

Self-Stabilizing Clustering Algorithm for Ad hoc Networks

Colette Johnen, Le Huy Nguyen
LRI–Université Paris Sud, CNRS UMR 8623
Bâtiment 490, F91405, Orsay Cedex, France
E-mail: colette@lri.fr, lehuy@lri.fr

24 janvier 2006

Abstract

Ad hoc networks consist of wireless hosts that communicate with each other in the absence of a fixed infrastructure. Such network cannot rely on centralized and organized connectivity. The clustering problem consists in partitioning network nodes into groups called clusters, thus giving at the network a hierarchical organization. Clustering is commonly used in order to limit the amount of routing information. A self-stabilizing algorithm, regardless of the initial system state, converges in finite time to a set of states that satisfy a legitimacy predicate without external intervention. Due to this property, self-stabilizing algorithms provide means for tolerating transient faults. In this paper we present a weight-based Self-stabilizing Clustering Algorithm for Ad hoc network. Our algorithm adapts to the changes in the network topology due to mobility of the nodes.

Keywords : Self-stabilization, Distributed algorithm, Clustering, Ad hoc network.

Résumé

Les réseaux ad-hoc se composent des postes sans fil qui communiquent les uns avec les autres en l'absence d'une infrastructure fixe. Un tel réseau ne peut pas se fonder sur une connectivité centralisée et organisée. Le problème d'agrégation consiste à partitionner les noeuds d'un réseau en grappes, donc donne au réseau une organisation hiérarchique. L'agrégation est généralement utilisée afin de réduire la quantité de l'information échangée en vu du routage. Un algorithme auto-stabilisant, indépendant de l'état initial du système, converge à un ensemble de l'état qui satisfait à un prédicat légitime dans un temps fini. Grâce à cette propriété, les algorithmes auto-stabilisants donnent des moyens pour tolérer les défaillances transitoires. Dans cet article, nous présentons un algorithme auto-stabilisant d'agrégation pour les réseaux ad-hoc. Notre algorithme s'adapte aux changements de topologie de réseau suite à la mobilité de noeuds.

Mots-clés : Auto-stabilization, Algorithme distribué, Agrégation, Réseau ad-hoc.

1 Introduction

An *ad-hoc* network is a self-organized multihop network especially one with wireless or temporary plug-in connections. Such a network may operate in a standalone fashion, or may be connected to the larger Internet [11]. In Latin, *ad hoc* literally means “for this”, further meaning “for this purpose only” and thus usually temporary. Mobile routers may move randomly; thus, the network’s topology may change rapidly and unpredictably. Such network cannot rely on centralized and organized connectivity. Significant examples include establishing survivable, efficient, dynamic communication for emergency/rescue operations, disaster relief efforts, and military networks, the meeting where participant will create a temporary wireless *ad hoc* network. Minimal configuration and quick deployment are needed in these situations.

Clustering means partitioning network nodes into groups called clusters, giving at the network a hierarchical organization. A cluster is a connected graph including a clusterhead and (possibly) some ordinary nodes. Each node belongs to only a cluster. In addition, a cluster is required to obey certain constraints that are used for network management, routing methods, resource allocation, etc. By dividing the network into nonoverlapped clusters, intraccluster routing is administered by the cluster leader and inter cluster routing can be done in reactive manner by cluster leaders and gateway. Clustering has the following advantages. First, clustering facilitates the reuse of resource, which can improve the system capacity. Members in a cluster can share the resource as software, memory space, printer, etc, thus increasing its disposability and its accessibility. Secondly, clustering-based routing reduces the amount of routing information propagated in the network. Finally, clustering can be used to reduce the amount of information that is used to store the network state. The clusterhead will collect the state of nodes in its cluster and built an overview of its cluster state. Distant nodes outside of the cluster usually do not need to know the details of specific events occurring inside the cluster. Hence, an overview of the cluster’s state is sufficient for those distant nodes to make control decisions.

For these reasons, it is not surprising that several distributed clustering algorithms have been proposed in this area during the last few years [12, 17, 2, 3, 1, 10, 7]. The clustering algorithms appear in [1, 10] build a spanning tree, then on top of the spanning tree, the clusters are constructed. In [1, 10], the clusterheads set is not a dominating set (i.e. a processor can be at distance greater than 1 of its clusterhead). Two network architectures for MANET (Mobile *ad hoc* Wireless network) are proposed in [12, 17] where nodes are organized into clusters. The clustering algorithms in [12, 17] are similar. The built clusterheads set is an independent (i.e. clusterheads are not neighbours) and also a dominating set. The clusterheads are selected according to their ids value. In [7] a weight-based distributed clustering algorithm taking into account several parameters (processor’s degree, transmission and battery power, processor mobility). In a neighbourhood, the processors elected are those most suitable to the clusterhead role (i.e. a processor optimizing all the parameters). In [3] a distributed and mobility-adaptive clustering algorithm, called DMAC, is presented; the clusterheads are selected

according a node's parameter (called *weight*). The bigger is the weight of a node, the more suitable this node is for the role of clusterhead. In the both paper, the clusterheads set is an independent and dominating set. An extended version of this algorithm, called Generalized DMAC (GDMAC), was proposed in [2]. In the latter algorithm, the clusterheads set does not have to be an independent set. This implies that, when due to mobility of the nodes two or more clusterheads become neighbours, none has to resign. Thus, the clustering management with GDMAC requires less overhead than the clustering management with DMAC in highly mobile environment. The DMAC and GDMAC algorithms are analyzed respectively in following paper [6, 5] with respect to their convergence time and message complexity. These two algorithms DMAC and GDMAC are not self-stabilizing.

In 1973 Dijkstra [8] introduced to computer science the notion of self-stabilization in the context of distributed systems. He defined a system as self-stabilizing when “regardless of its initial state, it is guaranteed to arrive at a legitimate state in a finite number of step”. A system which is not self-stabilizing may stay in an illegitimate state forever. The design of self-stabilizing distributed algorithms has emerged as an important research area in recent years [18, 9]. The correctness of self-stabilizing algorithms does not depend on initialization of variables, and a self-stabilizing algorithm converges to some predefined stable state starting from an arbitrary initial state. Self-stabilizing algorithms are thus inherently tolerant to transient faults in the system. Many self-stabilizing algorithms can also adapt dynamically to changes in the network topology or system parameters (e.g., communication speed, number of nodes). A state following a topology changes is seen as an inconsistent state from which the system will converge to a state consistent with the new topology. [13] presents a self-stabilizing algorithm that builds a maximal independent set (i.e. members of the set are not neighbours, and the set cannot contains any other processors). Notice that a maximal independent set is a good candidate for the clusterheads set because a maximal independent set is also a dominating set (i.e. any processor is member of the dominating set or has a neighbour that is member of the set). Appears in [19], a self-stabilizing algorithm that creates a minimal dominating set (i.e. if a member of the set quits the set, the set is not more a dominating set). Notice that a minimal dominating set is not always an independent set. We present in this paper a self-stabilizing version of DMAC and GDMAC algorithm.

The paper is organized as follows. In section 2, the formal definition of self-stabilization and clustering is given. The self-stabilization version of the DMAC algorithm is presented in section 3; self-stabilization proof is also presented. In section 4, the self-stabilizing GDMAC algorithm is presented with its proof.

2 Model

2.1 Distributed System

In this paper, we consider the state model [4, 15, 14]. A distributed system \mathcal{S} is a set of state machines called processors. Each processor can communicate with

a subset of the processors called neighbors. We model a distributed system by an undirected graph $G = (V, E)$ in which V , $|V| = n$, is the set of nodes and there is an edge $\{u, v\} \in E$ if and only if u and v can mutually receive each others' transmission (this implies that all the links between the nodes are bidirectional). In this case we say that u and v are neighbors. The set of the neighbors of a node $v \in V$ will be denoted by N_v . We assume the locally shared memory model of communication. Thus, each processor i has a finite set of *local variables* such that the variables at a processor i can be read by i or any neighbors of i , but can be modified by i only. Each processor has a program and processors execute their programs asynchronously. We assume that the program of each processor i consists of a finite set of guarded statements of the form $Rule : Guard \rightarrow Action$, where $Guard$ is a boolean predicate involving the local variables of i and the local variables of its neighbors, and $Action$ is an assignment that modifies the local variables in i . The *rule* R is executed only if the corresponding guard $Guard$ evaluates to true, in which case we say guard $Guard$ is enabled. The execution by processor i of an action rule with enabled guard is called a *step* of i . The *state* of a processor is defined by the values of its local variables. A *configuration* of a distributed system $G = (V, E)$ is an instance of the states of its processors. The set of configurations of G is denoted as \mathcal{C} . A computation e of a system G is a sequence of configurations c_1, c_2, \dots such that for $i = 1, 2, \dots$, the configuration c_{i+1} is reached from c_i by a single step of one or several processors. A sequence is *fairness* if any processor in G that is continuously enabled along the sequence, will eventually perform an action. *Maximality* means that the sequence is either infinite, or it is finite and in this later case no action of G is enabled in the final configuration. Let \mathcal{C} be the set of possible configurations and \mathcal{E} be the set of all possible computations of a system G . Then the set of computations of G starting with a particular *initial configuration* $c_1 \in \mathcal{C}$ will be denoted \mathcal{E}_{c_1} . Every computation $e \in \mathcal{E}_{c_1}$ is of the form c_1, c_2, \dots . The set of computations of \mathcal{E} whose initial configurations are all elements of $B \in \mathcal{C}$ is denoted as \mathcal{E}_B .

In this paper, we use the notion *attractor* [16] to define self-stabilization. intuitively, an attractor is a set of configurations of a system G that "attracts" another set of configurations of G for any computation of G .

Definition 1 (*Attractor*). Let B_1 and B_2 be subsets of \mathcal{C} . Then B_1 is an attractor for B_2 if and only if :

1. $\forall e \in \mathcal{E}_{B_2}, (e = c_1, c_2, \dots), \exists i \geq 1 : c_i \in B_1$ (*convergence*).
2. $\forall e \in \mathcal{E}_{B_1}, (e = c_1, c_2, \dots), \forall i \geq 1, c_i \in B_1$ (*closure*).

The set of configurations that matches the specification of problems is called the set of *legitimate* configurations, denoted as \mathcal{L} . In this paper, we treat only *static problems*, i.e., once the system reaches a desired configuration, the configuration remains unchanged forever. For example, the spanning-tree construction problem is a static problem, and the mutual exclusion problem is not a static problem [9]. $\mathcal{C} \setminus \mathcal{L}$ denotes the set of illegitimate configurations.

Definition 2 (Self-stabilization). A distributed system S is called self-stabilizing if and only if there exists a non-empty set $\mathcal{L} \subseteq \mathcal{C}$ such that the following conditions hold :

1. \mathcal{L} is an attractor for \mathcal{C} .
2. $\forall e \in \mathcal{E}_{\mathcal{L}}, e$ verifies the specification problem.

2.2 Clustering ad hoc network

Every node v in the *ad hoc* network is assigned an unique identifier (*ID*). For simplicity, here we identify each node with its *ID* and we denote both with v . We consider weighted networks, i.e., a weight w_v (a real number ≥ 0) is assigned to each node $v \in V$ of the network. In this paper we stipulate that each node has a different weight.

Clustering an ad hoc network means partitioning its nodes into *clusters*, each one with a *clusterhead* and (possibly) some *ordinary nodes*. The choice of the clusterheads is here based on the *weight* associated to each node : the bigger the weight of a node, the better that node for the role of clusterhead. In order to meet the requirements imposed by the wireless, mobile nature of these networks, a clustering algorithm is required to partition the nodes of the networks so that the following *ad hoc clustering properties* are satisfied :

1. Every ordinary node has at least a clusterhead as neighbor (*dominance* property)
2. Every ordinary node affiliates with the neighboring clusterhead that has the biggest weight
3. A clusterhead has not clusterhead neighbors (*independence* property)

3 A Self-stabilizing Clustering Algorithm

In the rest of this paper, we will refer to the guard of statement of process v as $G_i(v)$ and the rule of statement of process v as $R_i(v)$.

constants

$w_v : \mathbb{N}$; // the weight of node v

local variables of node v

Ch_v : boolean; // indicate that v is or is not a clusterhead

$Clusterhead_v$: IDs // the clusterhead of node v

Figure 2.1 : The algorithm on processor v

```

Do forever
   $G_1(v) \equiv (\forall z \in N_v : (Ch_z = F) \vee (w_v > w_z));$ 
   $G_2(v) \equiv (Ch_v = F) \vee (Clusterhead_v \neq v);$ 
   $G_3(v) \equiv (Ch_v = T) \vee (Clusterhead_v \neq \max_{w_z} \{z \in N_v : Ch_z = T\});$ 
   $R_1(v) : G_1(v) \wedge G_2(v) \rightarrow Ch_v := T; Clusterhead_v := v;$ 
   $R_2(v) : \neg G_1(v) \wedge G_3(v) \rightarrow Ch_v := F; Clusterhead_v := \max_{w_z} \{z : Ch_z = T\};$ 
od

```

3.1 Proof of convergence

Denote $Decided_i$, $i \in \mathbb{N}$ a set of nodes which have certainly selected the clusterhead at end of its step and this clusterhead stays in change. The convergence is done in step. During the i^{th} step, $\forall p \in Decided_i$ will choose their clusterhead. We define $Decided_i, i \in \mathbb{N}$ as the following recursive rule.

1 : $Decided_0 = \emptyset$.

2 : Denote v_{H_i} the node with the highest weight in $V - Decided_i$.

$Decided_{i+1} = Decided_i \cup \{v_{H_i} + N_{v_{H_i}}\}$.

We denote $L_i, L'_i, i \in \mathbb{N}$ a set of predicates on processor state. $L'_0 = True$. We will prove that at the end of i^{th} step, L'_{i+1} is verified.

Lemma 1 *Once L'_i is reached, then v_{H_i} becomes a clusterhead and stays forever a clusterhead.*

$(L'_{i+1} \equiv L'_i \text{ and } \{v_{H_i} \text{ is a clusterhead}\})$ is an attractor.

Proof : Notice that the guard $G_1(v_{H_i})$ is always verified because $w_{v_{H_i}} > w_z, \forall z \in N_{v_{H_i}}$. If v_{H_i} is not actually a clusterhead, then v_{H_i} verifies $G_1(v_{H_i})$ and $G_2(v_{H_i})$ up to the time where v_{H_i} performs the rule $R_1(v_{H_i})$. As all computations are fair v_{H_i} will eventually perform $R_1(v_{H_i})$. After the execution of $R_1(v_{H_i})$, v_{H_i} becomes a clusterhead. If v_{H_i} is actually a clusterhead then v_{H_i} will stay forever a clusterhead because v_{H_i} can never perform $R_2(v_{H_i})$. \square

Lemma 2 *Once L'_{i+1} is reached, all v_{H_i} 's neighbors in V_i choose v_{H_i} as their clusterhead and keep it.*

$(L'_{i+1} \equiv L_i \text{ and } \{\forall u \in (N_{v_{H_i}} \cap V_i) : Clusterhead_u = v_{H_i}\})$ is an attractor.

Proof : Let u be a v_{H_i} 's neighbor in V_i . Once L'_{i+1} is verified, u will never verify the guard $G_1(u)$ because $Ch_{v_{H_i}} = T$ and $w_{v_{H_i}} > w_u$. If $Clusterhead_u \neq v_{H_i}$ then u verifies $G_3(u)$, the rule $R_2(u)$ is enabled forever thus u will eventually perform $R_2(u)$. Once u have performed $R_2(u)$, the clusterhead of u is v_{H_i} , $R_2(u)$ becomes disabled and will never be enabled again. \square

Theorem 1 *The system reaches eventually a terminal configuration.*

Following Lemma 1 and Lemma 2, we have that there exists $k \in \mathbb{N} : Decided_k = V$. When $Decided_k = V$, no rule is executed in the system. Thus a terminal configuration is reached. \square

3.2 Proof of correctness.

Theorem 2 *Once the terminal configuration is reached, the ad hoc clustering properties are satisfied.*

Proof : In the terminal configuration, for every processor v we have $G_1(v) = T \wedge G_2(v) = F$ or $G_1(v) = F \wedge G_3(v) = F$.

Case 1. $G_1(v) = T \wedge G_2(v) = F$.

$G_2(v) = F$ means that v is a clusterhead. Hence, we need now to prove that v satisfies the *property 3*. Assume that there exists a processor $z \in N_v : Ch_z = T$.

Since $G_1(v) = T$ then $w_v > w_z$, thus $G_1(z) = F$. Since $R_2(z)$ is not executable, we have then $G_3(z) = F$. $G_3(z) = F$ implies that $Ch_z = F$, that is contrary. So there is no processor $z \in N_v : Ch_z = T$, thus v satisfies the *property 3*.

Case 2. $G_1(v) = F \wedge G_3(v) = F$.

$(G_1(v) = F) \equiv (\exists z \in N_v : (Ch_z = T) \wedge (w_z > w_v))$. Thus v has a clusterhead with bigger weight than its weight in its neighbors (*property 1 is verified*). $(G_3(v) = F) \equiv ((Ch_v = F) \wedge (Clusterhead_v = \max_{w_z} \{z \in N_v : Ch_z = T\}))$. That means v is an ordinary processor which affiliates with a clusterhead that has the biggest weight. Thus v satisfies the *property 2*. \square

4 A Generalized Self-stabilizing Clustering Algorithm

In the previous algorithm, we have requirement that the clusterheads are bound to never be neighbors. This implies that, when due to the mobility of the processors two or more clusterheads become neighbors, those with the smaller weights have to *resign* and affiliate with the now bigger neighboring clusterhead. Furthermore, when a clusterhead v becomes the neighbor of an ordinary processor u whose current clusterhead has weight smaller than v 's weight, u has to affiliate with (i.e., *switch* to the cluster of) v . These "resignation" and "switching" processes due to processor's mobility are a consistent part of the clustering management overhead that should be minimized. To overcome the above limitations, we introduce in this section a generalization of the previous algorithm. This algorithm is required to partition the nodes of the networks so that the following three requirements (called multi-hop clustering properties) are satisfied.

1. Every ordinary node always affiliates with (only) one clusterhead that has bigger weight than its weight (*affiliation condition*).
2. For every ordinary node v , for every clusterhead $z \in N_v : w_z \leq w_{Clusterhead_v} + h$ (*clusterhead condition*).
3. A clusterhead has at most k neighboring clusterheads (k being an integer, $0 \leq k < n$) (*k-neighborhood condition*).

The first requirement ensures that each ordinary node has direct access to at least one clusterhead (the one of the cluster to which it belongs), thus allowing fast intra and inter cluster communications. The second requirement guarantees that each ordinary node always stays with a clusterhead that gives it a "good" service. By varying the threshold parameter h it is possible to reduce the switching overhead associated to the passage of an ordinary node from its current clusterhead to a new neighboring one when it is not necessary. With this requirement we want to incur the switching overhead only when it is really convenient. When $h = 0$ we simply obtain that each ordinary node affiliates with the neighboring clusterhead with the biggest weight. Finally, the third requirement allows us to have up to k neighboring clusterheads, $0 \leq k < n$. When $k = 0$ we obtain that two clusterhead can not be neighbors. Notice that the case with $k = h = 0$ corresponds to the previous algorithm.

A Generalized Self-stabilizing Clustering Algorithm :

constants

$w_v : \mathbb{N}$; // the weight of node v

local variables of node v

Ch_v : boolean; // indicate that v is or is not a clusterhead.

$Clusterhead_v$: IDs // the clusterhead of node v .

SR_v : \mathbb{N} // the highest weight which violates the 3th condition in v 's neighbor.

Macros :

$N_v^+ = \{z \in N_v : (Ch_z = T) \wedge (w_z > w_v)\}$; // the set of v 's neighboring clusterhead that has bigger weight than v 's weight.

$Cl_v = |N_v^+|$; // the number of v 's neighboring clusterhead that has bigger weight than v 's weight.

Figure 2.2 : The algorithm on processor v

```

Do forever
   $\mathbf{G}_1(v) = \mathbf{G}_{11}(v) \vee \mathbf{G}_{12}(v)$ 
   $\mathbf{G}_{11}(v) \equiv [(Ch_v = F) \wedge (N_v^+ = \emptyset)]$ 
   $\mathbf{G}_{12}(v) \equiv [(Ch_v = T) \wedge (Clusterhead_v \neq v) \wedge (\forall z \in N_v^+ : w_v > SR_z) \wedge (Cl_v \leq k)]$ 
   $\mathbf{G}_2(v) = \mathbf{G}_{21}(v) \vee \mathbf{G}_{22}(v)$ 
   $\mathbf{G}_{21}(v) \equiv [(Ch_v = F) \wedge \{(\exists z \in N_v^+ : w_z > w_{Clusterhead_v} + h) \vee (Clusterhead_v \notin N_v^+)\}]$ 
   $\mathbf{G}_{22}(v) \equiv [(Ch_v = T) \wedge \{(\exists z \in N_v^+ : (w_v \leq SR_z)) \vee (Cl_v > k)\}]$ 
   $\mathbf{R}_1(v) : \mathbf{G}_1(v) \rightarrow Ch_v := T; Clusterhead_v := v;$ 
   $SR_v := \max(0, k + 1^{\text{th}}\{w_z : z \in N_v \wedge (Ch_z = T)\});$ 
   $\mathbf{R}_2(v) : \mathbf{G}_2(v) \rightarrow Ch_v := F; Clusterhead_v := \max_{w_z}\{z \in N_v : Ch_z = T\}; SR_v := 0;$ 
  // update the value of  $SR_v$ 
   $\mathbf{G}_3(v) \equiv (Ch_v = F) \wedge (SR_v \neq 0)$ 
   $\mathbf{G}_4(v) \equiv (Ch_v = T) \wedge (SR_v \neq \max(0, k + 1^{\text{th}}\{w_z : z \in N_v \wedge (Ch_z = T)\}))$ 
   $\mathbf{R}_3(v) : \mathbf{G}_3(v) \rightarrow SR_v := 0;$ 
   $\mathbf{R}_4(v) : \mathbf{G}_4(v) \rightarrow SR_v := \max(0, k + 1^{\text{th}}\{w_z : z \in N_v \wedge (Ch_z = T)\});$ 
od

```

4.1 Algorithm description

We split the possibles cases of a node v in the following mutually exclusive ones :

Case 1. v is an ordinary node. v has not a neighbor clusterhead whose weight is bigger than v 's weight. In this case, $G_{11}(v)$ is verified and v will become a clusterhead.

Case 2. v is a clusterhead. v does not violate the k -neighborhood condition but the value v 's clusterhead is incorrect. In this case, $G_{12}(v)$ is verified and v will correct the value of its clusterhead.

Case 3. v is an ordinary node and v violates the *clusterhead* condition. In this case, $G_{21}(v)$ is verified and v will become an ordinary node.

Case 4. v is a clusterhead and v violates the k -neighborhood condition. In this case, $G_{22}(v)$ is verified and v will become an ordinary node.

Once v becomes a clusterhead, v checks the number of its neighbors that are already clusterheads. If they exceed k , then v sets up the value of SR_v to the weight of the first clusterhead (namely, the one with the $(k+1)$ th biggest weight) that violates the k -neighborhood condition. Otherwise, SR_v is assigned to 0.

4.2 Proof of convergence

We first prove that the system reaches a terminal configuration.

Lemma 3 $A_1 = \{\mathcal{C} \mid \forall v : G_{12}(v) = F\}$ is an attractor.

Proof : If v verifies predicate $G_{12}(v)$ then v is enabled and will stay enabled up to the time where v performs $R_1(v)$. As all computations are fair, v eventually performs $R_1(v)$. After that $G_{12}(v)$ is never verified (see the rule action). \square

Lemma 4 In A_1 , once v had performed a rule $R_1(v)$ or $R_2(v)$, the guard of statements $G_i(v) : i = 1, 2$ remain false unless there exists a node u , $w_u > w_v$, that performs a rule $R_1(u)$ or $R_2(u)$.

Proof : In A_1 , $G_{12}(v)$ is never true.

Case 1. Once v had performed the rule $R_1(v)$, we have that $Ch_v = T$ and $Clusterhead_v = v$. Thus, the next rule performed by v will be $R_2(v)$.

Before doing $R_1(v)$, $G_{11}(v)$ is verified, we have $N_v^+ = \emptyset$. At time where v performs $R_2(v)$, $G_{22}(v)$ is verified, implies that $N_v^+ \neq \emptyset$, thus there is a node $u \in N_v$, $w_u > w_v$ that performed the rule $R_1(u)$ in meantime.

Case 2. Once v had performed the rule $R_2(v)$, we have that $Ch_v = F$ and $Clusterhead_v := \max_{w_z} \{z \in N_v : Ch_z = T\}$, next time v would perform a rule only if $G_{11}(v)$ or $G_{21}(v)$ is verified. Denote u the clusterhead of v , then after doing $R_2(v)$ we have $u \in N_v^+$ and $w_u = \max(w_z, \forall z \in N_v^+)$.

Case 2.1. v will performs $R_1(v)$ because $G_{11}(v)$ is verified. At time where v performs $R_1(v)$, $G_{11}(v)$ is verified then $N_v^+ = \emptyset$, implies that u performed the rule $R_2(u)$ in meantime.

Case 2.2. v will performs $R_2(v)$ because $G_{21}(v)$ is verified. G_{21} is verified, means that $(\exists z \in N_v^+ : w_z > w_u + h) \vee (u \notin N_v^+)$, implies that there exists a node $z \in N_v$, $w_z > w_u + h > w_u$ performed $R_1(z)$ or u performed $R_2(u)$ in meantime. \square

Lemma 5 $A_2 = A_1 \cup \{\mathcal{C} \mid \forall v : (G_1(v) = F) \wedge (G_2(v) = F)\}$ is an attractor.

Proof : We will prove by contradiction. Assume that A_2 is not an attractor. A processor cannot verify forever $G_1 \vee G_2$ (this processor would be enabled forever

and never performs a rule). Thus along a maximal computation there is a processor v that infinitely often verifies $G_1(v)$ or $G_2(v)$ and also infinitely often does not verify $G_1(v)$ or $G_2(v)$. Meaning that v executes infinitely often $R_1(v)$ or $R_2(v)$. Following Lemma 4, once v have performed a rule $R_1(v)$ or, $R_2(v)$ it would perform $R_1(v)$ or $R_2(v)$ again if there exists a processor u ($w_u > w_v$) that performs $R_1(u)$ or $R_2(u)$. Since the set of processors is finite, then v performs $R_1(v)$ or $R_2(v)$ infinity often only if there exists a processor u ($w_u > w_v$) that performs $R_1(u)$ or $R_2(u)$ infinity many times. Using a similar argument we have a infinite sequence of processors having increasing weight that performs R_1 or R_2 infinity often. Since the number of processors is finite, this is a contrary. Hence our hypothesis is false, and for every node v , $G_i(v) : i = 1, 2$ becomes eventually false and stay false. \square

Theorem 3 *The system reaches eventually a terminal configuration.*

Proof : By Lemma 5, A_2 is an attractor. In A_2 , processor v would only update of SR_v one time if necessary. \square

4.3 Proof of correctness.

Theorem 4 *Once a terminal configuration is reached, the multi-hop clustering properties are satisfied.*

Proof : In a terminal configuration, for every processor v , we have $G_i(v) = F : i = 1, 2$.

Case 1. v is an ordinary node.

$G_1(v) = F$ implies N_v^+ is not empty. $G_2(v) = F$ implies $(\nexists z \in N_v^+ : (w_z > w_{Clusterhead_v} + h))$ and $(Clusterhead_v \in N_v^+)$. Thus v satisfies property 1 and 2.

Case 2. v is a clusterhead node.

$(G_2(v) = F) \equiv (\forall z \in N_v^+ : w_v > SR_z) \wedge (Cl_v \leq k)$. $G_1(v) = F$ implies that $Clusterhead_v = v$. We now prove that v has at most k neighboring clusterheads. Since $Cl_v \leq k$, then v has at most k neighboring clusterheads with bigger weight than v 's weight. Assume that v has more than k neighboring clusterheads, thus there exists at least a neighboring clusterhead u of v such that $w_u \leq SR_v < w_v$. Hence, $G_{22}(u) = T$ because $v \in N_u^+(w_u \leq SR_v)$, that is a contrary. \square

Références

- [1] S. Banerjee and S. Khuller. A clustering scheme for hierarchical control in multi-hop wireless networks. In *INFOCOM 2001*, pages 1028–1037, 2001.
- [2] S. Basagni. Distributed and mobility-adaptive clustering for multimedia support in multi-hop wireless networks. In *VTC'99 : Proceedings of the IEEE 50th International Vehicular Technology Conference*, pages 889–893, 1999.
- [3] S. Basagni. Distributed clustering for ad hoc networks. In *I-SPAN'99 : Proceedings of the 1999 International Symposium on Parallel Architectures, Algorithms, and Networks*, pages 310–315, 1999.
- [4] J. Beauquier, M. Gradinariu, and C. Johnen. Memory space requirements for self-stabilizing leader election protocols. In *PODC '99 : Proceedings of the*

- eighteenth annual ACM symposium on Principles of distributed computing*, pages 199–207, 1999.
- [5] C. Bettstetter and B. Friedrich. Time and message complexities of the generalized distributed mobility-adaptive clustering (gdmac) algorithm in wireless multihop networks. In *VTC'03 : Proceedings IEEE Vehicular Technology Conference, Jeju, Korea*, pages 176–180, 2003.
 - [6] C. Bettstetter and R. Krausser. Scenario-based stability analysis of the distributed mobility-adaptive clustering (dmac) algorithm. In *MobiHoc'01 : Proceedings of the 2nd ACM Symposium on Mobile Ad Hoc Networking & Computing*, pages 232–241, 2001.
 - [7] M. Chatterjee, S. Das, and D. Turgut. Wca : A weighted clustering algorithm for mobile ad hoc networks”,. *Journal of Cluster Computing, Special issue on Mobile Ad hoc Networking*, 5(2) :193–204, 2002.
 - [8] E. W. Dijkstra. Selfstabilizing systems in spite of distributed control. *Comm. ACM*, 17, 11 :643–644, 1974.
 - [9] S. Dolev. *Self-Stabilization*. MIT Press, 2000.
 - [10] Y. Fernandess and D. Malkhi. K-clustering in wireless ad hoc networks. In *POMC '02 : Proceedings of the second ACM international workshop on Principles of mobile computing*, pages 31–37, 2002.
 - [11] M. Frodigh, P. Johansson, and P. Larsson. Wireless ad hoc networking : The art of networking without a network. In *Ericsson Review, No. 4*, 2000.
 - [12] M. Gerla and J. T. Tsai. Multiclustet, mobile, multimedia radio network. *Wireless Networks*, 1(3) :255–265, 1995.
 - [13] W. Goddard, S. T. Hedetniemi, D. P. Jacobs, and Pradip K. Srimani. Self-stabilizing protocols for maximal matching and maximal independent sets for ad hoc networks. In *WAPDCM'03 : 5th IPDPS Workshop on Advances in Parallel and Distributed Computational Models*, 2003.
 - [14] C. Johnen. Service time optimal self-stabilizing token circulation protocol on anonymous unidirectional rings. In *SRDS '02 : Proceedings of the 21st IEEE Symposium on Reliable Distributed Systems*, pages 80–89, 2002.
 - [15] C. Johnen, L. O. Alima, S. Tixeuil, and A. K. Datta. Self-stabilizing neighborhood synchronizer in tree networks. In *ICDCS '99 : Proceedings of the 19th IEEE International Conference on Distributed Computing Systems*, page 487, 1999.
 - [16] C. Johnen and S. Tixeuil. Route preserving stabilization. In *SSS'03 : Proceedings of the 6th International Symposium on Self-stabilizing System, Springer LNCS 2704*, pages 184–198, 2003.
 - [17] C. R. Lin and M. Gerla. Adaptive clustering for mobile wireless networks. *IEEE Journal on Selected Areas in Communications*, 15(7) :1265–1275, 1997.
 - [18] M. Schneider. Self-stabilization. *ACM Symposium Computing Surveys*, tm25 :45–67, 1993.
 - [19] Z. Xu, S. T. Hedetniemi, W. Goddard, and P. K. Srimani. A synchronous self-stabilizing minimal domination protocol in an arbitrary network graph. In *IWDC'03 : Proceedings of the 5th International Workshop on Distributed Computing, Springer LNCS 2918*, 2003.