Self-Stabilizing weight-based Clustering Algorithm for Ad hoc sensor 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

June 2, 2006

Abstract

Ad hoc sensor networks consist of large number of wireless sensors that communicate with each other in the absence of a fixed infrastructure. Fast self-reconfiguration and power efficiency are very important property on any sensor network management. The clustering problem consists in partitioning network nodes into groups called clusters, thus giving at the network a hierarchical organization. Clustering increase the scalability and the energy efficiency of communication among the sensors. A self-stabilizing algorithm, regardless of the initial system state, converges to a set of states that satisfy the problem specification without external intervention. Due to this property, self-stabilizing algorithms are adapted highly dynamic networks. In this paper we present a Self-stabilizing Clustering Algorithm for Ad hoc sensor network. Our algorithm adapts faster than other algorithms to topology changes.

Keywords: Self-stabilization, Distributed algorithm, Clustering, sensor network.

1 Introduction

An *ad hoc sensor* network consists of a number of sensors spread across a geographical area. Each sensor has wireless communication capability and some level of intelligence for signal processing and networking. Given the large number of nodes and their potential placement in hostile locations, it is essential that the network be able to self-organize; manual configuration is not feasible. Moreover, nodes may fail at any time (either from lack of energy or from physical destruction), and new nodes may join the network. Therefore, the network must be able to reconfigure itself so that it can continue to function. The lifetime of a sensor is determined by the battery life, thereby requiring the minimization of energy expenditure for network management procedure. Therefore, fast self-reconfiguration has to be the main feature of a sensor network management.

Clustering means partitioning network nodes into groups called clusters, giving to the network a hierarchical organization. A cluster is a connected graph composed of a clusterhead and (possibly)

some ordinary nodes. Each node belongs to only one cluster. In addition, a cluster is required to obey to certain constraints that are used for network management, routing methods, resource allocation, etc. By dividing the network into non-overlapped clusters, intra-cluster routing is administered by the clusterhead and inter-cluster routing can be done in reactive manner by clusterhead leaders and gateway. Clustering has the following advantages. First, clustering facilitates the reuse of resource, which can improve the system capacity. clustering-based routing reduces the amount of routing information propagated in the network. Clustering reduces 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. Clustering is vital for efficient ressource utilization and load balacing in large scale networks as sensor networks.

For these reasons, it is not surprising that several distributed clustering algorithms have been proposed during the last few years [12, 18, 2, 3, 1, 11, 8]. The clustering algorithms appeared in [1, 11] build a spanning tree. Then on top of the spanning tree, the clusters are constructed. In these papers, 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, 18] where nodes are organized into clusters. The built clusterheads set is an independent (i.e., clusterheads are not neighbors) and also a dominating set. The clusterheads are selected according to the value of their IDs. In [3], a Distributed and Mobility-Adaptive Clustering algorithm, called DMAC, is presented; the clusterheads are selected according to a node's parameter (called *weight*). The higher is the weight of a node, the more suitable this node is for the role of clusterhead. 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 neighbors, 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 papers [7, 6], with respect to their convergence time and message complexity. In [8], a weight-based distributed clustering algorithm is presented; also the computation of the node's weight according several parameters (node's degree, transmission power, battery power, ...). In [14, 23] probabilistic clustering constructions for ad hoc sensor network are presented.

In 1973, Dijkstra [9] 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 steps". 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 [21, 10]. 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 neighbors, 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 neighbor that is member of the set). In [22], 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) is presented. Notice that a minimal dominating set is not always an independent set.

Both algorithms DMAC and GDMAC are not self-stabilizing, i.e., they work assuming correct initialization. They cannot cope with the wake up problem. Sensors to conserve energy sleep a large portion of the time. During the sleeping period of a sensor, the network topology may have drastically changed. The sensor has to automatically adapt to the new situation. We present in this paper a self-stabilizing version of DMAC and GDMAC algorithm: they cope with any initial configuration. They also adapt to arbitrary topology changes due to node crash failures, communication link crash failures, node recovering or link recovering, merging of several networks, and so on.

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. The time complexity is analyzed in section 5. The paper ends with concluding remarks in section 6.

2 Model

2.1 Distributed System

In this paper, we consider the state model [4, 16, 15]. A distributed system $\mathcal{S}$ is a set of state machines called processors. Each processor can communicate with a subset of other 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 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 and any neighbors of i, but can only be modified by i. Each processor has a program and the 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 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 $c_1, c_2, ...$ such that for $i = 1, 2, ...$, the configuration c_{i+1} is reached from c_i by a single step of one or several processors. A computation is *fair* 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 $c_1, c_2, ...$ 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* [17] 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. $\forall e \in \mathcal{E}_{B2}, (e = c_1, c_2, ...), \exists i \geq 1 : c_i \in B_1$ *(convergence).*
2. $\forall e \in \mathcal{E}_{B_1}, (e = c_1, c_2, ...), \forall i \geq 1, c_i \in B_1$ *(closure).*

The set of configurations that matches the specification of problems is called the set of *legitimate* configurations, denoted as $\mathcal{L}$. $\mathcal{C} \backslash \mathcal{L}$ 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 for network

Every node v in the network is assigned an unique identifier (ID). For simplicity, here we identify each node with its ID and we denote both with v.

Clustering an ad hoc sensor network means partitioning its nodes into *clusters*, each one with a *clusterhead* and (possibly) some *ordinary nodes*. In order to meet the requirements imposed by the wireless, mobile nature of these networks, ordinary nodes has to be at distance 1 of their clusterhead. Thus, the following *clustering properties* have to be satisfied:

1. Every ordinary node has at least a clusterhead as neighbor (*dominance* property).
2. A clusterhead has not clusterhead neighbors (*independence* property).

We consider weighted networks, i.e., a weight w_v is assigned to each node $v \in V$ of the network. In ad hoc sensor networks, amount of bandwidth, memory space or battery power of a processor could be used to determine weight values.

The choice of the clusterheads will be based on the *weight* associated to each node: the higher the weight of a node, the better this node is suitable to be a clusterhead. Thus the following property has also to be verified:

3. Every ordinary node affiliates with the neighboring clusterhead that has the highest weight.

3 Self-stabilizing DMAC 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

Predicates

$G_1(v) \equiv (\forall z \in N_v : (Ch_z = F) \vee (w_v > w_z))$;

$G_2(v) \equiv (Ch_v = F) \vee (Clusterhead_v \neq v)$;

$G_3(v) \equiv (Ch_v = T) \vee (Clusterhead_v \neq max_{w_z}\{z \in N_v : Ch_z = T))$;

Rules

$R_1(v) : G_1(v) \wedge G_2(v) \rightarrow Ch_v := T; Clusterhead_v := v$;

$R_2(v) : \neg G_1(v) \wedge G_3(v) \rightarrow Ch_v := F; Clusterhead_v := max_{w_z}\{z : Ch_z = T\}$;

Algorithm 1 *Self-stabilizing DMAC algorithm*

v is a clusterhead iff $Ch_v = T$ otherwise v is an ordinary node. If v has not a clusterhead in its neighborhood whose weight is higher than its weight then v will become a clusterhead by performing the rule $R_1(v)$. Otherwise, v will be an ordinary node by performing the rule $R_2(v)$; v chooses its clusterhead by selecting the node having the highest weight among its neighbors which are clusterhead. If *Clusterhead* value on v is not correct according to v'status (i.e., clusterhead or ordinary) then a rule is enabled. The rule $R_1(v)$ (resp. $R_2(v)$.) is enabled if v is a clusterchead (resp. if v is not a clusterhead).

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 unchange. 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.*
($L_{i+1} \equiv L'_i$ ***and*** *$\{v_{H_i}$ 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 $R_2(v_{H_i})$. □

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.*
($L'_{i+1} \equiv L_i$ ***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. □

Theorem 1 *The system eventually reaches 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. □

3.2 Proof of correctness

Theorem 2 *Once the terminal configuration is reached, the 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 exits 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 higher 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 highest weight. Thus v satisfies the *property 2*. □

4 Self-stabilizing GDMAC 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 higher 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 in ad hoc sensor networks. To overcome the above limitations, we introduce in this section a generalization of the previous algorithm. This algorithm is used to partition the nodes of the networks so that the following three requirements (called ad-hoc sensor clustering properties) are satisfied.

1. Every ordinary node always affiliates with (only) one clusterhead which has higher 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 highest 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.

4.1 GDMAC algorithm description

Similarly to the algorithm 1, the rule R_1 sets up the performing node as a clusterhead; after the R_2 action, the performing node is ordinary. A clusterhead v checks the number of its neighbors that are clusterheads. If they exceed k, then it sets up the value of SR_v to the weight of the first clusterhead (namely, the one with the $(k+1)$th highest weight) that violates the k-neighborhood condition (R_4 action). Otherwise, SR_v is assigned to 0 (R_3 action). SR_v value of an ordinary node is 0 or R_3 is enabled to set the value to 0.

We split the possibles cases where a node v has to change its role (i.e., to become ordinary or clusterhead) in the following mutually exclusive ones:

Case 1. v is an ordinary node and v cannot select any neighbors as clusterhead - otherwise the affiliation condition will be violated -. $G_{11}(v)$ is verified. v will become a clusterhead (R_1 action).

Case 2. v is a clusterhead. v does not violate the k-neighborhood condition but the value v's clusterhead is incorrect. $G_{12}(v)$ is verified: v will correct the value of its clusterhead (R_1 action).

Case 3. v is an ordinary node and v violates the *clusterhead* condition. $G_{21}(v)$ is verified: v will become an ordinary node (R_2 action).

Case 4. v is a clusterhead and v violates the k-neighborhood condition. $G_{22}(v)$ is verified: v will become an ordinary node (R_2 action).

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 3^{th} 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 which has higher weight than v's weight.

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

Predicates

$\mathbf{G_1}(v) = \mathbf{G_{11}}(v) \vee \mathbf{G_{12}}(v)$

$\mathbf{G_{11}}(v) \equiv [(Ch_v = F) \wedge (N_v^+ = \emptyset)]$

$\mathbf{G_{12}}(v) \equiv [(Ch_v = T) \wedge (Clusterhead_v \neq v) \wedge (\forall z \in N_v^+ : w_v > SR_z) \wedge (Cl_v \leq k)]$

$\mathbf{G_2}(v) = \mathbf{G_{21}}(v) \vee \mathbf{G_{22}}(v)$

$\mathbf{G_{21}}(v) \equiv [(Ch_v = F) \wedge \{(\exists z \in N_v^+ : w_z > w_{Clusterhead_v} + h) \vee (Clusterhead_v \notin N_v^+)\}]$

$\mathbf{G_{22}}(v) \equiv [(Ch_v = T) \wedge \{(\exists z \in N_v^+ : (w_v \leq SR_z)) \vee (Cl_v > k)\}]$

$\mathbf{G_3}(v) \equiv (Ch_v = F) \wedge (SR_v \neq 0)$

$\mathbf{G_4}(v) \equiv (Ch_v = T) \wedge (SR_v \neq max(0, k + 1^{\text{th}}\{w_z : z \in N_v \wedge (Ch_z = T)\}))$

Rules

$\mathbf{R_1}(v) : \mathbf{G_1}(v) \rightarrow Ch_v := T; Clusterhead_v := v;$
$SR_v := max(0, k + 1^{\text{th}}\{w_z : z \in N_v \wedge (Ch_z = T)\});$

$\mathbf{R_2}(v) : \mathbf{G_2}(v) \rightarrow Ch_v := F; Clusterhead_v := max_{w_z}\{z \in N_v : Ch_z = T\}; SR_v := 0;$

// update the value of SR_v

$\mathbf{R_3}(v) : \mathbf{G_3}(v) \rightarrow SR_v := 0;$

$\mathbf{R_4}(v) : \mathbf{G_4}(v) \rightarrow SR_v := max(0, k + 1^{\text{th}}\{w_z : z \in N_v \wedge (Ch_z = T)\});$

Algorithm 2 *Self-stabilizing GDMAC algorithm*

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). □

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. □

Lemma 5 *$A_2 = A_1 \cup \{\mathcal{C} | \forall v : (G_1(v) = F) \wedge (G_2(v) = F)\}$ 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. □

Theorem 3 *The system eventually reaches 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. □

4.3 Proof of correctness.

Theorem 4 *Once a terminal configuration is reached, the ad-hoc sensor 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 higher weight than v's weight. Assume that v has more than k neighboring clusterheads, thus there exits 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. □

5 Time complexity

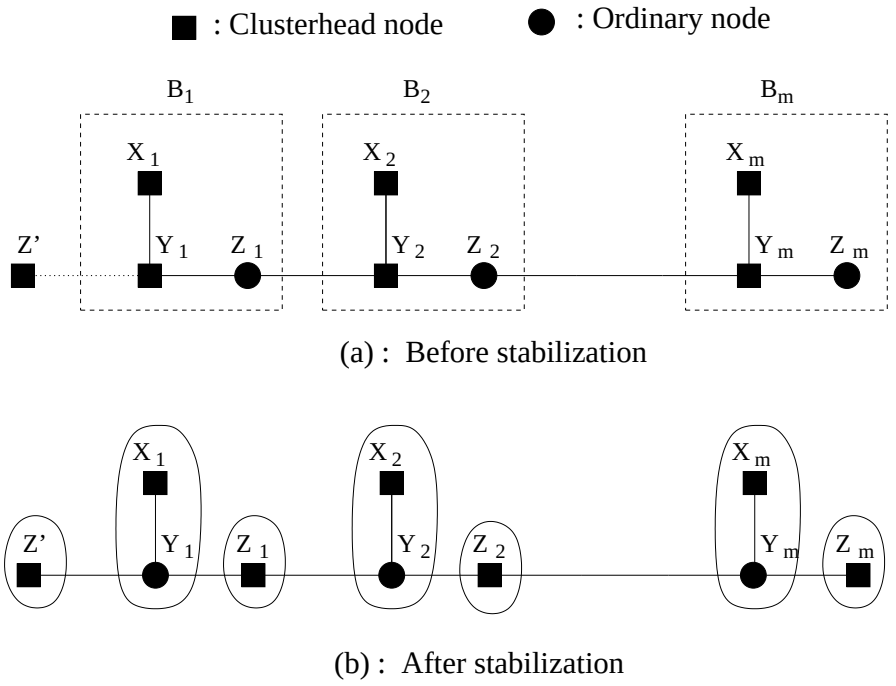

Figure 1: Stabilization time.

The stabilization time is the maximum number of rounds needed to reach a stabilized state from an arbitrary initial state. Figure 1 presents a scenario to measure stabilization time of GDMAC for $k = 1,\ h = 0$. Notice that the scenario can be easily generalized at any value of k and the starting configuration is the worst one. We have a system S composed by m blocs as depicted in Figure 1(a). Each bloc B_i includes two clusterheads X_i, Y_i and an ordinary node Z_i. We assume that the weight of nodes are ordered as the following: $X_i > Y_i > Z_i > Y_{i+1}$. A clusterhead node Z', $Z' > Y_1$ is also in S. We denote N the number of nodes in the system S, $N = m(k+2)+1$. Following Algorithm

2, each bloc B_i will one after another takes two rounds to reconstruct under the synchronous schedule. Thus, $2m$ rounds are needed to converge under the synchronous schedule. Then, the stabilization time is $O(2N/(k+2))$ in this example. We conclude that the worst stabilization time is $O(2N/(k+2))$. Notice that the worst stabilization time of DMAC is $O(N)$, same time that GDMAC when $k=0$.

6 Concluding remarks

The presented algorithms are designed for state model. Nevertheless, our algorithms can be easily transformed into algorithms for the message-passing model. Each node v periodically broadcasts to its neighbors its state (i.e., a message containing the values of Ch_v, SR_v, and w_v). Based on this message, v's neighbors decide to update or not their variables. After a change in the value of Ch_v, SR_v, or w_v, a node v broadcasts to its neighbors its new state.

Other self-stabilizing clustering algorithms have been designed. In [5], a self-stabilizing link-cluster algorithm under an asynchronous message-passing system model is presented (no convergence proofs are presented). The definition of cluster is not exactly the same as ours: an ordinary node can be at distance two of its clusterhead. The presented clustering algorithm requires three types of messages, our algorithms adapted to message passing model require one type of message.

A self-stabilizing algorithm for cluster formation is presented in [20]. A density criteria (defined in [19]) is used to select clusterhead: a node v chooses in its neighborhood the node having the highest density. A v's neighborhood contains all nodes at distance less or equal to 2 from v. Therefore, to choose clusterhead, communication at distance 2 is required. Our algorithms build clusters on local information; thus it requires only communication between nodes at distance 1 of each others.

We have presented in this paper a self-stabilizing version of DMAC and GDMAC algorithm: they cope with any initial configuration. They also adapt to arbitrary topology changes due to node crash failures, communication link crash failures, node recovering or link recovering, merging of several networks, and so on.

We have showed that the stabilization time of $GDMAC$ is $O(2N/(k+2))$. When $k=0$, the stabilization time is $O(N)$ as DMAC algorithm. By varying the threshold parameter k, the stabilization time can be reduced. Reducing the stabilization time is very important for ad hoc sensor networks: topology changes happen fairly often, conservation of sensor energy is a key factor. Thus network management procedure should be as simple as possible.

References

[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 multihop 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 *ISPAN'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] D. Bein, A. K. Datta, C. R. Jagganagari, and V. Villain. A self-stabilizing link-cluster algorithm in mobile ad hoc networks. In *ISPAN '05: Proceedings of the 8th International Symposium on Parallel Architectures, Algorithms and Networks*, pages 436–441, 2005.

[6] 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*, pages 176–180, 2003.

[7] 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.

[8] 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.

[9] E. W. Dijkstra. Selfstabilizing systems in spite of distributed control. *Comm. ACM, 17*, 11:643–644, 1974.

[10] S. Dolev. *Self-Stabilization.* MIT Press, 2000.

[11] 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.

[12] M. Gerla and J. T. Tsai. Multicluster, mobile, multimedia radio network. *Wireless Networks*, 1(3):255–265, 1995.

[13] W. Goddard, S. T. Hedetniemi, D. P. Jacobs, and P. 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] W. Heinzelman, A. Chandrakasan, and H. Balakrishnan. An application-specific protocol architecture for wireless microsensor networks. *IEEE Transactions on Wireless communications*, 1(4):660–670, 2002.

[15] 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.

[16] 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.

[17] 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.

[18] C. R. Lin and M. Gerla. Adaptive clustering for mobile wireless networks. *IEEE Journal on Selected Areas in Communications*, 15(7):1265–1275, 1997.

[19] N. Mitton, A. Busson, and E. Fleury. Self-organization in large scale ad hoc networks. In *The Third Annual Mediterranean Ad Hoc Networking Workshop, MED-HOC-NET 04*, June 2004.

[20] N. Mitton, E. Fleury, I. Guérin. Lassous, and S. Tixeuil. Self-stabilization in self-organized multihop wireless networks. In *WWAN'05: Proceedings of the 25th IEEE International Conference on Distributed Computing Systems Workshops*, pages 909–915, 2005.

[21] M. Schneider. Self-stabilization. *ACM Symposium Computing Surveys*, 25:45–67, 1993.

[22] 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.

[23] Ossama Younis and Sonia Fahmy. Distributed clustering for ad-hoc sensor networks: A hybrid, energy-efficient approach. In *Proceedings of IEEE INFOCOM*, 2004.