Special issue CARI'12

Vers une structuration auto-stabilisante des réseaux *Ad Hoc*

Mandicou Ba* — Olivier Flauzac* — Bachar Salim Haggar*
Rafik Makhloufi‡ — Florent Nolot* — Ibrahima Niang†

* Université de Reims Champagne-Ardenne
Laboratoire CReSTIC - Équipe SysCom EA 3804
{mandicou.ba, olivier.flauzac, bachar-salim.haggar, florent.nolot}@univ-reims.fr

† Université Cheikh Anta Diop
Département de Mathématiques et Informatique
Laboratoire d'Informatique de Dakar (LID)
iniang@ucad.sn

‡ École Nationale des Ponts et Chaussées
Laboratoire CERMICS
Laboratoire CERMICS - Equipe SOWG
makhlour@cermics.enpc.fr

RÉSUMÉ. Dans cet article, nous proposons un algorithme de structuration auto-stabilisant, distribué et asynchrone qui construit des *clusters* de diamètre au plus $2k$. Notre approche ne nécessite aucune initialisation. Elle se fonde uniquement sur l'information provenant des nœuds voisins à l'aide d'échanges de messages. Partant d'une configuration quelconque, le réseau converge vers un état stable après un nombre fini d'étapes. Nous montrons par preuve formelle que pour un réseau de n nœuds, la stabilisation est atteinte en au plus $n + 2$ transitions. De plus, l'algorithme nécessite une occupation mémoire de $(\Delta_u + 1) * \log(2n + k + 3)$ bits pour chaque nœud u où Δ_u représente le degré (nombre de voisins) de u et k la distance maximale dans les *clusters*. Afin de consolider les résultats théoriques obtenus, nous avons effectué une campagne de simulation sous OMNeT++ pour évaluer la performance de notre solution.

ABSTRACT. In this paper, we present a self-stabilizing asynchronous distributed clustering algorithm that builds non-overlapping k-hops clusters. Our approach does not require any initialization. It is based only on information from neighboring nodes with periodic messages exchange. Starting from an arbitrary configuration, the network converges to a stable state after a finite number of steps. Firstly, we prove that the stabilization is reached after at most $n+2$ transitions and requires $(\Delta_u+1)*\log(2n+k+3)$ bits per node, where Δ_u represents node's degree, n is the number of network nodes and k represents the maximum hops number. Secondly, using OMNet++ simulator, we performed an evaluation of our proposed algorithm.

MOTS-CLÉS : Réseaux *Ad Hoc*, *clustering*, algorithmes distribués, auto-stabilisation, OMNeT++.

KEYWORDS : Ad hoc networks, clustering, distributed algorithms, self-stabilizing, OMNeT++.

1. Introduction

Dans les réseaux *Ad Hoc*, la solution de communication la plus utilisée est la diffusion. C'est une technique simple qui nécessite peu de calcul. Cependant, elle est coûteuse en termes d'échanges de messages et peut entraîner une saturation du réseau. Afin d'optimiser les communications, une solution efficace consiste en la structuration du réseau en *arbres* [9] ou en *clusters* [15].

Plusieurs travaux comme [20, 32, 33] ont montré que le *clustering* offre une structuration efficace du réseau permettant ainsi d'optimiser les communications. Le *clustering* consiste à découper le réseau en groupes de nœuds appelés *clusters* donnant ainsi au réseau une structure hiérarchique [22]. Chaque *cluster* est représenté par un nœud particulier appelé *cluster-head*. Un nœud est élu *cluster-head* selon une métrique telle que le degré, la mobilité, l'identité des nœuds, la densité, etc. ou une combinaison de ces paramètres.

En outre, les réseaux *Ad Hoc* peuvent être également dynamiques. Un nœud peut se connecter ou se déconnecter du réseau à tout instant et peut donc induire des fautes transitoires par modification de la topologie du réseau. Donc, concevoir des solutions de *clustering* tolérantes aux fautes transitoires est plus que nécessaire pour des applications de réseaux *Ad Hoc* souvent critiques. Dans [21], Johnen et *al.* ont montré que l'auto-stabilisation, introduite par Dijkstra en 1974 dans [17], est une propriété intéressante de tolérance aux fautes transitoires dans un système distribué comme les réseaux Ad Hoc. Dijkstra considère qu'un système est auto-stabilisant si quelle que soit sa configuration de départ, il est garanti d'arriver à une configuration légale en un nombre fini d'étapes. Suite à l'apparition d'une faute transitoire, le système de lui même, retrouve une configuration légale après un temps fini. Donc, les algorithmes de *clustering* auto-stabilisants sont une solution efficace pour structurer le réseau dans le but d'optimiser les communications et de résister aux fautes transitoires qui peuvent survenir.

Plusieurs solutions de *clustering* auto-stabilisantes ont été proposées dans la littérature [13, 16, 19, 23, 24, 25, 26, 27, 28]. Elles sont classées en algorithmes à *1 saut* ou à *k sauts*. Dans les solutions à *1 saut* [19, 23, 24, 25, 28], les nœuds sont à distance *1* du *cluster-head* et le diamètre maximal des *clusters* est de *2*. Par contre, dans les solutions à *k sauts* [13, 16, 27, 26], les nœuds peuvent se situer jusqu'à une distance *k* du *cluster-head* et le diamètre maximal des *clusters* est donc au plus de *2k*. En outre, les solutions existantes utilisent soit un modèle à mémoire partagée (modèle à états ou modèle à registres) [13, 16, 23, 24] soit un modèle à passage de messages [13, 16, 26, 27]. Les solutions qui utilisent un modèle à états, comme elles ne se focalisent pas sur les communications (envoi et réception de messages), elles ne sont donc pas réalistes dans le cadre des réseaux *Ad Hoc*. Pour les solutions fondées sur un modèle à passage de messages, bien qu'étant plus réalistes dans le contexte des réseau *Ad Hoc*, elles sont souvent coûteuses en termes d'échanges de messages. Or, plusieurs études, comme Pottie et Kaiser dans [29], ont montré que les communications représentent la principale source de consommation de ressources énergétiques.

Ainsi, dans le but de réduire les communications et de résister aux fautes transitoires, nous proposons, dans cet article, un algorithme de *clustering* à *k sauts* qui est complètement distribué et auto-stabilisant. Notre solution utilise un modèle asynchrone à passage de messages et construit des *clusters* non-recouvrants de diamètre au plus $2k$. Elle ne nécessite pas d'initialisation et est tolérante aux fautes transitoires. De plus, nous utilisons le

critère de l'identité maximale des nœuds pour l'élection du *cluster-head* qui apporte une meilleure de stabilité. Pour les communications, nous considérons seulement un échange de messages avec le voisinage à distance à distance 1 (vue locale) pour construire des *clusters* à *k sauts*.

La suite de l'article est organisée comme suit. Dans la Section 2, nous étudions les solutions de *clustering* existantes. La Section 3 présente notre contribution. A la Section 4, nous décrivons le modèle sur lequel se fonde notre approche. Puis, à la section 5, nous donnons le principe d'exécution et les détails de notre algorithme. Dans la Section 6, nous montrons le schéma de la preuve de convergence, de clôture et de l'occupation mémoire de notre algorithme. Nous présentons une campagne de simulation sous OMNeT++ dans la Section 7 pour évaluer les performances moyennes de notre solution. Une conclusion et des perspectives sont données dans la Section 8.

2. État de l'art

Plusieurs propositions de *clustering* ont été faites dans la littérature [13, 16, 19, 23, 24, 25, 26, 27, 28].

Les approches auto-stabilisantes [25, 19, 24, 23, 28] construisent des *clusters* à *1 saut*. Mitton et *al.* [28] utilisent comme métrique la *densité*, dans un modèle asynchrone à passage de messages, afin de minimiser la reconstruction de la structure en cas de faible changement de topologie. Chaque nœud calcule sa densité et la diffuse à ses voisins situés à distance *1*. Le nœud avec la plus grande densité dans son voisinage devient *cluster-head*. Dans cette approche, un nœud avec une forte mobilité mobile ne s'attache à aucun *cluster*. De plus, du fait du calcul et de la diffusion de la densité, cette approche engendre d'importantes communications. Flauzac et *al.* ont proposé dans [19] un algorithme auto-stabilisant, dans un modèle asynchrone à passage de messages, qui combine la découverte de topologie et de *clustering* en une seule phase. Elle ne nécessite qu'un seul type de message échangé entre voisins. Un nœud devient *cluster-head* s'il possède la plus grande identité parmi tous ses voisins.

Dans [10, 11], Bui et *al.* ont proposé un algorithme de *clustering* adaptatif aux changements de topologie mais non auto-stabilisant et non déterministe. Ils utilisent une marche aléatoire pour construire d'abord un cœur de *cluster* composé de 2 à $MaxCoreSize$ nœuds. Ce cœur est ensuite étendu aux voisins immédiats, appelés nœuds ordinaires, pour former les *clusters*. Les approches de Bui et *al.* [10, 11], comme elles utilisent la marche aléatoire les communications, sont coûteuses en termes de messages échangés.

Johnen et *al.* ont proposé dans [24] un algorithme auto-stabilisant qui se fonde sur un modèle à états, qui construit des *clusters* de taille fixe. Les auteurs attribuent un poids à chaque nœud et fixent un paramètre $SizeBound$ qui représente le nombre maximal de nœuds dans un *cluster*. Un nœud ayant le poids le plus élevé devient *cluster-head* et collecte dans son *cluster* jusqu'à $SizeBound$ nœuds. Dans [23], Johnen et *al.* ont étendu leur proposition décrite dans [24] pour apporter la notion de robustesse. La robustesse est une propriété qui assure qu'en partant d'une configuration quelconque, le réseau est partitionné après un round asynchrone. Durant la phase de convergence, le réseau demeure toujours partitionné et vérifie un prédicat de sûreté. Dans [25], Kuroiwa et *al.* améliorent l'approche proposée par Johnen et *al.* dans [24]. Kuroiwa et *al.* proposent une méthode fondée sur un modèle à états synchrone afin d'apporter une meilleure stabilité dans la

structure des *clusters* en comparaison avec la solution décrite [24]. Les solutions proposées dans [24, 24, 25], utilisent un modèle à états, ne sont pas réalistes dans le cadre des réseaux *Ad Hoc*.

Les approches auto-stabilisantes [27, 26, 13, 16] construisent des *clusters* à *k sauts*.

Dans [27], Miton et *al.* ont étendu leurs travaux décrits dans [28] pour proposer un algorithme robuste de *clustering* auto-stabilisant à *k sauts*. Miton et *al.* utilisent un modèle asynchrone à passage de messages. Chaque nœud du réseau calcule sa *k-density* en utilisant les information de son voisinage à distance $k+1$ sauts ($\{k+1\}$-Neighborhood view). Puis, la *k-density* de chaque nœud est retransmise à distance k. Le nœuds avec la plus grande valeur de *k-density* devient *cluster-head*. Dans cet algorithme, un nœud trop mobile ne s'attache à aucun *cluster*. De plus, du fait du calcul et de la diffusion de la *k-density* jusqu'à une distance k, cette solution est coûteuse en termes d'échanges de messages.

Larsson et Tsigas ont proposé dans [26] un algorithme distribué, auto-stabilisant avec multiple chemins nommé (x,k)-clustering. Cet algorithme assigne, si possible, x *cluster-heads* dans une distance de k sauts à tout nœud du réseau. Les auteurs utilisent un modèle synchrone à passage de messages et procèdent par rounds. Ils ont montré que leur algorithme attribue, si possible, x *cluster-heads* à chaque nœud en $O(k)$ rounds. Puis, l'ensemble des *cluster-heads* se stabilise, avec une forte probabilité, en un minimum local de $O(g * k * \log(n))$ rounds. De plus, il nécessite $O(|G_u^k| * (\log(n) + \log(k)))$ bits pour chaque nœuds u du réseau, où n représente la taille du réseau, k le nombre de maximal de sauts et g une borne supérieure du nombre de nœuds dans un *cluster*. Cette approche génère beaucoup de messages. En effet, les messages de chaque nœuds sont retransmis jusqu'à une distance k.

Caron *et al.* [13], utilisant comme métrique un unique identifiant pour chaque nœud avec un poids attribué à chaque arrête du graphe, ont proposé un algorithme nommé k-*clustering* et fondé sur un modèle à états. Un k-*clustering* dans un graphe consiste à partitionner le réseau en *clusters* disjoints dans lesquels chaque nœud se situe au plus à distance k du *cluster-head*. Cette solution s'inspire partiellement dans l'algorithme de Amis et *al.* proposé dans [3]. Dans cette solution chaque nœud peut lire les états de ses voisins situés jusqu'à une distance $k+1$ et est le seul à disposer des privilèges d'écriture sur ses propres états. Les auteurs ont montré que cet algorithme s'exécute en $O(n * k)$ rounds et nécessite $O(log(n) + log(k))$ bits par nœud, où n est le nombre de nœuds du réseau.

Dans [16], utilisant le critère de l'identité minimale, Datta et *al.* ont proposé un algorithme, nommé *MINIMAL*, de *clustering* auto-stabilisant à k sauts. *MINIMAL* utilise un modèle à états et considère un graphe G quelconque de n nœud avec un identifiant unique attribué à chacun. *MINIMAL* construit d'abord un ensemble D qui représente un k-*dominating*. D est défini tel que tout nœud n'appartenant pas à D se situe au plus à distance k d'au moins d'un membre de D. Datta et *al.* ont prouvé que, pour construire D, *MINIMAL* requiert $O(n)$ rounds. Puis, considérant D comme l'ensemble de *cluster-heads*, *MINIMAL* structure G en *clusters* de rayon k et ceci nécessite $O(n^2)$ rounds. De plus, il requiert $\log(n)$ bits pour tout nœud de G.

3. Contribution

Les solutions de clustering auto-stabilisantes à k qui se fondent sur un modèle à états [13, 16, 24, 23] nécessitent un temps de stabilisant élevé. De plus, comme elles ne se focalisent pas sur les envois et réceptions de messages, elles ne sont pas réalistes dans le cadre des réseaux *Ad Hoc*. Quant aux solutions qui utilisent un modèle à passage de messages [25, 24, 23, 28], bien qu'elles sont plus réalistes dans le contexte des réseaux *Ad Hoc*, elles sont couteuses en termes d'échanges de messages. Or, dans les réseaux *Ad Hoc*, les communications représentent la majeure source de consommation de ressources [29].

Ainsi, nous proposons une approche de structuration qui utilise un modèle asynchrone à passage de messages contrairement aux solutions proposées dans [24, 23, 13, 16, 25]. Notre solution est distribuée et auto-stabilisante. Nous utilisons le critère d'identité maximale qui apporte plus de stabilité par rapport aux métriques variables utilisées dans [27, 28, 24, 23]. Notre algorithme structure le réseau en *clusters* disjoints deux à deux et de diamètre au plus égal à $2k$. Cette structuration ne nécessite aucune initialisation. Nous combinons la découverte de voisinage et le *clustering* en une seule phase. Nous utilisons uniquement les informations provenant des nœuds voisins situés à distance 1, par le biais d'échanges de messages, pour construire des *clusters* à *k sauts*.

D'une part, par une preuve formelle, nous montrons que pour un réseau de n nœuds, la stabilisation est atteinte dans le pire des cas en $n + 2$ transitions. De plus, il nécessite une occupation mémoire de $(\Delta_u + 1) * \log(2n + k + 3)$ bits pour chaque nœud u où Δ_u représente le degré - nombre de voisins - de u, n le nombre total de nœuds et k la distance maximale dans les *clusters*. D'autre part, par une campagne de simulation sous OMNeT++, nous évaluons les performances moyennes de notre solution. Le résultats de simulation montrent qu'en moyenne, nous obtenons des temps de stabilisation très inférieurs à celui du pire cas. De plus, suite à l'occurrence de fautes transitoires, le coût de la restructuration et le nombre de nœuds impliqués dans cette opération sont moindres.

4. Modélisation

Nous considérons notre réseau comme un système distribué que nous modélisons par un graphe connexe et non orienté $G = (V, E)$ [4, 30]. V désigne l'ensemble des nœuds du réseau avec $|V| = n$ et $n \geq 2$. E représente l'ensemble des connexions existantes entre les nœuds. Nous supposons que chaque nœud $u \in V$ est caractérisé par une « valeur d'identification » qui lui est propre ; son *identité* que nous notons id_u et telle que $0 \leq id_u \leq n - 1$. Cette identité est tirée d'un ensemble $\mathcal{I}$ non vide fini ou infini mais obligatoirement muni d'un *ordre total* qui peut être, par exemple, l'ordre numérique sur $\mathbb{N}$. Soient u et v deux nœuds appartenant à V. Une arête (u, v) existe si et seulement si u peut communiquer avec v et vice-versa. Ce qui implique que tous les liens sont bidirectionnels. Dans ce cas, les nœuds u et v sont voisins. L'ensemble des nœuds $v \in V$ voisins du nœud u situés à distance 1 est noté N_u. On définit la distance $d_{(u,v)}$ entre deux nœuds u et v quelconque dans le graphe G comme le nombre d'arêtes minimal le long du chemin entre u et v.

Nous utilisons un modèle de communication *asynchrone à passage de messages*. Ce choix est motivé par la validité de ce modèle ; validité due à sa similarité réalité (les réseaux *Ad Hoc*) comparé au modèle à états. Ainsi, chaque paire de nœuds (u, v) du réseau

est connectée par un canal de communication bidirectionnel. Les canaux de communication sont asynchrones ; le temps de transite des messages est fini et non borné. De plus, nous considérons ces canaux de communication comme fiables ; pas de perte ni de corruption de messages. Chaque nœud u envoie périodiquement à ses voisins $v \in N_u$ un message contenant les informations sur son état actuel. Nous supposons qu'un message envoyé est correctement reçu au bout d'un temps fini mais non borné par tous les voisins situés à distance 1. De plus, chaque nœud u du réseau maintient dans une table de voisinage les états actuels de ses voisins. La réception d'au moins d'un message d'un voisin v déclenche l'exécution de notre algorithme de *clustering*.

5. Structuration auto-stabilisante dans les réseaux Ad Hoc

5.1. Préliminaires

Dans cette section, nous donnons les différents concepts et définitions utilisés dans notre approche.

Définition 5.1 *(Cluster)*

Nous définissons un cluster à k sauts comme un sous graphe connexe du réseau, dont le diamètre est inférieur ou égal à $2k$. L'ensemble des nœuds d'un cluster u est noté V_u.

Définition 5.2 *(Identifiant du cluster)*

Chaque cluster possède un unique identifiant correspondant à la plus grande identité de tous les nœuds du cluster. L'identité d'un cluster auquel appartient le nœud u est notée cl_u.

Dans nos *clusters* (cf. Figure 1(a)), chaque nœud u possède un statut noté $statut_u$ (cf. Figure 1(b)) qui définit son rôle à jouer. Ainsi, un nœud peut être soit *Cluster-Head* (CH), soit *Simple Node* (SN), ou soit *Gateway Node* (GN). Le *cluster-head* est le nœud possédant la plus grande identité dans le *cluster*. Il a pour rôle de manager les communications au sein du *cluster*. Les nœuds *gateway nodes* assurent l'interconnexion entre *clusters* voisins et permettent les communications extra-*clusters*. Un nœud qui n'est ni *cluster-head* ni *gateway node* est alors qualifié de *simple node* d'un *cluster*. De plus, chaque nœud u adopte un voisin $v \in N_u$, noté gn_u, par lequel il passe pour atteindre son CH.

Définition 5.3 *(Statut des nœuds)*

• *Cluster-Head (**CH**) : un nœud u a le statut de CH s'il possède le plus grand identifiant parmi tous les nœuds de son cluster :*

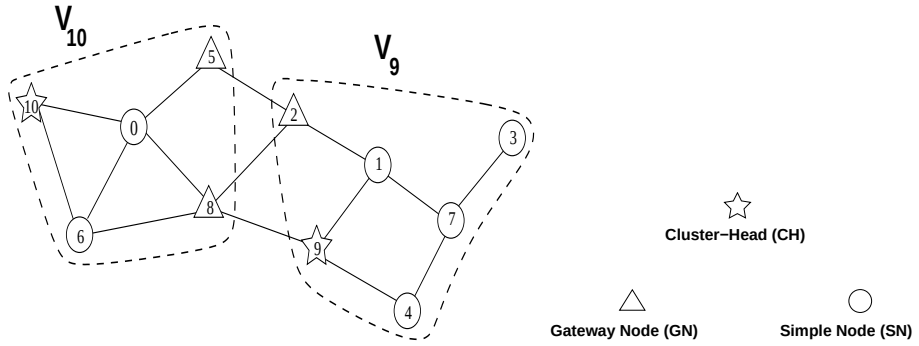
- $statut_u = CH \iff \forall v \in V_{cl_u}, (id_u > id_v) \wedge (dist_{(u,v)} \leq k)$.

• *Simple Node (**SN**) : un nœud u a le statut de SN si tous ses voisins appartiennent au même cluster que lui :*

- $statut_u = SN \iff (\forall v \in N_u, cl_v = cl_u) \wedge (\exists w \in V / (statut_w = CH) \wedge (dist_{(u,w)} \leq k))$.

• *Gateway Node (**GN**) : un nœud u a le statut de GN s'il existe un nœud v dans son voisinage appartenant un autre cluster :*

- $statut_u = GN \iff \exists v \in N_u, (cl_v \neq cl_u)$.



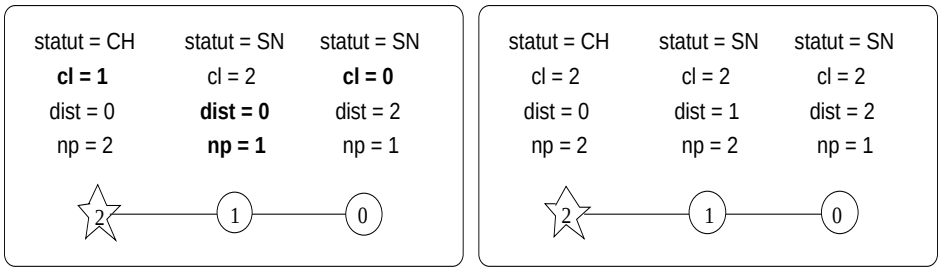
(a) Structuration en *clusters* (b) Statut des nœuds

Figure 1 – Structure des *clusters* et statuts des nœuds

Définition 5.4 *(Nœud cohérent)*

Un nœud u est cohérent (cf. Figure 2(b)) si et seulement si, il est dans l'un des états suivants :

- *Si* $statut_u = CH$ *alors* $(cl_u = id_u) \wedge (dist_{(u,CH_u)} = 0) \wedge (gn_u = id_u)$.
- *Si* $statut_u \in \{SN, GN\}$ *alors* $(cl_u \neq id_u) \wedge (dist_{(u,CH_u)} \neq 0) \wedge (gn_u \neq id_u)$.



(a) Nœuds incohérents (b) Nœuds cohérents

Figure 2 – Notion de cohérence des nœuds

Définition 5.5 *(Nœud stable)*

Un nœud u est dans un état stable si et seulement si, il est cohérent et satisfait l'une des conditions suivantes :

- *Si* $statut_u = CH$ *alors* $\{\forall v \in N_u, statut_v \neq CH\} \wedge \{\forall v \in N_u$ *tel que* $cl_v = cl_u$ *alors* $(id_v < id_u)\} \wedge \{\forall v \in N_u$ *tel que* $id_v > id_u$ *alors* $(cl_v \neq cl_u) \wedge (dist_{(v,CH_v)} = k)\}$.
- *Si* $statut_u = SN$ *alors* $\{\forall v \in N_u, (cl_v = cl_u) \wedge (dist_{(u,CH_u)} \leq k)\} \wedge \{\exists v \in N_u, (gn_v = id_u) \wedge (dist_{(v,CH_v)} = dist_{(u,CH_u)} + 1)\}$.
- *Si* $statut_u = GN$ *alors* $\exists v \in N_u, (cl_v \neq cl_u) \wedge \{(dist_{(u,CH_u)} = k) \vee (dist_{(v,CH_v)} = k)\}$.

Définition 5.6 ***(Réseau stable)***

Le réseau est stable si et seulement si tous les nœuds sont stables (cf. Figure 3).

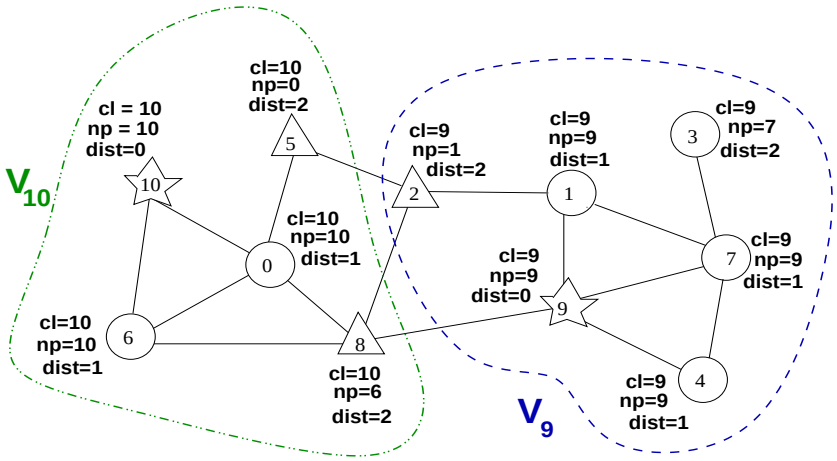


Figure 3 – Réseau stable

5.2. Principe d'exécution

Notre algorithme est auto-stabilisant, il ne nécessite ainsi aucune initialisation. Partant d'une configuration quelconque, avec seulement l'échange d'un seul type de message, les nœuds s'auto-organisent en *clusters* non-recouvrants au bout d'un nombre fini d'étapes. Ce message, dit *hello*, est échangé entre chaque pair de nœuds voisins. Il contient les quatre informations suivantes : l'identité du nœud u expéditeur (id_u), son identifiant de *cluster* (cl_u), son statut ($statut_u$) et la distance ($dist_{(u,CH_u)}$) qui le sépare de son *cluster-head*. Rappelons que l'identifiant du *cluster* d'un nœud est égal à l'identifiant de son *cluster-head* (cf. Définition 5.2). Donc, la structure des messages *hello* est la suivante : $hello(id_u, cl_u, statut_u, dist_{(u,CH_u)})$. De plus, chaque nœud maintient une table de voisinage, notée $StateNeigh_u$, contenant l'ensemble des états de ses voisins. $StateNeigh_u[v]$ contient les états du nœud v voisin du nœud u

La solution que nous proposons se déroule de la façon suivante :

Dès la réception d'un message *hello*, un nœud u exécute l'algorithme 1. Durant cet algorithme, u exécute trois étapes consécutives. D'abord, il effectue la mise à jour de sa table de voisinage, ensuite il vérifie la cohérence de ses variables locales et en dernier il procède la phase de *clustering*. A la fin de cette exécution, u envoie un message hello à touts ses voisins.

Après la phase de mise à jour, chaque nœud vérifie sa cohérence. Comme le *cluster-head* est le nœud avec la plus grande identité au sein du *cluster*, si un nœud a le statut de CH alors son identifiant de *cluster* doit obligatoirement être égal à son propre identifiant. Dans la figure 2(a), le nœud d'identité 2 est *cluster-head*, son identifiant de *cluster* est égal à 1 donc le nœud 2 n'est pas cohérent. Tout comme les nœuds 1 et 0, ils ne vérifient pas la définition 5.4. Chaque nœud détecte et corrige son incohérence durant la phase de vérification de cohérence telle que définit dans l'algorithme 1. La figure 2(b) montre les nœuds dans des états cohérents.

Durant l'étape de *clustering*, chaque nœud compare son identité à celle de touts ses voisins. Un nœud u s'élit *cluster-head* s'il la plus grand identité parmi tous les nœuds de son *cluster*. Si nœud u découvre un voisin v avec une plus grande identité que la sienne, alors il devient membre du même *cluster* que v avec un statut de SN. Si un nœud reçoit

un message provenant d'un voisin membre d'un autre *cluster*, alors il devient un nœud de passage avec un statut de GN. Comme les messages *hello* contiennent la distance entre chaque nœud u et son *cluster-head*, alors u peut déterminer si le diamètre maximal du *cluster* est atteint. Ainsi u pourra donc choisir un autre *cluster* si les k *sauts* sont atteints.

La figure 4 illustre le passage d'une configuration γ_i à γ_{i+1}. Dans cet exemple, $k = 2$. A γ_i, chaque nœud envoie à ses voisins un message *hello*. γ_{i+1} est une configuration stable. Dès la réception de messages venant des voisins, chaque nœud met à jour sa table de voisinage puis exécute l'algorithme 1. Dans cet exemple, le nœud n_1 qui est un nœud membre du *cluster* de n_3 détecte le nœud n_0 comme un *cluster* voisin. Il devient nœud de passage avec un statut de GN et envoie un message $hello$ à ses voisins pour une mise à jour. Les nœuds n_3, n_2 et n_0, en fonction de leurs états actuels ainsi que ceux de leurs voisins, ne changent pas d'état. γ_{i+i} correspond à un état stable.

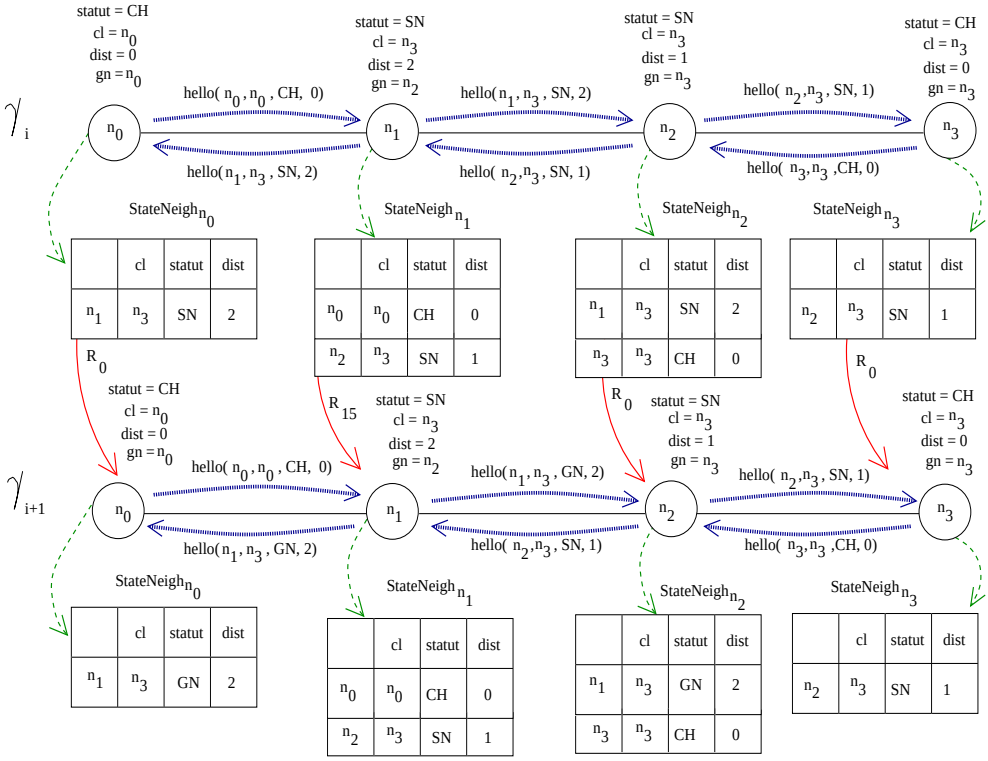


Figure 4 – Passage d'une configuration γ_i à γ_{i+1}

5.3. Algorithme auto-stabilisant de clustering à k sauts

Chaque nœud u du réseau connaît le paramètre k avec $k < n$ qui représente le nombre maximal de saut dans un *cluster*. De plus, tout nœud u possède les variables locales et *macros* qui sont définis dans le tableau 1. L'exécution de l'algorithme 1 est déclenchée par la réception d'au moins d'un message venant d'au moins d'un voisin v. Suite à cette exécution, chaque nœud u envoie un message à ses voisins pour les informer de son changement d'état.

Algorithme 1: Clustering auto-stabilisant à *k sauts* dans les réseaux Ad Hoc

/* Dès la réception d'un message hello d'un voisin */

Prédicats

$P_1(u) \equiv (statut_u = CH)$
$P_2(u) \equiv (statut_u = SN)$
$P_3(u) \equiv (statut_u = GN)$
$P_{10}(u) \equiv (cl_u \neq id_u) \vee (dist_{(u,CH_u)} \neq 0) \vee (gn_u \neq id_u)$
$P_{20}(u) \equiv (cl_u = id_u) \vee (dist_{(u,CH_u)} = 0) \vee (gn_u = id_u)$
$P_{40}(u) \equiv \forall v \in N_u, (id_u > id_v) \wedge (id_u \geq cl_v) \wedge (dist_{(u,v)} \leq k)$
$P_{41}(u) \equiv \exists v \in N_u, (statut_v = CH) \wedge (cl_v > cl_u)$
$P_{42}(u) \equiv \exists v \in N_u, (cl_v > cl_u) \wedge (dist_{(v,CH_v)} < k)$
$P_{43}(u) \equiv \forall v \in N_u / (cl_v > cl_u), (dist_{(v,CH_v)} = k)$
$P_{44}(u) \equiv \exists v \in N_u, (cl_v \neq cl_u) \wedge \{(dist_{(u,CH_u)} = k) \vee (dist_{(v,CH_v)} = k)\}$

Règles de clustering

/* Mise à jour du voisinage */
$StateNeigh_u[v] := (id_v, cl_v, statut_v, dist_{(v,CH_v)})$;

/* Cluster-1: Gestion de la cohérence */
$R_{10}(u) : P_1(u) \wedge P_{10}(u) \longrightarrow cl_u := id_u; gn_u = id_u; dist_{(u,CH_u)} = 0;$
$R_{20}(u) : \{P_2(u) \vee P_3(u)\} \wedge P_{20}(u) \longrightarrow$
$statut_u := CH; cl_u := id_u; gn_u = id_u; dist_{(u,CH_u)} = 0;$

/* Cluster-2: Clustering */
$R_{11}(u) : : \neg P_1(u) \wedge P_{40}(u) \longrightarrow$
$statut_u := CH; cl_u := id_v; dist_{(u,CH_u)} := 0; gn_u := id_u;$
$R_{12}(u) : : \neg P_1(u) \wedge P_{41}(u) \longrightarrow$
$statut_u := SN; cl_u := id_v; dist_{(u,v)} := 1; gn_u := NeighCH_u;$
$R_{13}(u) : : \neg P_1(u) \wedge P_{42}(u) \longrightarrow$
$statut_u := SN; cl_u := cl_v; dist_{(u,CH_u)} := dist_{(v,CH_v)} + 1; gn_u := NeighMax_u;$
$R_{14}(u) : : \neg P_1(u) \wedge P_{43}(u) \longrightarrow$
$statut_u := CH; cl_u := id_v; dist_{(u,CH_u)} := 0; gn_u := id_u;$
$R_{15}(u) : : P_2(u) \wedge P_{44}(u) \longrightarrow statut_u := GN;$
$R_{16}(u) : : P_1(u) \wedge P_{41}(u) \longrightarrow$
$statut_u := SN; cl_v := id_v; dist_{(u,v)} := 1; gn_u := NeighCH_u;$
$R_{17}(u) : : P_1(u) \wedge P_{42}(u) \longrightarrow$
$statut_u := SN; cl_u := cl_v; dist_{(u,CH_u)} := dist_{(v,CH_v)} + 1; gn_u := NeighMax_u;$

/* Envoi d'un message hello */
$R_0(u) : hello(id_u, cl_u, statut_u, dist_{(u,CH_u)});$

6. Schéma intuitif de la preuve de la stabilisation

Dans cette section, nous énonçons les principaux théorèmes et propriétés vérifiés par notre algorithme. Nous ne donnons pas tous les détails des preuves. Nous avons publié la preuve au complète dans [5].

Définition des variables et constantes
Paramètres : $\|V\| = n$: *nombre total de nœuds dans le réseau.* k : *nombre maximal de sauts dans un cluster.*
Variables locales du nœud u : id_u : *identifiant du nœud* u. N_u : *ensemble des nœuds voisins de* u. $StateNeigh_u$: *table de voisinage contenant l'ensemble des états des voisins de* u. $StateNeigh_u[v]$ *contient les états du nœud* v *voisin de* u. cl_u : *identifiant du cluster-head du nœud* u. $statut_u \in \{CH, SN, GN\}$: *statut du nœud* u. $dist_{(u,CH_u)}$: *distance entre un nœud* u *et son cluster-head* CH_u. np_u : *identifiant du voisin du nœud* u *lui permettant d'atteindre son cluster-head.*
Macros : $NeighCH_u = \{id_v \mid v \in N_u \wedge statut_v = CH \wedge cl_u = cl_v\}$. $NeighMax_u = (Max\{id_v \mid v \in N_u \wedge statut_v \neq CH \wedge cl_u = cl_v\}) \wedge (dist_{(v,CH_u)} = Min\{dist_{(x,CH_u)}, x \in N_u \wedge cl_x = cl_v\})$.

Tableau 1 – Définition des variables et constantes de notre algorithme

6.1. Convergence et clôture

Dans cette section, nous montrons les propriétés de convergence et de clôture de notre solution.

L'état d'un nœud est définit par la valeur de ses variables locales ainsi que celles de des ses voisins connues. Une *configuration* γ_i du réseau est une instance de l'état de tous les nœuds. Avec notre approche, les nœuds possédant les identités les plus grandes se fixent en premier. Un nœud u est dit *fixé* à partir de la configuration γ_i si le contenu de sa variable cl_u ne change plus. L'ensemble des nœuds fixés à γ_i est noté $\mathcal{F}_i$. Une *transition* τ_i est le passage d'une configuration γ_i à γ_{i+1}. Au cours d'une transition, chaque nœud a reçu un message d'au moins d'un voisin et a exécuté l'Algorithme 1. Ainsi, avec notre approche le nombre de nœuds fixés croit strictement à chaque configuration et tend vers n, n étant le nombre de nœuds du réseau.

Lemme 6.1 *Soit* γ_0 *une configuration quelconque. A* γ_1, $\forall u \in V$, u *est cohérent.*

Preuve.

A γ_0 quelque soit son état, chaque nœud vérifie et corrige sa cohérence par exécution de la règle R_{10} ou R_{20} durant la transitions τ_0. Ainsi, à γ_1 tout nœud est cohérent.

Corollaire 6.1 $|\mathcal{F}_1| \geq 1$.

Preuve.

Comme tous les nœuds sont cohérents d'après le lemme 6.1. Et $\exists\ u$ tel que $\forall\ v \in V$, $id_u > id_v$. Au moins u applique R_{11} durant τ_0 et est donc fixé à γ_1. D'où $u \in F_1$ et $|\mathcal{F}_1| \geq 1 \Rightarrow |\mathcal{F}_1| > |\mathcal{F}_0|$. Ce nœud u est noté CH_{Max}.

Théorème 6.1 $\forall\ i < k+1, |\mathcal{F}_{i+1}| > |\mathcal{F}_i|$ *et* $\mathcal{F}_i \subset \mathcal{F}_{i+1}$.

Preuve.

D'après le corollaire 6.1, $|\mathcal{F}_1| > |\mathcal{F}_0|$. Pour $i = 0$, le résultat est vrai. A γ_2, nous pouvons constater que les nœuds situés à distance 1 du CH_{Max} sont fixés soit par la règle R_{12} soit par la règle R_{16} selon que leur statut est SN ou CH. Donc, $|\mathcal{F}_2| > |\mathcal{F}_1|$. Nous prouvons ensuite par récurrence qu'à γ_i, les nœuds situés à distance $(i-1)$ de CH_{Max} se fixent par la règle R_{13} ou par la règle R_{17}. Pour $i = k$, nous obtenons par récurrence $|\mathcal{F}_{k+1}| > |\mathcal{F}_k|$.

Théorème 6.2 ***(Convergence)***
Partant d'une configuration quelconque, une configuration stable est atteinte au plus en $n + 2$ *transitions.*

Preuve.

Comme $|\mathcal{F}_1| \geq 1$, nous avons $|\mathcal{F}_{k+1}| > k$. En réitérant le processus à partir d'un nouveau CH_{Max} qui est le nœud d'identité maximale $\notin \mathcal{F}_{k+1}$, nous prouvons que $\forall\ i < n$, $|\mathcal{F}_{i+1}| > |\mathcal{F}_i|$ et $\mathcal{F}_i \subset \mathcal{F}_{i+1}$. Pour $i = n$, nous avons $|\mathcal{F}_{n+1}| > |\mathcal{F}_n|$ d'où $|\mathcal{F}_{n+1}| = n$. Il faut ensuite une transition de plus pour que l'état des nœuds ne change plus. Nous obtenons un temps de stabilisation d'au plus $n + 2$ transitions.

Théorème 6.3 ***(Clôture)***
A partir d'une configuration légale γ_i*, sans occurrence de fautes, chaque nœud restera dans une configuration légale.*

Preuve. Soit γ_i une configuration légale, $\forall\ u \in V$ u est fixé et seule la règle R_0 s'exécutera. Nous aurons donc $\forall j > i$, à γ_j une configuration légale.

Remarque 6.1 *(Pire des cas)*

Avec notre approche, nous observons le pire des cas dans une topologie où les nœuds forment une chaîne ordonnée. Dans une telle topologie, comme illustré au niveau de la figure 5, les nœuds se fixent du plus grand au plus petit et le temps de stabilisation est de $n + 2$ *transitions.*

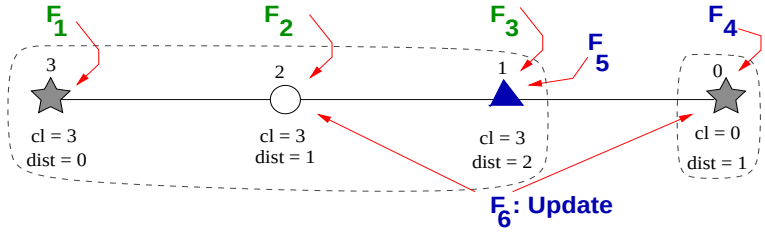


Figure 5 – Stabilisation dans une chaîne ordonnée

6.2. Occupation mémoire

Dans cette section, nous montrons l'occupation mémoire maximale nécessaire dans dans notre solution.

Lemme 6.2 *L'espace mémoire nécessaire pour chaque voisin est* $\log(2n + 3 + k)$ *bits.*

Preuve.

Pour fonctionner, chaque nœud u du réseau a besoin de connaître pour chacun de ses voisins $v \in N_u$ son l'identifiant (id_v), son identifiant de *cluster* (cl_v), son statut ($statut_v$) et sa distance $dist_{(v,CH_v)}$. Ainsi, pour un réseau de n nœuds, nous avons au plus :

- n valeurs possibles pour l'identifiant du nœud ;
- n valeurs possibles pour l'identifiant de *cluster* ;
- trois (3) valeurs de statut différentes que sont CH, SN et GN ;
- k valeurs possibles pour la distance.

Or, il est nécessaire d'avoir $\log(n)$ bits pour encoder une variable avec n valeurs possibles. Donc, pour chaque nœud voisin, l'algorithme a besoin d'au plus de $\log(2n+k+3)$ bits.

Corollaire 6.2 *Chaque nœud u a besoin de $(\Delta_u + 1) * \log(2n + 3 + k)$ bits.*

Preuve.

D'après le lemme 6.2, chaque voisin v d'un nœud u requiert $\log(2n+k+3)$ bits. Soit N_u le voisinage du nœud u. Posons $\Delta_u = |N_u|$ comme étant le degré - nombre de voisins - du nœud u. Or, pour l'exécution de l'algorithme, chaque nœud u a besoin de connaître l'état de touts ses Δ_u voisins. De plus, u mémorise aussi les mêmes informations pour lui.

Donc, chaque nœud u a besoin de $(\Delta_u + 1) * \log(2n + 3 + k)$ bits.

Corollaire 6.3 *Chaque nœud u a besoin d'au plus $n * \log(2n + 3 + k)$ bits.*

Preuve.

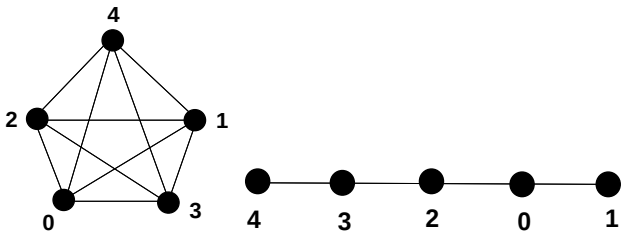
Pour un réseau de n nœuds, le voisinage N_u de chaque nœud u peut contenir au maximum $n - 1$ voisins (pour un graphe complet). Dans ce cas précis, pour fonctionner, notre algorithme a besoin de $n * \log(2n + 3 + k)$ bits par nœud.

6.3. Discussions

Nous venons de montrer avec le corollaire 6.3 que nous observons l'occupation mémoire la plus importante dans le cas d'un graphe complet où chaque nœud a besoin de $n * \log(2n + 3 + k)$ bits. Or, comme dans un graphe complet tous les nœuds sont voisins (c.f. Figure 6(a)), nous pouvons aisément remarquer qu'en 2 transitions tous les nœuds se fixent. En effet, dans un graphe complet, partant d'une configuration quelconque γ_0, à la configuration γ_1, le nœud possédant la plus grande identité se fixe en premier. Puis, à la configuration γ_2, tous les autres nœuds se fixent au nœud d'identité maximale. Et nous obtenons un seul *cluster*. Dans ce cas, l'occupation mémoire est maximale et le temps de stabilisation est minimal.

Cependant, comme nous l'avons montré dans la section 6.1, pour l'autre critère déterminant de algorithme, à savoir le temps de stabilisation, nous observons le pire des cas dans une topologie en chaîne ordonnée. Or, comme illustré dans l'exemple de la chaîne ordonnée de la figure 6(b), les nœuds situés au deux extrémités dans la chaîne ont un (1) seul voisin alors que ceux à l'intérieur de la chaîne n'ont que deux (2) voisins. Dans ce cas, l'occupation mémoire est minimale et le temps de stabilisation est maximale.

En résumé, dans le cas où nous observons un temps de stabilisation au maximal (chaîne ordonnée), l'occupation mémoire est la moindre. Et inversement, dans le cas ou l'occupation mémoire est au maximale (graphe complet), le temps de stabilisation est au plus petit.



(a) Graphe complet (b) Chaîne ordonnée

Figure 6 – Graphe complet vs Chaîne Ordonnée

6.4. Comparaison analytique

Le tableau 2, illustre une comparaison du temps de stabilisation, de l'espace mémoire par voisin et du voisinage de notre algorithme avec ceux des meilleurs solutions de *clustering* auto-stabilisantes à k sauts fondées soit sur un sur un modèle à états [16, 13] soit sur un modèle synchrone à passage de messages [26].

Dans un premier temps, nous pouvons noter que le temps de stabilisation de notre solution n'est pas fonction du paramètre k contrairement aux solutions proposées par dans [13, 26]. Étant donné que notre approche combine la découverte de voisinage et la procédure de *clustering* en une seule phase, elle présente un unique temps de stabilisation contrairement à la solution de Datta *et al.* [16] et de Larsson and Tsigas [26]. Dans ces deux solutions, les auteurs commencent d'abord par construire l'ensemble de *cluster-heads*. Puis, partant de cet ensemble, ils structurent le réseaux en *clusters* de rayon k.

Nous pouvons aussi remarquer que notre solution considère un voisinage à distance 1 pour construire des *clusters* à k sauts. Alors que dans [16] et [13], les processus ont respectivement des rayons de lecture de k et $k+1$ sauts. Cependant, nous notons que notre algorithme nécessite plus de mémoire par nœud comparé aux solutions qui utilisent un modèle à états [16, 13]. En effet, contrairement aux solutions décrites dans [16] et [13], nous utilisons un modèle asynchrone à passage de messages. Or, dans un tel modèle, chaque nœud mémorise les informations définissant les états de ses voisins. Ce qui n'est pas le cas dans un modèle à états qui est utilisé dans [16, 13].

Remarquons néanmoins que notre algorithme requiert moins de mémoire que la solution de Larsson et Tsigas [26] qui utilise un modèle synchrone à passage de messages. En effet, dans la solution de Larsson et Tsigas [26], les messages sont retransmis jusqu'à une distance k alors que dans la notre, les échanges de messages se font uniquement entre voisins de distance 1.

	Temps de stabilisation	Occupation mémoire par nœud	Voisinage
Notre solution	$n+2$	$\log(2n+k+3)$	1 saut
Caron *et al.* [13]	$O(n*k)$	$O(\log(n)+\log(k))$	k+1 sauts
Datta *et al.* [16]	$O(n)$, $O(n^2)$	$O(\log(n))$	k sauts
Larsson et Tsigas [26]	$O(k)$, $O(g*k*\log(n))$	$O(\lvert G_i^k\rvert *(\log(n)+\log(k)))$	k saut

Tableau 2 – Comparaison analytique du temps de stabilisation, de l'occupation mémoire et du voisinage

7. Évaluation de performances moyennes

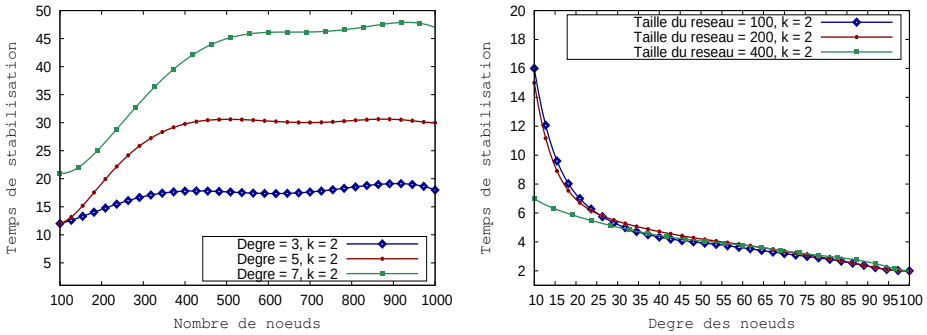
Dans cette section, nous présentons une implémentation de notre algorithme et une évaluation de ses coûts et performances moyennes par simulation.

Comme nous l'avons montré dans la Section 6, notre réseau converge dans le pire des cas en $n + 2$ transitions. Ceci correspond au cas le plus défavorable d'une topologie où les nœuds forment une chaine ordonnée. Or, un réseau *Ad Hoc* est caractérisé par une topologie dense et aléatoire. Ainsi, nous faisons recours à une campagne de simulation pour évaluer les performances moyennes de notre algorithme.

Nous avons utilisé le simulateur *OMNeT++* [31] et la librairie *SNAP* [1] comme générateur de graphes aléatoires. Toutes nos simulations sont effectuées sous *Grid'5000* [12]. De plus, nous fixons un intervalle de confiance de 99% et nous fournissons des valeurs moyennes sur une série de 100 simulations pour chaque taille de réseaux.

7.1. Impact de la taille du réseaux et du degré des nœuds

Nous entamons l'évaluation des performances moyennes de notre solution en étudiant l'impact de la densité et du nombre de nœuds du réseau sur le temps de stabilisation du réseau.



(a) Temps de stabilisation en fonction du nombre de nœuds

(b) Temps de stabilisation en fonction du degré des nœuds

Figure 7 – Impact de la taille du réseaux et du degré des nœuds sur le temps de stabilisation

Dans la première série d'expériences présentée au niveau la figure 7(a), nous fixons le paramètre $k = 2$ et nous faisons varier le nombre de nœuds de 100 à 1000 par pas de 100. Pour chaque taille de réseau fixée, à l'aide de la librairie SNAP, nous générons des graphes aléatoires $\delta-$réguliers, où δ est reprendre degré des nœuds (nombre de voisins). Nous fixons arbitrairement le degré δ à 3, 5 et 7. Puis, pour chaque taille de réseau, nous calculons le temps de stabilisation comme étant le nombre maximal de transitions obtenus jusqu'à la formation des *clusters* stables.

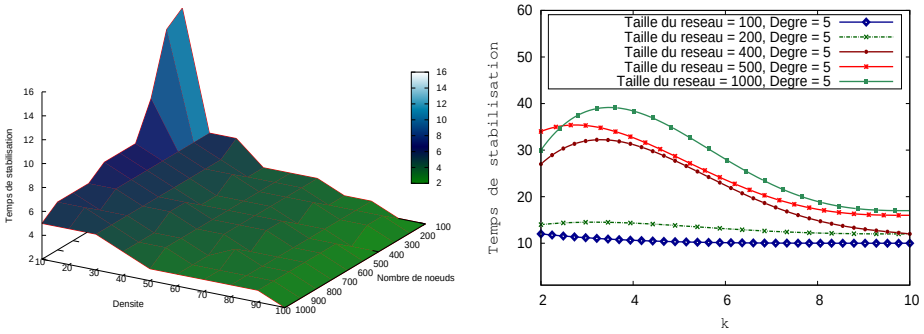
Ainsi, au niveau de la figure 7(a), nous remarquons que le temps de stabilisation moyen augmente légèrement avec l'augmentation du nombre de nœuds mais varie peu à partir d'une certaine taille du réseau. De plus, nous notons que pour des topologies totalement aléatoires, le temps de stabilisation moyen est très en dessous de $n + 2$ transitions (valeur formelle prouvée dans le pire des cas).

Pour mieux observer l'impact du degré des nœuds sur le temps de stabilisation, comme illustré avec la figure 7(b), nous fixons la taille du réseau et nous faisons varier le degré de nœuds.

Nous observons que pour des tailles de réseau fixées à 100, 200 et 400 nœdus (cf. Figure 7(b)), plus le degré des nœuds augmente, plus le temps de stabilisation diminue. En effet, si le degré augmente, les nœuds ayant les plus grandes identités du réseau ont plus de voisins. Donc, ils attirent plus de nœuds dans leurs *clusters* durant chaque transition. Ainsi, avec notre approche, plus le degré des nœuds augment, plus le temps de stabilisation diminue. Or, les réseaux *Ad Hoc* sont souvent caractérisés par une forte densité.

7.2. Étude du passage à l'échelle et impact du paramétrer k

Pour étudier le passage à l'échelle de notre solution, nous faisons varier le nombre de nœuds dans le réseau en même temps que la densité de connexité du réseau. Pour $k = 2$, nous faisons évoluer le nombre de nœuds de 100 à 1000 par pas de 100. Pour chaque taille de réseau fixée, nous faisons varier la densité du réseau de 10% à 100% par pas de 10 en générant des graphes aléatoires suivant le modèle de Erdos Renyi [18] à l'aide de la librairie SNAP. Nous obtenons la courbe *3D* de la figure 8(a).



(a) Passage à l'échelle (b) Impacte du paramètre k

Figure 8 – Passage à l'échelle - impacte du paramètre k

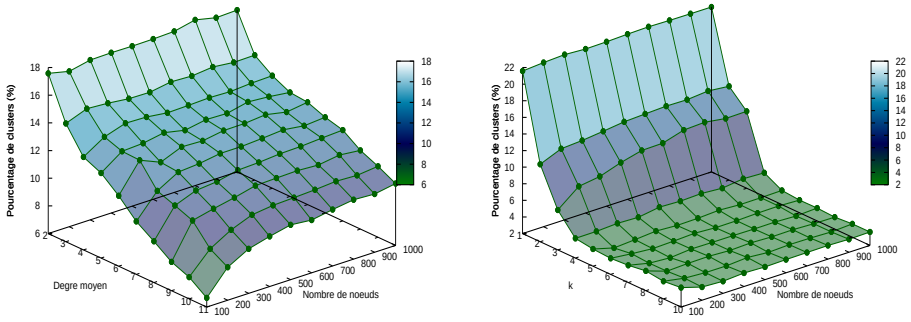
Nous remarquons que sauf pour de faibles densités (10% et 20%), le temps de stabilisation varie légèrement avec l'augmentation du nombre de nœuds. Et cas de faibles densités, nous observons un pic. Mais avec l'augmentation du nombre de nœuds, le temps de stabilisation diminue et nous observons le même phénomène que la figure 7(b). Avec cette série de simulations, nous pouvons soulever deux observations. *(i)* Seule la densité de connexité est le facteur déterminant avec notre approche. *(ii)* En moyenne, pour des réseaux avec une topologie quelconque, le temps de stabilisation est très inférieur à celui du pire des cas ($n + 2$ transitions). Rappelons que ce pire cas est une topologie où les nœuds forment une chaîne ordonnée comme nous l'avons prouvé dans Section 6.

Pour analyser l'impact du paramètre k, nous fixons arbitrairement un degré de 5 et nous considérons des réseaux de taille 100, 200, 400, 500 et 1000 nœuds. Pour chaque taille de réseau, nous faisons varier la valeur de k de 2 à 10. La figure 8(b) montre le temps de stabilisation en fonction de la valeur de k. Nous observons une diminution du temps de stabilisation avec l'augmentation de la valeur de k. En effet, comme les messages $hello$ échangés contiennent la valeur de la distance k, si k augmente, le champs d'influence des

nœuds avec une plus grande identité augmente. Les nœuds effectuent moins de transitions pour se *fixer* à un CH. Avec de petites valeurs du paramètre k, nous avons des *clusters* de faibles diamètres. Donc, il nécessite plus de transitions pour atteindre un état stable dans tous les *clusters*. Notons que quelle que soit la valeur du paramètre k et pour des graphes de réseaux totalement aléatoires, nous obtenons des temps de stabilisation très inférieurs au pire des cas ($n + 2$ transitions).

7.3. Évaluation du nombre de *clusters*

Avec la campagne de simulation présentée au niveau de la figure 9, nous évaluations nombre *clusters* construit par notre solution. Pour ce faire, nous calculons le pourcentage de *clusters* construit comme étant le nombre *cluster-head* obtenu sur le nombre total de nœuds dans le réseau. Cette évaluation s'est faite en fonction du degré moyen du réseau et en fonction du paramètre k.



(a) Pourcentage de *clusters* construit en fonction du degré du réseau

(b) Pourcentage de *clusters* construit en fonction du paramètre k

Figure 9 – Pourcentage de *clusters*

Dans la figure 9(a), nous fixons le paramètre $k = 2$ puis, nous faisons varier le nombre de nœuds de 100 à 1000. Pour chaque taille de réseau fixée, nous faisons varier le degré moyen de 2 à 11. Nous observons d'abord que, pour chaque valeur du degré moyen fixée, le pourcentage de *clusters* varie peu avec la l'augmentation de la taille du réseau. Ceci atteste qu'en termes de pourcentage de *clusters* construit, notre solution assure le passage à l'échelle. Par ailleurs, nous remarquons que, pour chaque taille de réseau fixée, le pourcentage de *clusters* construit diminue avec l'augmentation du degré moyen dans le réseau. En effet, l'augmentation du degré moyen entraîne un plus grand nombre de voisins pour les nœuds d'identité plus grande. Ainsi, ces derniers attirent dans leurs *clusters* un nombre plus important de nœuds. D'où une diminution du pourcentage de *clusters*

Au niveau de la figure 9(b), nous fixons un degré moyen de 6 voisins puis, nous faisons varier le nombre de nœuds de 100 à 1000. Pour chaque taille de réseau fixée, nous faisons évoluer le paramètre k de 1 à 10. Nous observons d'abord que pour chaque valeur du paramètre k fixée, le taux de *cluster-head* varie très peu avec l'augmentation du nombre de nœuds dans le réseau. Ceci atteste du passage à l'échelle de notre solution pour le pourcentage de *clusters* construit en fonction fonction du paramètre k. Par ailleurs, nous notons une forte chute du taux de *cluster-head* avec suivant l'augmentation du paramètre

k. En effet, avec l'augmentation du paramètre k, le diamètre des *clusters* augmente. Ainsi, nous obtenons des *clusters* contenant plus de nœuds. D'où une diminution du pourcentage de *cluster-head*. Notons aussi que les valeurs du paramètre qui donnent les pourcentages de *cluster-head* les plus intéressant sont 2 et 3. Au delà de 3, nous commençons à obtenir des *clusters* de diamètres plus grands.

La figure 9(a) nous montre que, selon le degré du réseau, notre approche construit un taux de *cluster-head* entre 6% et 18%. La figure 9(b) quant à elle illustre que, pour les valeurs les plus intéressantes du paramètre k (2 et 3), nous obtenons un taux de *cluster-head* entre 7% et 16%. Plusieurs études, comme [2, 14, 32], ont montré que le nombre optimal de *cluser-heads* doit correspondre entre 5% et 20% du nombre total de nœuds dans le réseau.

7.4. Tolérance aux fautes transitoires : évaluation du coût de re-clustering

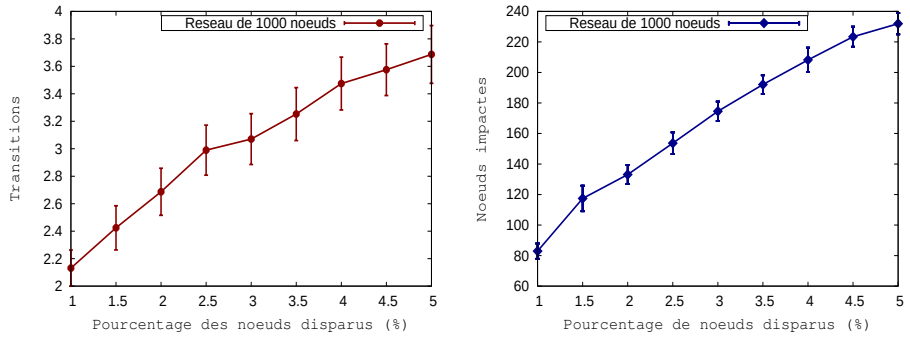
Dans les réseaux *Ad Hoc*, la mobilité des nœuds entraîne des changements de topologie qui peuvent occasionner des fautes transitoires par modification de la structure des *clusters*. Ceci peut arriver lorsqu'un nœud jouant le rôle de *cluster-head* venait à changer de position dans le réseau. De plus, un nœud peut disparaître du réseau du fait l'épuisement de son énergie, d'une panne de son module de communication ou d'autres facteurs externes dûs à l'environnement de déploiement qui est souvent hostile.

Nous supposons la mobilité (arrivée ou départ d'un nœud dans un *cluster*) ou la disparition d'un nœud entraînant un changement dans le *cluster* comme une faute transitoire. Celle-ci est alors détectée et corrigée automatiquement par l'algorithme 1 sans intervention extérieure. Nous considérons qu'une faute transitoire survient après la stabilisation c'est à dire après la formation de *clusters* stables. En effet, une faute lors de la phase de *clustering* est transparente. La détection d'une faute s'effectue toujours à l'aide des messages *hello*. Un nœud qui ne reçoit plus de réponses d'un voisin après un temps bien déterminé le considère comme disparu ou ayant changé de position dans le réseau. Ainsi, nous évaluons l'impact de fautes transitoires en mesurant le coût de la procédure de *re-clustering* en termes de nombre de transitions supplémentaires pour retrouver un état stable et le nombre de nœuds impactés.

Dans la série d'expériences illustrée au niveau de la figure 10, nous fixons un réseau de 1000 nœuds avec un degré moyen de 6. Ensuite, nous exécutons l'algorithme 1 jusqu'à la stabilisation. Puis, nous introduisons des fautes transitoires en faisant disparaître aléatoirement 1 à 5% des nœuds du réseau. Pour chaque pourcentage de nœuds disparus, une série de 100 simulations est effectuée.

Dans la figure 10(a), nous calculons le temps de stabilisation maximal comme étant le nombre de transitions supplémentaires nécessaire pour retrouver un nouvel état légal. Nous observons que le nombre de transitions nécessaire pour corriger les fautes transitoires augmente en suivant l'augmentation du taux de nœuds disparus. En effet, comme illustré dans la figure 10(b), l'augmentation du pourcentage de nœuds disparus fait croître logiquement le nombre nœuds impactés dans le réseau. De ce fait, il nécessite plus de transitions pour que le réseau devient stable à nouveau. Cependant, une disparation jusqu'à 5% des nœuds du réseau n'a d'impact que sur environs $1/4$ des nœuds du réseau. Ceci atteste que les fautes transitoires n'impactent que les *clusters* dans lesquels elles se sont produites et éventuellement les *clusters* adjacents. Également, nous observons que

pour une disparition jusqu'à 5%, le nombre de transitions supplémentaires requis pour que le réseau retrouve un état stable est très moindre comparé au temps de stabilisation en partant d'une configuration quelconque (cf. Figure 7 et Figure 8(a)).



(a) Transitions supplémentaires (b) Pourcentage de nœuds impactés

Figure 10 – Impact de la disparation de 1 à 5 $\%$ de nœuds

Les résultats expérimentaux illustrés dans la figure 10 montrent que la correction de fautes transitoires nécessite un nombre moindre de transitions. En effet, les fautes transitoires impactent les *cluters* dans lesquels elles se sont produites et éventuellement les *clusters* adjacents. De plus, les nœuds réutilisent les informations déjà présentes dans leurs tables de voisinage pour la correction de fautes transitoires. Ainsi, notre solution s'adapte aux fautes transitoires dues à des modifications topologiques.

7.5. Discussions

Dans cette section, nous venons d'évaluer les performances moyennes de notre solution. Les résultats de simulation ont notamment montré que pour des topologies réseaux suivant des graphes totalement aléatoires, nous obtenons des temps de stabilisation très inférieurs à celui du pire des cas. Rappelons que, comme montré dans la Section 6, le pire des cas correspond à la topologie particulière de chaîne ordonnée. Dans ce cas, nous obtenons un temps de stabilisation de $n+2$ transitions. De plus, en cas fautes transitoires, les simulations montrent d'une part que le nombre de transitons nécessaire pour retrouver un état stable est moindre. D'autre part, les fautes transitoires impactent les *cluters* dans lesquels elles se sont produites et éventuellement les *clusters* adjacents.

Dans [7, 8], nous avons appliqué notre algorithme dans le cadre des Réseaux de Capteurs Sans Fil (RCSF) avec contrainte énergétique. Nous avons montré que notre solution est générique et complète. En plus du critère principal d'élection du *cluster-head* (identité maximale), notre algorithme peut s'utiliser aisément avec d'autres critères tels que le degré des nœuds ou l'énergie résiduelle. Nous avons validé cette étude en évaluant, suivant plusieurs critère d'élection du *cluster-head*, le coût de communication en termes de nombre total de messages échangés jusqu'à la formation des *clusters* stables dans tout le réseau et la consommation énergétique qui en découle. Les études que nous avons mené dan [7, 8] montrent que le critère d'identité maximale apporte plus de stabilité lors de la construction des *clusters*. Rappelons que pour mesurer la consommation énergétique, nous avons implémenté le modèle énergétique de référence dans les RCSF proposé par Heinzelman et *al.* [20].

Également, dans [6], nous avons mené une étude comparative, toujours dans le contexte des RCSF avec contrainte énergétique, de notre solution avec l'approche proposée par Mitton et *al.* dans [27]. En effet, à notre connaissance, l'algorithme de Mitton et *al.* est la solution de référence dans notre modèle. C'est à dire un algorithme de *clustering* auto-stabilisant à k sauts, distribué et utilisant un modèle asynchrone à passage de messages. Nous avons comparé les deux approches, sous le même modèle et environnement de tests, en évaluant le coût de communication et la consommation énergétique. Les résultats obtenus dans [6] montrent que notre approche réduit le coût de communication et la consommation énergétique par un facteur 2 comparé à la solution de Mitton et *al.* [27] .

8. Conclusion

Dans cet article, nous avons présenté un algorithme complètement distribué et auto-stabilisant pour structurer le réseau en *clusters* non-recouvrants de diamètre au plus $2k$. Sans initialisation, notre solution n'utilise que des informations provenant des nœuds voisins à distance 1. Elle combine la découverte du voisinage et la construction des *clusters*.

Nous avons montré que, partant d'un état quelconque, un état stable est atteint en au plus $n+2$ transitions. De plus, il nécessite une occupation mémoire de $\Delta_u * \log(2n+k+3)$ bits pour chaque nœud u où Δ_u représente le degré - nombre de voisins - de u, n le nombre total de nœuds et k la distance maximale dans les *clusters*.

Les compagnes de simulation, menées sous OMNeT++, montrent d'une part que pour des réseaux totalement aléatoires, le temps de stabilisation moyen est très inférieur à celui du pire des cas. D'autre part, après l'occurrence de fautes transitoires, le nombre de transitions supplémentaires pour retrouver un nouvel état stable et le nombre de nœuds impactés sont moindres.

Actuellement, nous travaillons sur l'élaboration d'un protocole de routage avec aggravation de donnée utilisant la structure de nos *clusters* dans le but d'optimiser l'acheminement de l'information.

Remerciements

Ces travaux ont été effectués dans le cadre du projet CPER CapSec ROFICA co-financé par la région de Champagne-Ardenne et le FEDER. Nous remercions également le Centre de Calcul de Champagne-Ardenne ROMEO [1] et Grid'5000.

9. Bibliographie

[1] 2013. SNAP : Stanford Network Analysis Platform.

1. Le Centre de Calcul de Champagne-Ardenne ROMEO est une plateforme technologique de l'Université de Reims Champagne-Ardenne soutenue par la région Champagne-Ardenne depuis 2002. Pour plus d'information : https ://romeo.univ-reims.fr

[2] Navid Amini, Alireza Vahdatpour, Wenyao Xu, Mario Gerla, and Majid Sarrafzadeh. Cluster size optimization in sensor networks with decentralized cluster-based protocols. *Computer Communications*, pages 207 – 220, 2012.

[3] 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*, pages 32–41, 2000.

[4] H. Attiya and J. Welch. *Distributed computing : fundamentals, simulations, and advanced topics*. Wiley series on Parallel and Distributed Computing. 2004.

[5] Mandicou Ba, Olivier Flauzac, Bachar Salim Haggar, Florent Nolot, and Ibrahima Niang. Self-Stabilizing k-hops Clustering Algorithm for Wireless Ad Hoc Networks. In *7th ACM IMCOM (ICUIMC)*, pages 38 :1–38 :10, 2013.

[6] Mandicou Ba, Olivier Flauzac, Rafik Makhloufi, Florent Nolot, and Ibrahima Niang. Comparison Between Self-Stabilizing Clustering Algorithms in Message-Passing Model. In *9th ICAS*, pages 27–32, 2013.

[7] Mandicou Ba, Olivier Flauzac, Rafik Makhloufi, Florent Nolot, and Ibrahima Niang. Energy-Aware Self-Stabilizing Distributed Clustering Protocol for Ad Hoc Networks : the case of WSNs. *KSII Transactions on Internet and Information Systems*, page 19p, 2013. To appear.

[8] Mandicou Ba, Olivier Flauzac, Rafik Makhloufi, Florent Nolot, and Ibrahima Niang. Evaluation Study of Self-Stabilizing Cluster-Head Election Criteria in WSNs. In *6th CTRQ*, pages 64–69, 2013.

[9] Lalia Blin, Maria Gradinariu Potop-Butucaru, and Stephane Rovedakis. Self-stabilizing minimum degree spanning tree within one from the optimal degree. *JPDC*, pages 438 – 449, 2011.

[10] Alain Bui, Abdurusul Kudireti, and Devan Sohier. A fully distributed clustering algorithm based on random walks. *IPDPS*, pages 125–128, 2009.

[11] Alain Bui, Devan Sohier, and Abdurusul Kudireti. A random walk based clustering with local recomputations for mobile ad hoc networks. *IPDPSW*, pages 1–8, 2010.

[12] F. Cappello and et *al.* Grid'5000 : A large scale and highly reconfigurable grid experimental testbed. In *GRID*, pages 99–106, 2005.

[13] Eddy Caron, Ajoy K. Datta, Benjamin Depardon, and Lawrence L. Larmore. A self-stabilizing k-clustering algorithm for weighted graphs. *JPDC.*, pages 1159–1173, 2010.

[14] Ali Chamam and Samuel Pierre. A distributed energy-efficient clustering protocol for wireless sensor networks. *Computers & Electrical Engineering*, pages 303 – 312, 2010.

[15] Ajoy Datta, Lawrence Larmore, and Priyanka Vemula. Self-stabilizing leader election in optimal space. In *SSS*, pages 109–123. 2008.

[16] Ajoy K. Datta, Stéphane Devismes, and Lawrence L. Larmore. A self-stabilizing O(n)-round k-clustering algorithm. In *SRDS*, pages 147–155, 2009.

[17] Edsger W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Communications of the ACM*, pages 643–644, 1974.

[18] P. Erdos and A. Renyi. On the evolution of random graphs. *Publication of the Mathematical Institute of the Hungarian Academy of Sciences*, pages 17–61, 1960.

[19] Olivier Flauzac, Bachar Salim Haggar, and Florent Nolot. Self-stabilizing clustering algorithm for ad hoc networks. *ICWMC*, pages 24–29, 2009.

[20] Wendi Rabiner Heinzelman, Anantha Chandrakasan, and Hari Balakrishnan. Energy-efficient communication protocol for wireless microsensor networks. In *HICSS*, 2000.

[21] Colette Johnen and Fouzi Mekhaldi. Self-stabilization versus robust self-stabilization for clustering in ad-hoc network. In *17th Euro-Par*, pages 117–129, 2011.

[22] Colette Johnen and Le Nguyen. Self-stabilizing weight-based clustering algorithm for ad hoc sensor networks. In *ALGOSENSORS*, pages 83–94. 2006.

[23] Colette Johnen and Le Huy Nguyen. Robust self-stabilizing weight-based clustering algorithm. *TCS*, pages 581 – 594, 2009.

[24] Colette Johnen and LeHuy Nguyen. Self-stabilizing construction of bounded size clusters. *ISPA*, pages 43–50, 2008.

[25] J. Kuroiwa, Y. Yamauchi, Weihua Sun, and M. Ito. A self-stabilizing algorithm for stable clustering in mobile ad-hoc networks. In *4th IFIP NTMS*, pages 1–7, 2011.

[26] Andreas Larsson and Philippas Tsigas. Self-stabilizing (k,r)-clustering in wireless ad-hoc networks with multiple paths. In *14th OPODIS*, pages 79–82, 2010.

[27] N. Mitton, E. Fleury, I. Guerin Lassous, and S. Tixeuil. Self-stabilization in self-organized multihop wireless networks. In *ICDCSW*, pages 909–915, 2005.

[28] Nathalie Mitton, Anthony Busson, and Eric Fleury. Self-organization in large scale ad hoc networks. In *MED-HOC-NET*, 2004.

[29] Gregory J Pottie and William J Kaiser. Wireless integrated network sensors. *Communications of the ACM*, pages 51–58, 2000.

[30] G. Tel. *Introduction to Distributed Algorithms*. Cambridge University Press, 2000.

[31] András Varga and Rudolf Hornig. An overview of the OMNeT++ simulation environment. In *Simutools*, pages 60 :1–60 :10, 2008.

[32] Ossama Younis and Sonia Fahmy. Heed : a hybrid, energy-efficient, distributed clustering approach for ad hoc sensor networks. *IEEE Transactions on Mobile Computing*, pages 366 – 379, 2004.

[33] Jiguo Yu, Yingying Qi, Guanghui Wang, and Xin Gu. A cluster-based routing protocol for wireless sensor networks with nonuniform node distribution. *AEU - International Journal of Electronics and Communications*, pages 54 – 61, 2012.