tel que $x \neq y$ en provenance d'un nœud e, alors $C_x^{\gamma'}$ reste en destruction.*

Démonstration. Dans les deux cas, un nœud $i \in C_x^\gamma$ qui reçoit un message $Recruit[y]$ ou un message $Delete[y]$ l'ignore (fonctions $JoinOrdinary$ et $Leave$). Ainsi, $C_x^{\gamma'} = C_x^\gamma$,$\mathcal{F}_x^{\gamma'} = \mathcal{F}_x^\gamma$ et $\mathcal{M}^{\gamma'} = \mathcal{M}^\gamma \setminus \{(Delete,(y),e,i)\}$ ou $\mathcal{M}^{\gamma'} = \mathcal{M}^\gamma \setminus \{(Recruit,(y),e,i)\}$.

Par conséquent C_x reste en destruction. □

Lemme 91. *Si C_x^γ est en destruction et si un nœud i membre de C_x^γ reçoit un jeton $Token$ venant d'un nœud e appartenant à un autre cluster, $C_x^{\gamma'}$ reste en destruction.*

Démonstration. Si un nœud i appartenant à C_x^γ reçoit un jeton $Token$ en provenance d'un autre cluster, il le renvoie à l'expéditeur ou envoie un message $Delete[y]$ suivis d'un message $Recruit[x]$ à l'expéditeur (fonction $Leave$). Ainsi $C_x^{\gamma'} = C_x^\gamma$, $\mathcal{F}_x^{\gamma'} = \mathcal{F}_x^\gamma$ et $\mathcal{M}^{\gamma'} = \mathcal{M}^\gamma \setminus \{(Token,(y),e,i)\} \cup \{(Token,(y),i,e)\}$ ou $\mathcal{M}^{\gamma'} = \mathcal{M}^\gamma \setminus \{(Token,(y),e,i)\} \cup \{(Delete,(y),i,e)\} \cup \{(Recruit,(x),i,e)\}$. Dans tous les cas, $C_x^{\gamma'}$ reste en destruction. □

Des lemmes 88, 89, 90 et 91, on déduit la proposition 26.

Proposition (Rappel de la proposition 26)**.** *Si C_x^γ est en destruction et si $\gamma \vdash \gamma'$ est déclenchée par la réception, sur un nœud de C_x^γ, d'un message émis par un nœud externe à C_x^γ, alors $C_x^{\gamma'}$ reste en destruction.*

Chapitre 5

Clustering pour l'ordonnancement des communications

5.1 Introduction

Le clustering est apparu dans un premier temps dans le contexte du routage ([BE81] et [McQ74]). Les clusters calculés sont constitués d'un leader qui recrute les machines qui l'entourent. Le routage au sein du réseau s'appuie ensuite sur ces leaders. Lorsqu'un nœud veut envoyer un message à une autre machine du réseau, il l'envoie au leader de son cluster. Si le destinataire se situe dans le même cluster, c'est un voisin du leader, qui peut lui transmettre le message. Les communications inter-clusters nécessitent quant à elles que les leaders maintiennent une table de routage entre eux. L'émetteur du message l'envoie à son leader qui calcule un chemin vers le leader du cluster de destination. Ce dernier peut alors remettre le message au nœud destinataire.

Le clustering est donc, dès son apparition, utilisé comme un outil permettant d'adapter à de grands réseaux des algorithmes fonctionnant localement. C'est l'usage que nous faisons ici du clustering pour résoudre un problème issu des réseaux de capteurs.

Un réseau de capteurs est constitué d'une collection d'appareils, capables de mesurer une grandeur physique de leur environnement, et capables de communiquer entre eux par une interface de communication sans fil. Lorsqu'un capteur envoie un message, tous les capteurs à portée de communication le reçoivent. Le réseau sans fil ainsi formé est dépourvu d'architecture centrale. Les données mesurées par les capteurs peuvent être collectées et éventuellement agrégées par le réseau.

Ces réseaux de capteurs offrent un vaste domaine d'utilisation, allant de la prévention de catastrophes naturelles (détection des feux de forêt, prévention des tsunamis...) à diverses applications militaires (surveillance, contrôle de robot,...) en passant par des applications à la domotique.

Ces appareils fonctionnant sur batterie, la gestion de l'énergie est un problème crucial dans les réseaux de capteurs. Les communications sans fil représentent une part importante de la consommation énergétique de ces appareils. Chaque envoi et chaque réception de message consomme en effet de l'énergie. Les communications peuvent engendrer un gaspillage d'énergie. En effet, lorsqu'un capteur envoie un message, tous les capteurs à portée de communication le reçoivent. Si le message ne leur est pas destiné, il est ignoré, mais cette réception cause une consommation inutile d'énergie. Ce phénomène est appelé *overhearing*. De plus, si deux messages sont émis en même temps à destination d'un même capteur, ceux-ci sont corrompus et le destinataire ne peut

les recevoir. Les messages doivent alors être envoyés de nouveau, ce qui cause également une dépense inutile d'énergie. Ce phénomène est appelé une *collision*. Nous cherchons à organiser les communications dans le réseau de façon à éviter ce phénomène.

Lorsque le réseau est une clique, il est possible d'organiser les communications en évitant l'overhearing ainsi que toute collision. [BF11] propose d'organiser les capteurs de la clique suivant un anneau virtuel. À un moment donné, seul deux capteurs sont éveillés dans la clique. L'un peut envoyer un message, et l'autre peut recevoir des messages. Les capteurs s'éveillent ainsi deux par deux à tour de rôle, dans l'ordre de l'anneau virtuel. De cette façon, il n'y a aucun overhearing et aucune collision dans le réseau.

C'est cette méthode, très locale, que nous adaptons à un réseau quelconque via le clustering.

Nous commençons par présenter, dans une première section, différents pré-requis à cette étude. Ensuite, nous présentons dans la section 5.2 un algorithme distribué de clustering adapté à la résolution de ce problème. Enfin, dans la section 5.4 nous utilisons ce clustering pour calculer un planning sur les communications dans le réseau.

5.2 Préliminaire

5.2.1 Différence de modèle

Pour fonctionner correctement, le protocole de communication sur une clique [BF11] et l'algorithme de clustering présentés dans ce chapitre nécessite des d'hypothèses supplémentaires sur le modèle.

En effet, dans le protocole à un saut, les capteurs suivent tous un planning de communication. Cela nécessite un système synchrone, et une dérive des horloges négligeable devant le temps d'exécution de l'algorithme.

De plus, chaque capteur dispose d'une période où il peut envoyer des messages et une où il peut en recevoir. Cela ne peut fonctionner que si tous les temps de communication sont bornés par une même borne, petite devant la taille de la fenêtre de communication entre deux capteurs.

On continue de considérer que chaque capteur dispose d'un identifiant unique. Un capteur et son identifiant ne sont à nouveau pas distingués.

Ces hypothèses réalistes sont des hypothèses généralement admises dans les modèles de réseaux de capteurs.

Les communications sont en outre modélisées de manière différente dans les réseaux de capteurs, suivant un schéma *one-to-all*. Ainsi, lorsqu'un capteur envoie un message, celui-ci sera reçu par tous les capteurs à portée de communication.

Enfin, chaque émission et chaque réception de message consomme une certaine quantité d'énergie.

5.2.2 Protocole de communication sur une clique

[BF11] présente un protocole de communication léger, conçu pour fonctionner sur une clique de capteurs. Dans ce protocole, les communications reposent sur une structure d'anneau. Cet anneau représente l'ordre dans lequel les communications vont avoir lieu. En effet, chaque capteur va successivement recevoir des informations de son prédécesseur dans l'anneau, avant de pouvoir en envoyer à son successeur dans l'anneau. Ce processus se répète ensuite de manière périodique. Ainsi, sur une période, chaque capteur dispose de deux créneaux temporels, un durant lequel le capteur pourra recevoir des données (noté R), et un autre durant lequel le capteur pourra émettre (noté S). Le reste du temps, le système de communication du capteur est mis en sommeil. Les

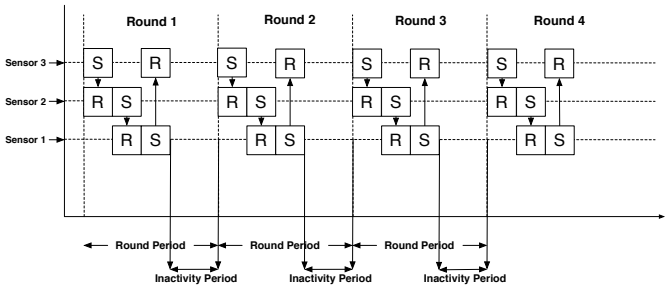

FIGURE 5.1 – Protocole de communication entre les capteurs

capteurs s'éveillent périodiquement deux par deux, de sorte que, à un instant donné, il y a au plus un nœud en émission et un nœud en réception. La figure 5.1 décrit le fonctionnement de ce protocole. Cette façon de faire garantit qu'il n'y aura aucune collision des communications, et permet également d'éviter le phénomène d'overhearing, ce qui diminue la consommation d'énergie des capteurs.

Ce protocole de communication se met en place de manière distribuée. Pour cela, les nœuds commencent par se synchroniser et par s'accorder sur un ordre entre les capteurs. Cet ordre correspond à la relation de précédence dans la structure d'anneau. Ils calculent ensuite un planning de communication, qui se résume globalement à l'ordre dans lequel les capteurs vont communiquer avec leur successeur dans l'anneau.

Ainsi, une collecte des données mesurées par les capteurs peut être organisée. Pour cela, on fait circuler dans le réseau une trame, que les capteurs se transmettent en y adjoignant leurs propres mesures. De cette façon, le dernier capteur en réception dans le planning possède toutes les mesures effectuées par les capteurs de la clique au cours de ce tour.

5.2.3 Adaptation à un réseau quelconque

Le protocole de communication présenté à la section précédente permet de gérer les communications au sein d'une clique. Pour adapter ce protocole à un réseau quelconque, on organise celui-ci en clusters. Chaque cluster est une clique du réseau. Les communications locales, internes à un cluster, sont gérées par le protocole de communication de la section précédente.

Contrairement aux clustering présentés dans les chapitre 2 et 3, les clusters peuvent ici avoir des capteurs en commun. Ainsi, deux clusters sont dit *voisins* s'ils ont au moins un capteur en commun. Ces capteurs communs sont par la suite appelés capteurs *passerelles*.

Un capteur passerelle, en étant impliqué dans le protocole de communication exécuté par chacun des clusters auxquels il appartient, va permettre de faire passer une information d'un cluster à l'autre.

Les communications inter-clusters s'appuient sur les capteurs passerelles. Pour réaliser une communication point-à-point dans le réseau, le message envoyé est transporté durant le protocole de communication exécuté sur son cluster. Il est transmis par un capteur passerelle à un cluster voisin. Il est ainsi transporté à travers une suite de clusters deux à deux voisins, jusqu'à ce qu'il parvienne au cluster de destination et atteigne le capteur destinataire. Une collecte des données mesurées par les capteurs peut également être effectuée de cette façon.

5.3 Clustering adapté au problème

5.3.1 Caractéristique du clustering

Les clusters que nous calculons exécutent le protocole de communication présenté à la section 5.2.2. Par conséquent, ceux-ci sont des cliques du réseau. Afin de diminuer le nombre de clusters, nous calculons des cliques maximales.

Pour qu'une communication inter-clusters soit relayée de clusters en clusters, il faut que les clusters forment une couverture connexe du réseau, c'est-à-dire que chaque capteur fait partie d'au moins un cluster, et que tous les clusters sont reliés à tous les autres par une chaine de clusters voisins. Le graphe des clusters doit donc être un graphe connexe.

Enfin, pour que le clustering calculé reflète l'ensemble des contraintes sur les communications entre capteurs, il faut que chaque arête du réseau fasse partie d'un cluster.

Un tel clustering est présenté en exemple dans la figure 5.2.

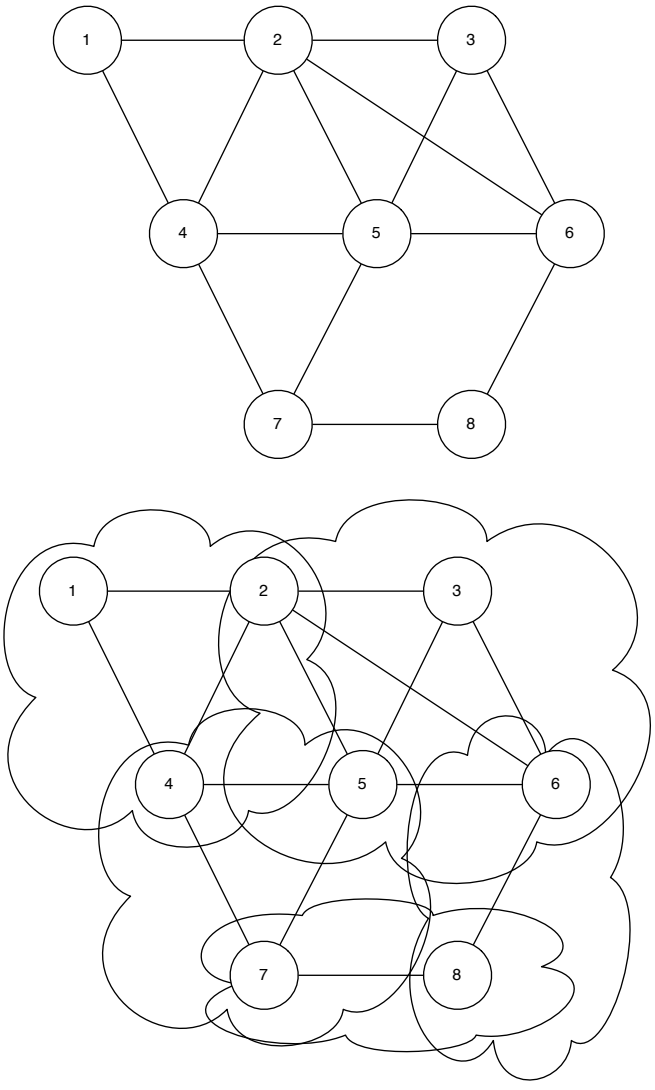

FIGURE 5.2 – Un réseau de capteurs sans fil et son clustering

Le clustering calculé doit donc vérifier la spécification suivante :

Définition 92. *On dit qu'un clustering est valide si et seulement si les propriétés suivantes sont vérifiées :*

- ***Couverture :*** *Chaque arête fait partie d'au moins un cluster ;*
- ***Connectivité :*** *Le graphe formé par le sous-graphe de tous les clusters est connexe ;*
- ***Clique maximale :*** *Tous les capteurs au sein d'un cluster sont voisins deux à deux, et il n'y a aucun autre capteur voisin de chaque capteur du cluster.*

5.3.2 Algorithme détaillé

Pour calculer les clusters, les capteurs échangent des messages pour connaître leurs voisinages à deux sauts. Ensuite, chaque capteur calcule localement les clusters auxquels il appartient. L'ensemble des clusters calculés par cet algorithme est par la suite noté $\mathcal{C}$.

L'algorithme comporte les trois phases suivantes :

- **Phase d'initialisation** durant cette phase, chaque capteur envoie successivement à tous ses voisins son identifiant, puis son voisinage. Lorsqu'un capteur reçoit ces données de l'un de ses voisins, il met à jour ses variables locales. À la fin de cette étape, chaque capteur connaît son voisinage à deux sauts.
- **Phase de calcul des cliques** Durant cette phase, chaque capteur va calculer localement les cliques dont il fait partie.
- **Phase de nettoyage** Dans cette dernière étape, les clusters qui ne sont pas nécessaires sont effacés. Un cluster n'est pas nécessaire lorsque toutes ses arêtes se trouvent déjà dans d'autres clusters.

Première phase - Calcul du voisinage à deux sauts

Durant cette phase, chaque capteur collecte une vision de son voisinage à deux sauts. Pour cela, chaque capteur envoie un message contenant son identifiant (ligne 7 et 8 algorithme 20). Lorsqu'un capteur reçoit un tel message, cela signifie que le capteur identifié est un de ses voisins. Il l'ajoute donc à l'ensemble de ses voisins, noté $Neigh$ (ligne 12 à 15 algorithme 21). Chaque capteur attend d'avoir reçu l'identifiant de chacun de ses voisins. À la fin de cette étape, chaque capteur connaît l'ensemble de ses voisins.

Ensuite, chaque capteur envoie à chacun de ces voisins un message contenant l'ensemble de ses voisins (ligne 18 et 19 algorithme 20). Lorsqu'un capteur reçoit un tel message, il enregistre ces informations dans un tableau de tableaux, noté $Neigh2$ (ligne 23 à 31). Chaque capteur attend d'avoir reçu ce message en provenance de chacun de ses voisins. La réunion de tous ces tableaux forme le voisinage à deux sauts du capteur, calculé au fur et à mesure des réceptions dans un tableau noté $2hop\_Neigh$.

À la fin de cette phase, chaque capteur connaît son voisinage à deux sauts, ainsi que le sous-graphe composé de ces capteurs et des liens de communication les reliant. Ce sous-graphe du graphe de communication sera utilisé durant la deuxième et la troisième phase.

En appliquant cet algorithme à l'exemple présenté dans la figure 5.2, on obtient les résultats ci-dessous. Nous ignorons délibérément les variables $Neigh2_{i,j}$ puisque $Neigh2_{i,j}$ est simplement la variable $Neigh_j$ que le capteur i a reçu et stocké.

Var.\ Sens.	1	2	3
Neigh	2,4	1,3,4,5,6	2,5,6
2h-Neigh	1,2,3,4,5,7	1,2,3,4,5,6,7,8	1,2,3,4,5,6,7,8

Algorithme 20 Phase d'initialisation sur le capteur i

$Neigh_i \leftarrow \emptyset$
pour tout $j \in N_i$ **faire**
  $Neigh2_{i,j} \leftarrow \emptyset$
fin pour
$2hop\_Neigh_i \leftarrow \emptyset$
pour tout $j \in N_i$ **faire**
  {Indiquer son identifiant à tous ses voisins}
  Envoyer id_i à j
fin pour
$waiting \leftarrow N_i$
tant que $waiting \neq \emptyset$ **faire**
  {À la réception de l'identifiant d'un voisin}
  Recevoir id_j en provenance d'un voisin j
  $Neigh_i \leftarrow Neigh_i \cup \{id_j\}$
  $waiting \leftarrow waiting \setminus \{j\}$
fin tant que
pour tout $j \in N_i$ **faire**
  {Envoi du voisinage à ses voisins}
  Envoyer $Neigh_i$ to j
fin pour
$waiting \leftarrow N_i$
tant que $waiting \neq \emptyset$ **faire**
  {À la réception du voisinage d'un voisin}
  Recevoir de $Neigh_j$ en provenance d'un voisin j
  $Neigh2_{i,j} \leftarrow Neigh_j$
  pour tout $k \in Neigh_j$ **faire**
    si $k \notin 2hop\_Neigh_i$ **alors**
      $2hop\_Neigh_i \leftarrow 2hop\_Neigh_i \cup \{k\}$
    fin si
  fin pour
  $waiting \leftarrow waiting \setminus \{j\}$
fin tant que

Var.\ Sens.	4	5	6
Neigh	1,2,5,7	2,3,4,6,7	2,3,5,8
2h-Neigh	1,2,3,4,5,6,7,8	1,2,3,4,5,6,7,8	1,2,3,4,5,6,7

Var.\ Sens.	7	8
Neigh	4,5,8	6,7
2h-Neigh	1,2,3,4,5,6,7	2,3,4,5,6,8

Cet algorithme nécessite que chaque capteur envoie deux messages à chacun de ses voisins, ce qui entraîne $2\deg(i)$ réception de message sur chaque capteur i, soit au total $2m$ messages.

Deuxième phase - Calcul des clusters

Cette deuxième phase débute lorsque chaque capteur connaît son voisinage à deux sauts. Elle se déroule sans échange de message. Durant cette phase, chaque capteur calcule les clusters

auxquels il appartient à l'aide des informations collectées durant la première phase, ainsi que les clusters adjacents.

Le clustering calculé doit vérifier une propriété de couverture des arêtes. Chaque arête doit être contenue dans un cluster. Ainsi, pour chaque arête (j, k) dont il a connaissance, le capteur calcule une clique maximale contenant cette arête (algorithme 21). Les deux capteurs j et k attachés à cette arête forment une nouvelle clique $C_{j,k}$ (ligne 3). Ensuite, pour chaque voisin de j, on vérifie s'il est voisin avec les capteurs déjà présents dans $C_{j,k}$. Si c'est le cas, ce capteur est ajouté à la clique $C_{j,k}$. De cette façon, une clique maximale contenant l'arête (j, k) est calculée.

Algorithme 21 Phase de calcul des cliques sur le capteur i

```
pour tout  j ∈ 2hop_Neigh_i faire
  pour tout k ∈ Neigh2_{i,j} faire
    C_{j,k} ← {j,k}
    {C_{j,k} est une clique contenant l'arrête (j,k)}
    pour tout l ∈ Neigh2_{i,j} faire
      {Un capteur l voisin de tous les capteurs déjà dans C_{j,k} est ajouté à cette clique}
      Ok ← vrai
      pour tout m ∈ C_{j,k} faire
        si m ∉ Neigh2_{i,l} alors
          Ok ← faux
        fin si
      fin pour
      si Ok alors
        C_{j,k} ← C_{j,k} ∪ {l}
      fin si
    fin pour
    {C_{j,k} est alors maximale}
    C ← C ∪ C_{j,k}
  fin pour
fin pour
{C est un ensemble de cliques maximales qui contiennent chaque arrêtes du voisinage à deux
sauts du capteur i}
```

À la fin de cette seconde phase, pour l'exemple présenté à la figure 5.2, $\mathcal{C} = \{\{1, 2, 4\}, \{2, 4, 5\}, \{2, 3, 5, 6\}, \{4, 5, 7\}, \{6, 8\}, \{7, 8\}\}$.

Considérons δ une borne supérieure sur les degrés des capteurs. Puisqu'une clique ne peut pas contenir plus de δ capteurs (vu que tous les capteurs d'une clique sont voisins), la boucle la plus interne de cet algorithme est en $O(\delta^2)$ et chacune des trois autres boucles est en $O(\delta)$. Ainsi, la complexité de cet algorithme local est $O(\delta^5)$.

Troisième phase - Phase de nettoyage

La phase de nettoyage vise à supprimer les clusters qui ne sont pas nécessaires pour traduire l'ensemble des contraintes sur les communications. En effet, le fait que les clusters sont des cliques maximales garantit qu'un cluster ne peut être contenu dans un autre. Cependant un cluster peut-être inclus dans la réunion de plusieurs autres clusters. Ainsi, si toutes les arrêtes couvertes par un cluster c sont présentes dans d'autres clusters, le cluster c n'est pas utile et peut-être supprimé.

Par exemple dans l'exemple précédent (fig 5.2), la clique $\{2,4,5\}$ peut être supprimée puisque les arrêtes $(2,4)$, $(2,5)$, et $(4,5)$ appartiennent respectivement aux cliques $\{1,2,4\}$, $\{2,3,5,6\}$ et $\{4,5,7\}$.

Pour chaque clique C contenant le capteur courant i, l'algorithme vérifie si cette clique est nécessaire. Pour cela, le capteur vérifie si toute les arêtes de C sont dans un autre cluster D (ligne 3 à 13). Si c'est le cas, alors la clique C peut être effacée (ligne 15 à 19).

Algorithme 22 Phase de nettoyage sur le capteur i

```
pour tout C ∈ 𝒞 with i ∈ C faire
  discard ← vrai
  pour tout (l,m) ∈ C² with l ≠ m faire
    found ← faux
    pour tout D ∈ 𝒞, D ≠ C ∧ ¬found faire
      si (l,m) ∈ D alors
        found ←vrai
      fin si
    fin pour
    si ¬found alors
      {C ne peut être supprimé car l'arrête (l,m) n'est dans aucune autre clique}
      discard ← faux
    fin si
  fin pour
  si discard alors
    {C peut être effacé car chacune de ses arrête est présente dans une autre clique}
    𝒞 ← 𝒞\C
  fin si
fin pour
```

À la fin de la procédure de clusterisation, pour l'exemple donné dans la figure 5.2, $\mathcal{C} = \{\{1,2,4\}, \{2,3,5,6\}, \{4,5,7\}, \{6,8\}, \{7,8\}\}$. Le cluster $\{2,4,5\}$ est effectivement supprimé lors de cette phase.

La complexité de cet algorithme local est majorée par $O(\delta^5)$ ($O(\delta)$ pour la boucle de la ligne 1, et $O(\delta^2)$ pour les boucles des lignes 3 et 5).

5.4 Ordonnancement des communications - Modèle mathématique

Nous montrons maintenant que ce clustering est adapté à l'établissement d'un planning de communication. Pour cela, nous utilisons les clusters construits précédemment afin de calculer un planning évitant toute collision. Nous utilisons des techniques classiques de recherche opérationnelle et présentons cette recherche de planning comme un programme mathématique.

On peut obtenir facilement un planning valide. On choisit un cluster, qui exécutera le protocole présenté à la section 5.2. Lorsque ce protocole est terminé, on choisit un autre cluster, qui exécute à son tour le protocole. On continue ainsi jusqu'à ce que tous les clusters aient exécuté le protocole. De cette façon, on obtient un planning évitant toute collision, puisqu'il n'y a à chaque instant une seule communication dans le réseau.

Ce planning est cependant inutilement long et n'exploite pas le parallélisme. En effet, les

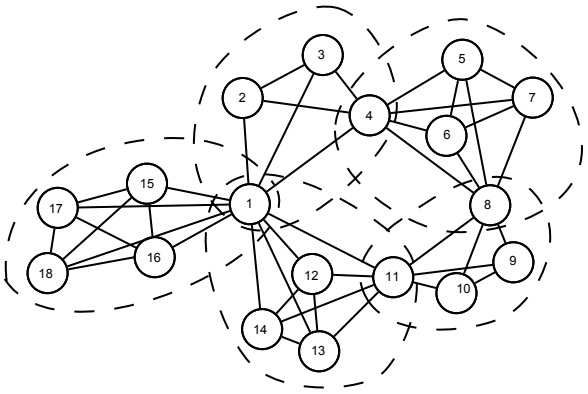

FIGURE 5.3 – Graphe de communication d'un réseau de capteur sans fil

communications au sein de deux clusters éloignées dans le réseau ne peuvent pas interférer : ces clusters peuvent donc exécuter en parallèle ce protocole, sans qu'il n'y ait aucun risque de collision.

Ce n'est cependant pas le cas pour deux clusters voisins, les messages envoyés par l'un pouvant entrer en collision avec les messages envoyés par l'autre. Il faudra donc faire attention, lorsque deux clusters sont voisins, à la planification des communications des capteurs passerelles. En effet, seul un capteur appartenant à deux clusters en même temps risque de subir ou de causer une collision.

Dans cette section, nous montrons par un exemple que les communications peuvent être grandement parallélisées. Nous modélisons ensuite les contraintes sur les communications qui définissent un planning valide. Nous commençons par présenter les variables de décision du problème, puis nous présentons les contraintes qui permettent de définir un planning correspondant au protocole de communication pour les cliques présenté à la section 5.2. Nous exposons ensuite les contraintes sur les communications des cliques ayant des capteurs communs. Nous discutons enfin de la fonction objectif et de son interprétation, avant d'expliquer les méthodes permettant de résoudre ce problème mathématique.

5.4.1 Exemple de planification

L'exemple donné ici illustre le résultat de l'algorithme de clustering, ainsi que les contraintes qu'il indique sur l'organisation des communications.

Soit le réseau présenté dans la figure 5.3.

Premièrement, nous appliquons l'algorithme de clustering présenté dans la section 5.3.1 à ce réseau. Les clusters obtenus sont les suivants : $A = \{1, 2, 3, 4\}$, $B = \{4, 5, 6, 7, 8\}$, $C = \{8, 9, 10, 11\}$, $D = \{1, 11, 12, 13, 14\}$ et $E = \{1, 15, 16, 17, 18\}$.

On remarque que, dans un cluster, les rôles des capteurs qui ne sont pas passerelles sont totalement symétriques. En effet, les places dans le planning de deux de ces capteurs, par exemple les capteurs 2 et 3 dans l'exemple précédent, peuvent être échangés sans que cela change la validité du planning.

Un planning de communication valide sur ce réseau est présenté dans la figure 5.4.

Dans cet exemple, les clusters A et C peuvent communiquer en même temps, puisqu'ils n'ont aucun nœud en commun. Les clusters B et D sont dans la même situation.

$n\backslash t$	1	2	3	4	5	6	7	8	9	10	11
1	R	S			R	S				R	S
2		R	S								
3	S			R							
4			R	S				R	S		
5					S				R		
6							R	S			
7						R	S				
8	R	S			R	S					
9		R	S								
10	S			R							
11			R	S				R	S		
12						R	S				
13							R	S			
14					S				R		
15									R	S	
16								R	S		
17							R	S			
18							S				R

FIGURE 5.4 – Planning de communication pour tout le réseau

$n\backslash t$	1	2	3	4
1	R	S		
2		R	S	
3	S			R
4			R	S

FIGURE 5.5 – Planning pour le cluster A

Ici, nous avons choisi de placer les communications des clusters A et C au début du planning, suivies dès que possible par les communications des clusters B et D. Les communications du cluster E terminent le planning.

Pour le cluster A, les deux premières périodes de communication sont utilisées pour la communication du capteur 1, qui est un capteur passerelle avec le cluster D. Cela permet de lever au plus tôt les contraintes empêchant la communication du cluster D. De manière similaire, les communications du cluster C démarrent avec le capteur 8. Puisque les capteurs 4 et 8 font partie du cluster B, ils ne peuvent être y impliqués dans une communication en même temps. Ainsi le capteur 4 peut émettre pour le protocole du cluster A seulement aux périodes 3 et 4. Pour les même raisons, le capteur 11 peut communiquer uniquement aux périodes 3 et 4. Cela nous amène aux plannings présentés dans les figures 5.5 et 5.6.

Nous construisons le planning des communications pour les clusters B et D, présentés dans les figures 5.7 et 5.8, de la même manière. Puisque le capteur 4 est impliqué dans une communication lors de la plage 4, les capteurs du cluster B ne peuvent pas communiquer avant la période 5 ; de manière similaire, les capteurs du cluster D ne peuvent pas communiquer avant la période 5 à cause du capteur 11.

$n\backslash t$	1	2	3	4
8	R	S		
9		R	S	
10	S			S
11			R	S

FIGURE 5.6 – Planning pour le cluster C

$n\backslash t$	5	6	7	8	9
4				R	S
5	S				R
6			R	S	
7		R	S		
8	R	S			

FIGURE 5.7 – Planning pour le cluster B

$n\backslash t$	5	6	7	8	9
1	R	S			
11				R	S
12		R	S		
13			R	S	
14	S				R

FIGURE 5.8 – Planning pour le cluster D

$n\backslash t$	7	8	9	10	11
1				R	S
15			R	S	
16		R	S		
17	R	S			
18	S				R

FIGURE 5.9 – Planning pour le cluster E

Enfin, il faut placer les communications du cluster E. Puisque les capteurs du cluster D communiquent jusqu'à la période 9, le capteur 1 ne peut pas communiquer dans le protocole du cluster E avant la période 10. C'est pour cela que les communications du capteur 1 pour le clusters E sont placées aux périodes 10 et 11 (figure 5.9).

La combinaison des plannings pour chaque cluster donne la solution présentée dans la figure 5.4.

5.4.2 Variables

Ce modèle s'appuie sur la connaissance de :

– $G = (V, E)$: le graphe de communication du système et

– $\mathcal{C}$: l'ensemble des clusters calculés par l'algorithme de clustering de la section 5.2.

Dans un premier temps, on introduit un majorant de la durée du planning calculé, que l'on note T, afin de limiter l'horizon de notre recherche.

Il est possible de construire un planning sans aucune collision en plaçant les communications les unes à la suite des autres, et en ne plaçant qu'une communication dans chaque période. Dans chaque cluster, les capteurs qui le composent communiquent chacun à leur tour. La durée de ce planning est égale au nombre de capteurs du réseau, chacun compté autant de fois que le nombre de cluster auquel il appartient.

Nous cherchons donc un planning d'une longueur inférieure à $T = \sum_{c \in \mathcal{C}} |c|$.

Nous définissons les variables de décision suivantes, dont l'instanciation définit un planning :

– $\forall c \in \mathcal{C}, \forall t \in \{0, \ldots, T\}, \forall i \in C$,

$$e_i^c(t) = \begin{cases} 1 \text{ si } i \text{ est émetteur au temps } t \text{ pour le cluster } c \\ 0 \text{ sinon} \end{cases}$$

– $\forall c \in \mathcal{C}, \forall t \in \{0, \ldots, T\}, \ \forall i \in C$,

$$r_i^c(t) = \begin{cases} 1 \text{ si } i \text{ est en réception au temps } t \text{ pour le cluster } c \\ 0 \text{ sinon} \end{cases}$$

– $\forall c \in \mathcal{C}, \forall i \in C$,

$$s_i^c = \begin{cases} 1 \text{ si } i \text{ commence le cycle de communication du cluster } c \\ 0 \text{ sinon} \end{cases}$$

Nous utilisons la convention que $\forall c \in \mathcal{C}, \forall i \in c, \forall t \leqslant 0, e_i^c(t) = 0$.

5.4.3 Contraintes définissant le protocole de communication sur une clique

Le premier capteur qui communique dans un cluster, au cours d'une ronde, est appelé capteur initiateur. Selon le protocole de communication décrit dans la section 5.2, un capteur non-initiateur est en émission durant la période suivant sa période de réception (communication en anneau). La variable $r_i^c(t)$ peut ainsi être déduite de $e_i^c(t)$ et de $s_i^c(t)$. Un capteur non initiateur i sera en émission juste après avoir été en réception. Pour ces nœuds, on a $s_i^c = 0$ et $r_i^c(t) = e_i^c(t+1)$.

Si i est initiateur, il est en réception à la fin du protocole, et en émission au début. Il est donc émetteur $|c|$ périodes avant d'être en réception. Pour ces capteurs, on a $s_i^c = 1$, et $e_i^c(t - |c|) = r_i^c(t)$.

Par conséquent, $r_i^c(t) = (1 - s_i^c) \times e_i^c(t+1) + s_i^c \times e_i^c(t - |c|)$.

Cette équation n'est pas linéaire, et pour utiliser des méthodes de résolution de programme linéaire en nombre entier, nous la remplaçons par un ensemble d'équations ou d'inéquations linéaires équivalent.

Puisque toutes ces variables sont positives, cette équation implique :

$$\begin{cases} r_i^c(t) \geqslant (1 - s_i^c) \times e_i^c(t+1) \\ r_i^c(t) \geqslant s_i^c \times e_i^c(t - |c|) \end{cases}$$

De plus, puisque $s_i^c(t) = 0$ ou $1 - s_i^c(t) = 0$, alors ces inéquations impliquent que $r_i^c(t) \geq (1 - s_i^c) \times e_i^c(t+1) + s_i^c \times e_i^c(t - |c|)$, et puisque tous les capteurs sont en émission et en réception exactement une fois pour le cluster (ce qui est garantit par une contrainte exprimée plus loin), les contraintes exprimées ci-dessus sont équivalentes à $r_i^c(t) = (1 - s_i^c) \times e_i^c(t+1) + s_i^c \times e_i^c(t - |c|)$.

Maintenant, puisque pour a et b dans $\{0, 1\}$, on a $a.b \geq a + b - 1$, alors

$$\begin{cases} r_i^c(t) \geqslant (1 - s_i^c) + e_i^c(t+1) - 1 = e_i^c(t+1) - s_i^c \\ r_i^c(t) \geqslant s_i^c + e_i^c(t - |c|) - 1. \end{cases}$$

Et réciproquement, ces deux inégalités sont équivalentes aux précédentes puisque $r_i^c(t) \geq 0$.

Cela nous amène à l'ensemble de contraintes linéaires suivant :

$$\forall c \in \mathcal{C}, \forall i \in c, \forall t \in \{0, \ldots, T\},\ r_i^c(t) \geqslant e_i^c(t+1) - s_i^c \tag{5.1}$$

$$\forall c \in \mathcal{C}, \forall i \in c, \forall t \in \{0, \ldots, T\},\ r_i^c(t) \geqslant s_i^c + e_i^c(t - |c|) \tag{5.2}$$

Il y a au plus un capteur en émission dans un cluster à un instant donné :

$$\forall c \in \mathcal{C}, \forall t \in \{0, \ldots, T\}, \sum_{i \in c} e_i^c(t) \leqslant 1 \tag{5.3}$$

Et il y a au plus un capteur en réception dans un cluster :

$$\forall c \in \mathcal{C}, \forall t \in \{0, \ldots, T\}, \sum_{i \in c} r_i^c(t) \leqslant 1 \tag{5.4}$$

Pour un cluster c, tous les capteurs de ce cluster sont émetteurs exactement une fois (pour c) durant un cycle, ce qui implique que pour chaque capteur i, il y a une unique période t telle que i est en émission :

$$\forall c \in \mathcal{C}, \forall i \in c,\ \sum_{t=0}^{T} e_i^c(t) = 1 \tag{5.5}$$

Et tous les capteurs sont également en émission une unique fois :

$$\forall c \in \mathcal{C},\ \forall i \in c,\ \sum_{t=0}^{T} r_i^c(t) = 1 \tag{5.6}$$

Lorsqu'un capteur est en émission pour un cluster, alors il y a au moins un capteur qui l'écoute, c'est-à-dire qui est en réception. Si le capteur i n'est pas en émission, alors $e_i^c(t) = 0$, la contrainte suivante n'est donc pas restrictive :

$$\forall c \in \mathcal{C},\ \forall i \in c, \forall t \in \{0, \ldots, T\}, e_i^c(t) \leqslant \sum_{j \in c} r_j^c(t) \tag{5.7}$$

De la même manière, lorsqu'un capteur est en réception pour un cluster, alors il y a au moins un capteur en émission dans ce cluster :

$$\forall c \in \mathcal{C},\ \forall i \in c, \forall t \in \{0, \ldots, T\}, r_i^c(t) \leqslant \sum_{j \in c} e_j^c(t) \tag{5.8}$$

5.4.4 Contraintes entre les clusters

Les contraintes entre les clusters modélisent le fait que lorsqu'un capteur passerelle est en émission pour un cluster c, aucun de ses voisins dans un autre cluster que c ne peut être en réception. De même, lorsque ce nœud est en réception pour un cluster c, aucun nœud d'un autre cluster ne peut être en émission.

Lorsqu'un capteur i appartient à plusieurs clusters, si i est en émission, aucun capteur faisant partie d'un autre cluster contenant i ne peut être en réception :

$$\forall c \in \mathcal{C}, \forall i \in c, \forall t \in \{0, \ldots, T\},$$
$$\sum_{\substack{c' \neq c \\ c' \ni i}} \sum_{j \in c'} r_j^{c'}(t) \leqslant (1 - e_i^c(t)) \times n \times |\mathcal{C}| \tag{5.9}$$

La quantité $n \times |\mathcal{C}|$ est forcément toujours plus grande que $\sum_{c' \neq c, c' \ni i} \sum_{j \in c'} r_j^{c'}(t)$, puisque $r_j^{c'}(t)$ est toujours plus petit que 1. Ainsi, cette contrainte implique simplement que lorsqu'un capteur i est en émission pour le cluster c (lorsque $e_i^c(t) = 1$), alors $\sum_{c' \neq c, c' \ni i} \sum_{j \in c'} r_j^{c'}(t) = 0$, c'est-à-dire que pour tout j, $r_j^{c'}(t) = 0$ (puisque $r_j^{c'}(t) \geq 0$). Ainsi, aucun capteur j dans un cluster c' différent de c et contenant i ne peut être en réception.

De la même manière, si i est en réception pour un cluster, aucun capteur dans un autre cluster contenant i ne peut être en émission :

$$\forall c \in \mathcal{C}, \forall i \in c, \forall t \in \{0, \ldots, T\},$$
$$\sum_{\substack{c' \neq c \\ c' \ni i}} \sum_{j \in c'} e_j^{c'}(t) \leqslant (1 - r_i^c(t)) \times n \times |\mathcal{C}| \tag{5.10}$$

Les contraintes 5.1 à 5.10 définissent un planning de communication dans lequel tous les capteurs sont émetteurs et récepteurs exactement une fois pour chacun de leurs clusters, et tel qu'il ne peut y avoir de collision dans les communications.

5.4.5 Fonction objectif et interprétation

Tout planning vérifiant les contraintes 1 à 10 est un planning valide. Cependant, nous cherchons ici un planning optimal sous certains critères. Ces critères sont exprimés par la fonction objectif, que nous visons à minimiser (ou à maximiser).

Un exemple possible est d'essayer d'établir un planning le plus court possible. Cela pourrait s'appliquer à un réseau ou les informations locales doivent être mises à jour aussi souvent que possible.

Pour décrire une telle situation, nous introduisons une nouvelle variable Y qui représente un majorant de la durée du planning considéré. Pour chaque période t, s'il existe un capteur en émission à cette période, c'est-à-dire s'il existe un $e_i^c(t) \neq 0$, alors $Y \geqslant t$. En effet, si $e_i^c(t) = 1$, alors $Y \geqslant t \times e_i^c(t)$. À l'inverse, si $e_i^c(t) = 0$, cette contrainte est $Y \geq 0$, ce qui est toujours vrai.

Cela nous amène à introduire les contraintes suivantes :

$\forall t \in \{0, \ldots, T\}, \forall c \in \mathcal{C},\ \forall i \in c,$

$$Y \geqslant t \times e_i^c(t) \tag{5.11}$$

Ensuite, pour avoir un planning le plus court possible, nous cherchons à minimiser Y. Le plus petit majorant est la longueur du planning le plus court possible. La fonction objectif est donc :

$$\min\{Y\} \tag{5.12}$$

5.4.6 Résolution

Les variables $e_i^c(t)$, $r_i^c(t)$, s_i^c et Y, les contraintes 1 à 11, ainsi que la fonction objectif $\min\{Y\}$ forment un programme linéaire en nombre entier.

La résolution d'un tel programme est un problème classique de recherche opérationnelle. Résoudre un problème de programmation linéaire en nombres entiers est un problème NP-complet. Cette méthode nécessite (au pire) un nombre d'opérations élémentaires qui croît exponentiellement avec le nombre de variables et de contraintes.

Les solutions d'un tel programme peuvent être calculées par un solveur comme par exemple CPLEX, qui donnera le planning le plus court possible pour un réseau donné.

Cette méthode centralisée (qui n'a pas vocation à être implémentée sur les capteurs) calcule un planning de taille la plus courte possible. Cela permet de fournir une borne inférieure sur la taille des plannings que les capteurs pourraient calculer de manière distribuée.

5.5 Conclusion

Dans ce chapitre, nous avons utilisé le clustering pour résoudre le problème des collisions dans les réseaux de capteurs. Pour éviter les collisions, nous cherchons à organiser les communications à l'aide d'un planning.

Nous avons présenté, dans un premier temps, une méthode distribué permettant à une clique de capteurs de communiquer sans aucune collision. Les nœuds s'accordent sur l'ordre dans lequel ils communiquent, puis les nœuds s'éveillent et communiquent deux à deux suivant cet ordre.

Pour organiser les communications dans un réseau quelconque, nous avons proposé de calculer des clusters, sur lesquels ce protocole de communication peut être utilisé. Nous avons présenté un algorithme distribué calculant un clustering adapté à ce problème : les clusters sont des cliques maximales du réseau.

Nous avons ensuite montré que ce clustering exprime les contraintes nécessaires pour planifier les communications dans le système. Pour cela, nous exprimons le calcul d'un planning comme un programme linéaire en nombres entiers, exprimé à l'aide des clusters.

Les méthodes classiques de recherche opérationnelle fournissent une méthode centralisée pour calculer un planning de communication optimal évitant les collisions.

Chapitre 6

Conclusion

Dans cette thèse, nous nous sommes intéressés à l'organisation et à la gestion des systèmes distribués dynamiques. Les réseaux ayant une taille sans cesse croissante, il est plus que jamais crucial de mettre en place des structures permettant l'organisation et le contrôle de ces systèmes. Nous proposons pour cela de mettre en place un clustering hiérarchique imbriqué du système. Cette structure est constituée d'une partition du réseau en parties connexes disjointes, appelées clusters. Ces clusters sont à leur tour regroupés en clusters de clusters. Les clusters de niveau k forment les nœuds du niveau $k+1$, qui sont regroupés en clusters.

Dans un réseau de grande taille, les changements topologiques ne sont plus des phénomènes exceptionnels, et ils doivent être pris en compte. Le clustering hiérarchique doit donc être capable de s'adapter en cas de changement topologique.

Pour construire cette structure, nous utilisons un algorithme de clustering orienté taille, présenté au chapitre 2. Cet algorithme construit un clustering orienté taille du système à l'aide de marches aléatoires. Chaque cluster possèdent une partie dominante connexe, dont la taille est le plus souvent possible égale à un paramètre de l'algorithme. Cela nous permet d'avoir un contrôle sur la taille des clusters.

Cet algorithme réagit aux changements topologiques : lorsqu'un changement topologique survient dans un cluster, celui-ci finit par être réparé ou détruit. Cette réparation est un processus local, dans le sens où seul ce cluster, et éventuellement ceux qui lui sont adjacents, sont modifiés.

Cet algorithme est utilisé par la suite pour construire un clustering hiérarchique du système.

Pour cela, nous avons définit au chapitre 3 les primitives nécessaires pour que le clustering forme un nouveau système distribué. Les clusters, nœuds de ce nouveau système, doivent se comporter comme tels. Ils doivent être capable de détecter dynamiquement leur voisinage, de communiquer par messages avec les clusters adjacents et également d'exécuter atomiquement un algorithme. Ces mécanismes forment ainsi une couche d'abstraction, définissant un système distribué exécutant l'algorithme distribué de notre choix.

Cette couche d'abstraction nous a permis de construire un clustering hiérarchique du système par une approche ascendante. À l'aide de l'algorithme de clustering, nous calculons un clustering du système distribué initial. Grâce à notre couche d'abstraction, les clusters ainsi calculés forment un nouveau système distribué, qui exécute également l'algorithme de clustering du chapitre 3. Il en résulte le calcul de clusters de clusters, qui forment à leur tour un nouveau système distribué. En itérant ce processus, on obtient un clustering hiérarchique imbriqué du système.

Puisque notre algorithme de clustering est écrit pour des systèmes dynamiques, le clustering hiérarchique calculé s'adapte aux changements topologiques.

Le fonctionnement de cet algorithme dépend essentiellement de la capacité pour le cluste-

ring de former un nouveau système distribué. Ainsi, pour démontrer le bon fonctionnement de l'algorithme de clustering hiérarchique, nous avons formulé au chapitre 4 une spécification, en terme d'exécutions, du fait que le clustering émule un système distribué. Nous avons ainsi défini un système formel dont les nœuds sont les clusters calculés sur le système initial.

Les configurations du système émulé peuvent être extraites des configurations du système initial. Il existe ainsi une fonction extrayant une configuration du système émulé à partir de toute configuration du système initial.

Au cours de toute exécution, des modifications sont apportées aux configurations extraites. Ces modifications forment une suite de transitions de l'algorithme du système émulé, ainsi que d'éventuels changements topologiques autorisés par le modèle.

La fonction d'extraction transforme ainsi toute exécution de l'algorithme d'émulation en une exécution de l'algorithme distribué choisi sur le système émulé. De plus, cette exécution vérifie les propriétés du modèle : les messages émis finissent par être reçus et les changements topologiques finissent par être détectés. Le système émulé est donc bien un système distribué exécutant l'algorithme de notre choix.

Cette propriété nous a permis par la suite de démontrer par récurrence le bon fonctionnement de l'algorithme hiérarchique présenté à la section 3.4.

Enfin, au chapitre 5, nous utilisons cette démarche pour résoudre un problème issu des réseaux de capteurs, la gestion des collisions. En effet, lorsque deux capteurs envoient simultanément un message à un de leur voisin commun, un phénomène d'interférence peut corrompre ces messages et rendre leur réception impossible. Ces messages doivent alors être envoyés à nouveau, ce qui est coûteux en énergie. Pour éviter ce phénomène, nous proposons de planifier les communications au sein du réseau de capteurs.

Lorsque le réseau est une clique, il est possible d'organiser les communications à l'aide d'une structure d'anneau virtuel. Les nœuds commencent par mettre en place une structure d'anneau dans le réseau, puis ils émettent et reçoivent chacun leur tour le long de l'anneau.

Pour adapter cette méthode à un réseau quelconque, nous proposons de calculer un clustering adapté du système. Les clusters sont des cliques maximales du réseau. Les communications au sein de chaque cluster peuvent ainsi être organisées par un anneau virtuel. Le clustering que nous calculons représente l'intégralité des contraintes portant sur les communications simultanées.

Nous avons ensuite présenté une méthode de calcul de planning s'appuyant sur des méthodes classiques de recherche opérationnelle. La recherche d'un planning est alors exprimée comme un programme linéaire binaire. La résolution de ce problème par des méthodes classiques de Branch and Bound permet l'obtention d'un planning garantissant qu'aucune collision ne peut survenir dans le système.

Cette étude soulève quelques questions. Le clustering hiérarchique que nous calculons s'adapte aux changements topologiques, mais n'est pas auto-stabilisant. En particulier, nous n'avons aucun mécanisme permettant de détecter la perte d'un jeton *Token* en cas de dysfonctionnement du média de communication. L'introduction des vagues de réinitialisation, présentes dans [BKS12] n'est pas suffisante pour rendre notre algorithme auto-stabilisant. En effet, ce mécanisme nécessite pour fonctionner d'avoir un modèle synchrone et des temps de communications bornés, hypothèses non vérifiées par le système émulé par les clusters. Nous envisageons de chercher à un autre moyen de rendre cet algorithme auto-stabilisant, ou tout au moins un autre moyen de gérer la perte de jeton *Token*, d'une façon rendant compatible le résultat du clustering avec le modèle d'exécution.

Pour calculer un clustering hiérarchique, nous avons doté les clusters de la capacité de se comporter comme les nœuds d'un nouveau système. Les clusters forment un système distribué, capable d'exécuter n'importe quel algorithme distribué $\mathcal{A}$. Nous avons choisi d'exécuter notre

algorithme de clustering sur le système des clusters, mais nous pourrions exécuter n'importe quel autre algorithme écrit pour un système conforme à notre modèle. Nous voulons explorer les possibilités offertes par cette couche d'abstraction. Quels sont les algorithmes intéressants à exécuter sur un clustering à un seul niveau, sur les différents niveaux de notre structure hiérarchique ?

Enfin, notre algorithme d'émulation produit un système distribué conforme à notre modèle, qui contient très peu d'hypothèses. Est-il possible d'ajouter des hypothèses à ce modèle ? Peut-on forcer le système émulé à avoir un comportement particulier, et si oui, avec quelles hypothèses sur le modèle du système initial ? Nous comptons mener une étude sur les liens entre le modèle du système initial et le comportement du système émulé.

Bibliographie

[APVH00] A.D. Amis, R. Prakash, T.H.P. Vuong, and D.T. Huynh. Max-min d-cluster formation in wireless ad hoc networks. In *INFOCOM 2000. Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, volume 1, pages 32–41 vol.1, 2000.

[Bas99a] S. Basagni. Distributed and mobility-adaptive clustering for multimedia support in multi-hop wireless networks. In *Vehicular Technology Conference, 1999. VTC 1999 - Fall. IEEE VTS 50th*, volume 2, pages 889–893 vol.2, 1999.

[Bas99b] S. Basagni. Distributed clustering for ad hoc networks. In *Parallel Architectures, Algorithms, and Networks, 1999. (I-SPAN '99) Proceedings. Fourth International-Symposium on*, pages 310–315, 1999.

[BBF05] Thibault Bernard, Alain Bui, and Olivier Flauzac. Topological adaptability for the distributed token circulation paradigm in faulty environment. In Jiannong Cao, LaurenceT. Yang, Minyi Guo, and Francis Lau, editors, *Parallel and Distributed Processing and Applications*, volume 3358 of *Lecture Notes in Computer Science*, pages 146–155. Springer Berlin Heidelberg, 2005.

[BBPS10] Thibault Bernard, Alain Bui, Laurence Pilard, and Devan Sohier. A distributed clustering algorithm for dynamic networks. *CoRR*, abs/1011.2953, 2010.

[BBS13] Thibault Bernard, Alain Bui, and Devan Sohier. Universal adaptive self-stabilizing traversal scheme : Random walk and reloading wave. *Journal of Parallel and Distributed Computing*, 73(2) :137 – 149, 2013.

[BCS12] A Bui, S Clavière, and D Sohier. A top-down algorithm for clustering in large-scale distributed networks. In *PDPTA'12 - The 2012 International Conference on Parallel and Distributed Processing Techniques and Applications.*, 2012.

[BE81] D.J. Baker and Anthony Ephremides. The architectural organization of a mobile radio network via a distributed algorithm. *Communications, IEEE Transactions on*, 29(11) :1694–1701, 1981.

[Bea03] Jacob Beal. A robust amorphous hierarchy from persistent nodes, 2003.

[BF11] T. Bernard and H. Fouchal. Slot assignment over wireless sensor networks. In *Global Telecommunications Conference (GLOBECOM 2011), 2011 IEEE*, pages 1–5, Dec 2011.

[BIZ89] Judit Bar-Ilan and Dror Zernik. Random leaders and random spanning trees. In Jean-Claude Bermond and Michel Raynal, editors, *Distributed Algorithms*, volume 392 of *Lecture Notes in Computer Science*, pages 1–12. Springer Berlin Heidelberg, 1989.

[BKL01] P. Basu, N. Khan, and T. D C Little. A mobility based metric for clustering in mobile ad hoc networks. In *Distributed Computing Systems Workshop, 2001 International Conference on*, pages 413–418, 2001.

[BKS09] A. Bui, A. Kudireti, and D. Sohier. A fully distributed clustering algorithm based on random walks. In *Parallel and Distributed Computing, 2009. ISPDC '09. Eighth International Symposium on*, pages 125–128, 2009.

[BKS12] Alain Bui, Abdurusul Kudireti, and Devan Sohier. An adaptive random walk-based distributed clustering algorithm. *International Journal of Foundations on Computer Science*, 23(4) :802–830, 2012.

[CDT01] Mainak Chatterjee, Sajal K. Das, and Damla Turgut. Wca : A weighted clustering algorithm for mobile ad hoc networks. *Journal of Cluster Computing (Special Issue on Mobile Ad hoc Networks*, 5 :193–204, 2001.

[DDH72] O. J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, editors. *Structured Programming*. Academic Press Ltd., London, UK, UK, 1972.

[DGL+04] Shlomi Dolev, Seth Gilbert, Nancy A. Lynch, Elad Schiller, Alex A. Shvartsman, and Jennifer L. Welch. Virtual mobile nodes for mobile ad hoc networks. In *in DISC04*, pages 230–244, 2004.

[Dij68] Edsger W. Dijkstra. Letters to the editor : Go to statement considered harmful. *Commun. ACM*, 11(3) :147–148, March 1968.

[Dij74] Edsger W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Commun. ACM*, 17(11) :643–644, November 1974.

[DT09] Shlomi Dolev and Nir Tzachar. Empire of colonies : Self-stabilizing and self-organizing distributed algorithm. *Theoretical Computer Science*, 410 :514 – 532, 2009. Principles of Distributed Systems.

[EWB87] Anthony Ephremides, J.E. Wieselthier, and D.J. Baker. A design concept for reliable mobile radio networks with frequency hopping signaling. *Proceedings of the IEEE*, 75(1) :56–73, 1987.

[GTCT95] Mario Gerla and Jack Tzu-Chieh Tsai. Multicluster, mobile, multimedia radio network. *Wireless Networks*, 1(3) :255–265, 1995.

[IJ90] Amos Israeli and Marc Jalfon. Token management schemes and random walks yield self-stabilizing mutual exclusion. In *Proceedings of the Ninth Annual ACM Symposium on Principles of Distributed Computing*, PODC '90, pages 119–131, New York, NY, USA, 1990. ACM.

[JM10] Colette Johnen and Fouzi Mekhaldi. Robust self-stabilizing construction of bounded size weight-based clusters. In *Proceedings of the 16th international Euro-Par conference on Parallel processing : Part I*, EuroPar'10, pages 535–546, Berlin, Heidelberg, 2010. Springer-Verlag.

[JN06] Colette Johnen and LeHuy Nguyen. Self-stabilizing weight-based clustering algorithm for ad hoc sensor networks. In SotirisE. Nikoletseas and JoséD.P. Rolim, editors, *Algorithmic Aspects of Wireless Sensor Networks*, volume 4240 of *Lecture Notes in Computer Science*, pages 83–94. Springer Berlin Heidelberg, 2006.

[JN09] Colette Johnen and Le Huy Nguyen. Robust self-stabilizing weight-based clustering algorithm. *Theor. Comput. Sci.*, 410(6-7) :581–594, February 2009.

[KR95] AnnaR. Karlin and Prabhakar Raghavan. Random walks and undirected graph connectivity : A survey. In David Aldous, Persi Diaconis, Joel Spencer, and J.Michael Steele, editors, *Discrete Probability and Algorithms*, volume 72 of *The IMA Volumes in Mathematics and its Applications*, pages 95–101. Springer New York, 1995.

[KVCP97] P. Krishna, N. H. Vaidya, M. Chatterjee, and D. K. Pradhan. A cluster-based approach for routing in dynamic networks. *SIGCOMM Comput. Commun. Rev.*, 27(2) :49–64, April 1997.

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7) :558–565, July 1978.

[LC00] Hwa-Chun Lin and Yung-Hua Chu. A clustering technique for large multihop mobile wireless networks. In *Vehicular Technology Conference Proceedings, 2000. VTC 2000-Spring Tokyo. 2000 IEEE 51st*, volume 2, pages 1545–1549 vol.2, 2000.

[LG97] C.R. Lin and M. Gerla. Adaptive clustering for mobile wireless networks. *Selected Areas in Communications, IEEE Journal on*, 15(7) :1265–1275, Sep 1997.

[Lov93] László Lovász. Random walks on graphs : A survey, 1993.

[Lyn96] Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1996.

[MBF04] Nathalie Mitton, Anthony Busson, and Eric Fleury. Self-organization in large scale ad hoc networks. In *Mediterranean ad hoc Networking Workshop (MedHocNet'04).*, page 0000, Bodrum, Turquie, June 2004.

[McQ74] John M McQuillan. Adaptive routing algorithms for distributed computer networks. Technical report, BOLT BERANEK AND NEWMAN INC CAMBRIDGE MASS, 1974.

[MCS10] Jean Frédéric Myoupo, Aboubecrine Ould Cheikhna, and Idrissa Sow. A randomized clustering of anonymous wireless ad hoc networks with an application to the initialization problem. *J. Supercomput.*, 52(2) :135–148, May 2010.

[MFLT05] N. Mitton, E. Fleury, I.G. Lassous, and S. Tixeuil. Self-stabilization in self-organized multihop wireless networks. In *Distributed Computing Systems Workshops, 2005. 25th IEEE International Conference on*, pages 909–915, June 2005.

[NL07] Tina Nolte and Nancy Lynch. Self-stabilization and virtual node layer emulations. In *Proceedings of the 9h international conference on Stabilization, safety, and security of distributed systems*, SSS'07, pages 394–408, Berlin, Heidelberg, 2007. Springer-Verlag.

[Nor98] J.R. Norris. *Markov Chains*. Number 2008 in Cambridge Series in Statistical and Probabilistic Mathematics. Cambridge University Press, 1998.

[Par94] Abhay K Parekh. Selecting routers in ad-hoc wireless networks. *Proceedings SBT/IEEE Intl Telecommunications Symposium*, pages 420–424, 1994.

[Rab09] C. Rabat. Dasor, a Discret Events Simulation Library for Grid and Peer-to-peer Simulators. *Studia Informatica Universalis*, 7(1), 2009.

[SM02] J. Sucec and I. Marsic. Location management handoff overhead in hierarchically organized mobile ad hoc networks. In *Parallel and Distributed Processing Symposium., Proceedings International, IPDPS 2002, Abstracts and CD-ROM*, pages 10 pp–, 2002.

[SSS06] Sulyun Sung, Yuhwa Seo, and Yongtae Shin. Hierarchical clustering algorithm based on mobility in mobile ad hoc networks. In Marina Gavrilova, Osvaldo Gervasi, Vipin Kumar, C.J.Kenneth Tan, David Taniar, Antonio Laganá, Youngsong Mun, and

Hyunseung Choo, editors, *Computational Science and Its Applications - ICCSA 2006*, volume 3982 of *Lecture Notes in Computer Science*, pages 954–963. Springer Berlin Heidelberg, 2006.

[Tel94] Gerard Tel. *Introduction to Distributed Algorithms*. Cambridge University Press, New York, NY, USA, 1994.

[TR98] D.G. Thaler and C.V. Ravishankar. Distributed top-down hierarchy construction. In *INFOCOM '98. Seventeenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, volume 2, pages 693–701 vol.2, 1998.

[TW91] Prasad Tetali and Peter Winkler. On a random walk problem arising in self-stabilizing token management. In *Proceedings of the Tenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '91, pages 273–280, New York, NY, USA, 1991. ACM.

[YC03] J.Y. Yu and P.H.J. Chong. 3hbac (3-hop between adjacent clusterheads) : a novel non-overlapping clustering algorithm for mobile ad hoc networks. In *Communications, Computers and signal Processing, 2003. PACRIM. 2003 IEEE Pacific Rim Conference on*, volume 1, pages 318–321 vol.1, Aug 2003.

[YC09] Shin-Jer Yang and Hao-Cyun Chou. Design issues and performance analysis of location-aided hierarchical cluster routing on the manet. In *Communications and Mobile Computing, 2009. CMC '09. WRI International Conference on*, volume 2, pages 26–31, 2009.