Robust Self-Stabilizing Clustering Algorithm

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

April 27, 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 network management. The clustering problem consists in partitioning network nodes into groups called clusters, thus giving at the network a hierarchical organization. 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 tolerate transient faults.

In this paper we present a robust self-stabilizing clustering algorithm for ad hoc network. The robustness property guarantees that, starting from an arbitrary state, in one round, network is partitioned into clusters. After that, network stays partitioned during the convergence toward a legitimate configuration where the clusters partition is optimal.

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

Résumé

Les réseaux ad hoc se composent d'hôtes qui communiquent les uns avec les autres en l'absence d'une infrastructure fixe. A eux de prendre en charge l'organisation du réseau (routage, gestion de la bande passante, connectivité). Un tel réseau ne peut pas compter sur la connectivité centralisée et organisée. Le problème clustering consiste à partitionner les noeuds d'un réseau en grappes, donc établissant une organisation hiérarchique au réseau. 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, sans intervention externe. Grâce à cette propriété, les algorithmes auto-stabilisants tolèrent les défaillances transitoires. Dans cet article nous présentons un algorithme auto-stabilisant robuste d'agrégation pour les réseaux ad hoc. La propriété de robustesse garantit que, à partir d'un état arbitraire, en un round, le réseau est partitionné en grappes. Après, le système évolue pour converger vers une configuration légitime où la partition est optimale.

Mots-clés: Auto-stabilization, Algorithme distribuée, Clustering, Réseau Ad hoc.

1 Introduction

An *ad hoc* network is a self-organized 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 [12]. 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 network management. 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 is also a typical example. Minimal configuration and quick deployment are needed in these situations.

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. Members of a cluster can share resources such 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 years [13, 19, 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 [13, 19] 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 [8], a weight-based distributed clustering algorithm taking into account several parameters (processor's degree, transmission and battery power, processor mobility) is presented. In a neighborhood, the processors elected are those that are the most suitable for 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 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 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 one. 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. [14] 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.

Several self-stabilizing algorithms for clusters formation and clusterheads selection have been proposed [5, 20, 17]. These algorithms are not robust. We present in this paper a robust version of GDMAC self-stabilizing algorithm [2]. Starting from an arbitrary state:

(1) the system satisfies a safety predicate in one round under the synchronous schedule; and

(2) once the system satisfied the safety predicate, it performs correctly its task; event during the stabilization phase where the system makes progress toward re-establishing the proper clusters.

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

The paper is organized as follows. In section 2, the formal definition of self-stabilization is presented. The clustering problem is discussed in the section 3. A robust version of [17] is described

in section 4. The self-stabilization proof is presented in section 5. Section 6 discusses about the robustness of our algorithm. Finally, the time complexity is analyzed in section 7.

2 Model

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. 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. 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 rule $Rule$ is enabled. 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 processor states. 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 computation, will eventually perform an action. *Maximality* means that the computation 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. The set of computations of G starting with the particular *initial configuration* $c \in \mathcal{C}$ will be denoted $\mathcal{E}_c$. 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* [18] to define self-stabilization.

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

One motivation for our robust stabilization is that a system should react gracefully to the changes of inputs - preserving a safety predicate in the presence of the changes of inputs. The safety predicate is chosen to ensure that the system still perform correctly its task during the period of convergence. A self-stabilizing protocol is robust with respect to changes of inputs, if starting from a legitimate state followed by changes of inputs, the safety predicate holds continuously until the protocol converges to a legitimate state.

Definition 3 *(**Robustness under Input Change [18]**). Let $\mathcal{SP}$ be a safety predicate, let $\mathcal{IC}$ be a set of changes of inputs in the system. A self-stabilizing distributed system $\mathcal{S}$ is robust under $\mathcal{IC}$ if and only if SC the set of configurations which satisfy $\mathcal{SP}$ verifies the following properties (i) SC is closed, and (ii) SC is closed under any change in $\mathcal{IC}$.*

3 Clustering for ad hoc network

Clustering an ad hoc 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, nodes in the same cluster has to be at distance at most 1 of their clusterhead. Thus, the following *clustering property* has to be satisfied:

1. Every ordinary node has at least a clusterhead as neighbor (*dominance* 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 networks, amount of bandwidth, memory space or battery power of a processor could be used to determine weight values. For simplicity, in this paper we assume that each node has a different weight. The choice of the clusterheads is 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.

Assume 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 network where the topology changes fairly often. To overcome the above limitations, in [2] Basagni introduced a generalization of the previous clustering property called *Ad hoc clustering properties* defined as follow:

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.

Safety property for clustering algorithm The safety property has to ensure that the network is partitioned into clusters and each cluster has a leader that performs clusterhead task. In a clustered network, the role of clusterhead is to act as a local coordinator within a cluster, performing information aggregation and exchange to neighboring clusters.

4 Robust Self-stabilizing Clustering Algorithm

In this section, we present a clustering algorithm (see Algorithm 1). This algorithm is self-stabilizing and robust to the input changes. Even during the stabilization phase, it is desired that network is correctly partitioned, i.e., each node belongs to only a cluster. This property, called "safety", guarantees functioning of the applications using the hierarchical structure established by Algorithm 1, because each node belongs to a cluster.

After the $R_1(v)$ action, v is a truly clusterhead ($Ch_v = T$). After the $R_2(v)$ action, v is an ordinary node ($Ch_v = F$). After the $R_3(v)$ action, v is a nearly ordinary node ($Ch_v = NF$).

A truly 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 ($R_5(v)$ action). Otherwise, SR_v is assigned to 0 ($R_4(v)$ action). SR_v value of an ordinary node is 0 or $R_4(v)$ is enabled to set the value to 0.

A truly clusterhead ($Ch_v = T$) has to resign its role iff it violates the k-neighborhood condition. A clusterhead v having to resign takes the nearly ordinary state ($Ch_v = NF$) - it performs $R_3(v)$ action. v stays in this nearly ordinary state until all of nodes in its cluster have joined another cluster.

A node v that has the state "nearly ordinary" is requiring that the member of its cluster join another cluster. Thus, the members of v's cluster are enabled (G_{11} or G_{21} predicate is verified), till v is nearly ordinary. As the scheduler is fair, the members of v's cluster will perform the rule R_1 or R_2. Thus, they will quit the v's cluster. At some time, v's cluster will contains one member: v. $(\forall z \in N_v : Clusterhead_z \neq v)$. At that time, v becomes an ordinary node (rule R_2) if v has at least a neighbor clusterhead whose weight is higher than v's weight. Otherwise, v becomes a clusterhead (rule R_1).

The safety predicate $\mathcal{SP}$ is defined as follow:

$$\mathcal{SP} \equiv \forall v \in V : (Clusterhead_v \in N_v \cup \{v\}) \wedge (Ch_{Clusterhead_v} \neq F).$$

$\mathcal{SP}$ predicate ensures that (i) each node belongs to a cluster and that (ii) the clusterheads are not ordinary nodes. As a nearly ordinary node and truly node acts as a clusterhead; each cluster has a clusterhead that performs its tasks correctly. Thus, the hierarchical structure exists if the $\mathcal{SP}$ is verified.

Due to an incorrect initial configuration, a node v has to correct the value of $Clusterhead_v$ and/or SR_v. In this case it verifies one of the following predicates: G_{12}, G_{32}, G_4.

Constants

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

Local variables of node v

Ch_v: {T, F, NF} // indicates the role of node v

$Clusterhead_v : IDs$ // 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 that has higher weight than v's weight

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

Predicates

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

$\mathbf{G_{11}}(v) \equiv (Ch_v \neq T) \wedge (N_v^+ = \emptyset)$

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

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

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

$\mathbf{G_{22}}(v) \equiv (Ch_v = NF) \wedge \{(\forall z \in N_v : Clusterhead_z \neq v) \wedge (N_v^+ \neq \emptyset)$

$\mathbf{G_3}(v) = \mathbf{G_{31}}(v) \vee \mathbf{G_{32}}(v)$

$\mathbf{G_{31}}(v) \equiv (Ch_v = T) \wedge \{(\exists z \in N_v^+ : (w_v \leq SR_z)) \vee (Cl_v > k)\}$

$\mathbf{G_{32}}(v) \equiv (Ch_v = NF) \wedge (Clusterhead_v \neq v)$

$\mathbf{G_4}(v) \equiv (Ch_v \neq T) \wedge (SR_v \neq 0)$

$\mathbf{G_5}(v) \equiv (Ch_v = T) \wedge (SR_v \neq max(0,\ k + 1^{\text{th}}\{w_z : z \in N_v \wedge (Ch_z = T)\}))$

Rules

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

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

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

// update the value of SR_v

$\mathbf{R_4}(v) : (\neg\mathbf{G_1}(v) \wedge \neg\mathbf{G_2}(v) \wedge \neg\mathbf{G_3}(v)) \wedge \mathbf{G_4}(v) \rightarrow SR_v := 0$

$\mathbf{R_5}(v) : (\neg\mathbf{G_1}(v) \wedge \neg\mathbf{G_2}(v) \wedge \neg\mathbf{G_3}(v)) \wedge \mathbf{G_5}(v) \rightarrow$
$SR_v := max(0,\ k + 1^{\text{th}}\{w_z : z \in N_v \wedge (Ch_z = T)\})$

Algorithm 1 *Robust Self-stabilizing Clustering Algorithm*

We split the possibles local state of node v needing a v's action in the following mutually exclusive ones:

Case 1. v is an ordinary node or a nearly ordinary and v cannot become an ordinary node - otherwise the affiliation condition will not be respected. $G_{11}(v)$ is verified, v will become a clusterhead (rule R_1).

Case 2. v is a clusterhead and 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 (rule R_1).

Case 3. v is an ordinary node and v violates the *clusterhead* condition. $G_{21}(v)$ is verified, v will select another neighbor as clusterhead (rule R_2).

Case 4. v is a nearly ordinary node and the safety predicate will be preserved if v becomes ordinary (i.e., none of v's neighbors selected v as their clusterhead). In addition, v can select one of its neighbors as clusterhead without violating the affiliation condition - v has at least a neighbor clusterhead whose weight is higher than v's weight. $G_{22}(v)$ is verified, v will become an ordinary node (rule R_2).

Case 5. v is a clusterhead and v violates the k-neighborhood condition. $G_{31}(v)$ is verified, v will become a nearly ordinary node (rule R_3).

Case 6. v is a nearly ordinary node but its clusterhead value is incorrect. $G_{32}(v)$ is verified, v will correct its clusterhead value (rule R_3).

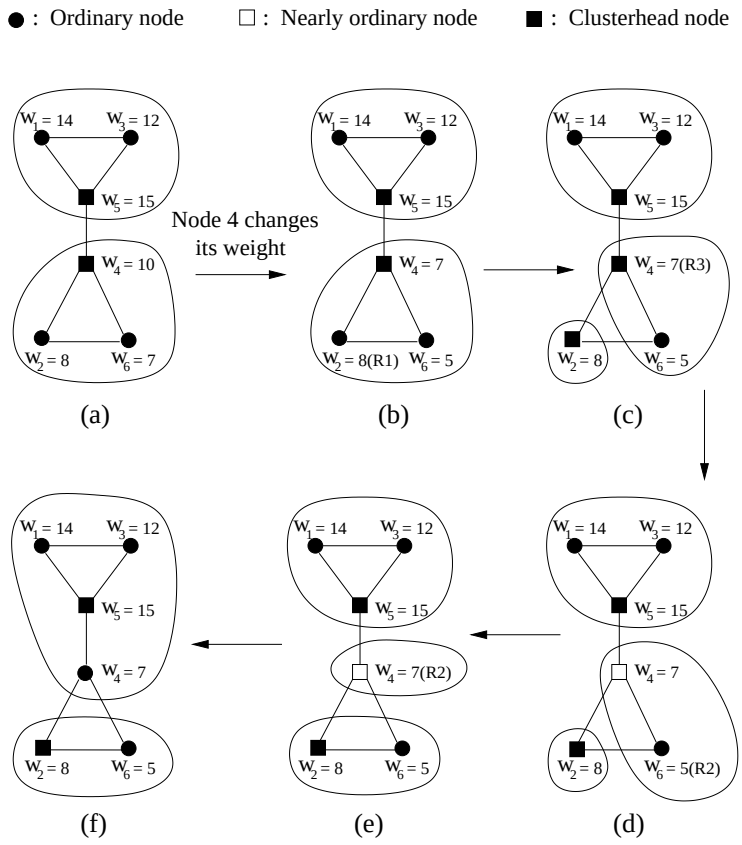

Figure 1: Convergence after a change of node's weight in the case k = 1, h = 0.

Algorithm 1 is illustrated in Figure 1. Initially we have a stabilized network (Figure 1(a)). Due to the change of the weight of node 4 (Figure 1(b)). Node 2 cannot stay ordinary because it would violate the affiliation condition; i.e., all neighbors of node 2 have a weight that is smaller than its weight. Node 2 becomes clusterhead (Figure 1(c)). Node 4 switches to nearly ordinary state (Figure 1(d)). It cannot stay a clusterhead because it would violate the 1-neighborhood condition: there are two clusterheads in its neighbor (node 2 and 5) that have a higher weight that its weight.

Node 6 cannot stay ordinary because it would violate the affiliation condition; i.e., it does not affiliate with a clusterhead. Node 6 chooses node 2 as its clusterhead (Figure 1(e)). After that, node 4 joins the cluster of node 5 (Figure 1(f)): the network is stabilized. During the convergence stage, the safety property $\mathcal{SP}$ is always verified: at any time the network is partitioned.

5 Proofs of self-stabilization

5.1 Proof of convergence

We first prove that the system reaches a terminal configuration.

Lemma 1 *$A_1 = \{C \mid \forall v : (G_{12}(v) = F) \wedge (G_{32}(v) = F)\}$ is an attractor.*

Proof: If v verifies predicate $G_{12}(v)$ (resp $G_{32}(v)$) then v is enabled and will stay enabled up to the time where v performs $R_1(v)$ (resp $R_3(v)$). As all computations are fair, v eventually performs $R_1(v)$ (resp $R_3(v)$). After that $G_{12}(v)$ (resp $G_{32}(v)$) is never verified. □

Lemma 2 *In A_1, once v had performed a rule $R_1(v)$, (v) does not perform $R_1(v)$, $R_2(v)$ or $R_3(v)$ until there exists a node u, $w_u > w_v$, that had performed $R_1(u)$.*

Proof: In A_1, $G_{12}(v)$ and $G_{32}(v) = F$ is never true.
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 $R_3(v)$.
Before doing $R_1(v)$, $G_{11}(v)$ is verified, we have $N_v^+ = \emptyset$. At time where v performs $R_3(v)$, $G_{31}(v)$ is verified, implies that $N_v^+ \neq \emptyset$. Thus in meantime, a node $u \in N_v$, $w_u > w_v$ performed the rule $R_1(u)$.

Lemma 3 *In A_1, once v had performed a rule $R_2(v)$, (v) does not perform $R_1(v)$, $R_2(v)$ or $R_3(v)$ until there exists a node u, $w_u > w_v$, that had performed a rule $R_1(u)$ or $R_3(u)$.*

Once v had performed the rule $R_2(v)$, we have that $Ch_v = F$ and $Clusterhead_v := max_{w_z}\{z \in N_v^+\}$. Denote u the clusterhead of v, we have $u \in N_v^+$ and $w_u = max_{w_z}\{z \in N_v^+\} > w_v$. Next time that v will perform a rule, $G_{11}(v)$ **or** $G_{21}(v)$ is verified.

Case 1. $G_{11}(v)$ is verified. At time where v performs $R_1(v)$, $N_v^+ = \emptyset$, implies that u performed the rule $R_3(u)$ in meantime.

Case 2. $G_{21}(v)$ is verified. We have $(\exists z \in N_v^+ : w_z > w_u + h) \vee (u \notin N_v^+)$, implies that in meantime u performed $R_3(u)$ or a node $z \in N_v$ such that $w_z > w_u + h > w_u$ performed $R_1(z)$. □

Lemma 4 *Let v be a node. The value of Ch_v cannot be NF forever.*

Proof: We prove by contradiction. Assume that $Ch_v = NF$ is verified forever. Assume that there is a node $u \in N_v$ such that $Clusterhead_u = v$.

Case 1. $Ch_u = F$. Since $Ch_v = NF$ then $Clusterhead_u \notin N_u^+$ (see the definition of N_u^+). Thus, $G_{21}(u)$ is verified. As all computations are fair, u will perform $R_2(u)$. After doing $R_2(u)$, $Clusterhead_u \neq v$ is verified forever.

Case 2. $Ch_u = T$. Since $Clusterhead_u = v \neq u$. Thus, $G_{12}(u)$ or $G_{31}(u)$ is verified. As all computations are fair, u will perform $R_1(u)$ or $R_3(u)$. After doing $R_1(u)$ or $R_3(u)$, $Clusterhead_u \neq v$ is verified forever.

Case 3. $Ch_u = NF$. Since $Clusterhead_u = v \neq u$. Thus, $G_{32}(u)$ is verified. As all computations are fair, u will perform $R_3(u)$. After doing $R_3(u)$, $Clusterhead_u \neq v$ is verified forever.

Therefore, $\forall u \in N_v,\ Clusterhead_u \neq v$ is verified. Thus, $G_{11}(v)$ or $G_{22}(v)$ is verified. As all computations are fair, v will perform $R_1(v)$ or $R_2(v)$. After doing $R_1(v)$ or $R_2(v)$, $Ch_v = NF$ is not verified. That is a contrary. □

Corollary 1 *In A_1, once v had performed a rule $R_3(v)$, v will certainly perform $R_1(v)$ or $R_2(v)$.*

Lemma 5 *Every fair computation e that starts in A_1 has a suffix where in any reached configuration $\forall v \in V : (G_i(v) = F),\ i = \{1..3\}$.*

Proof: We will prove by contradiction. Assume that e has not a suffix in which $\forall v \in V : (G_i(v) = F),\ i = \{1..3\}$. A processor cannot verify forever $G_1(v) \vee G_2(v) \vee G_3(v)$ (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 $G_1(v)$, $G_2(v)$ or $G_3(v)$ and also infinitely often does not verify $G_1(v)$, $G_2(v)$ and $G_3(v)$. Meaning that v executes infinitely often $R_1(v)$, $R_2(v)$ or $R_3(v)$. Following Corollary 1, if v executes infinitely often $R_3(v)$ then v executes also infinitely often $R_1(v)$ or $R_2(v)$. Following Lemma 2, 3 and 4, once v have performed a rule $R_1(v)$, $R_2(v)$ or $R_3(v)$, it will perform $R_1(v)$, $R_2(v)$ or $R_3(v)$ again if there exists a processor u ($w_u > w_v$) that performs $R_1(u)$, $R_2(u)$ or $R_3(u)$. Since the set of processors is finite, then v performs $R_1(v)$, $R_2(v)$ or $R_3(v)$ infinitely often only if there exists a processor u ($w_u > w_v$) that performs $R_1(u)$, $R_2(u)$ or $R_3(u)$ infinite many times. Using a similar argument we have a infinite sequence of processors having increasing weight that performs R_1, R_2 or R_3 infinitely often. Since the number of processors is finite, this is a contrary. Hence our hypothesis is false, and for every node v, $G_i(v) : i = 1, 2, 3$ becomes false forever. □

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

Proof: By Lemma 5, $G_i(v), i = \{1..3\}$ is not verified, processor v would only update of SR_v one time if necessary. When $G_i(v) = F,\ i = \{1..5\}$ for every node v, the system reaches a terminal configuration. □

5.2 Proof of correctness

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

Proof: In a terminal configuration, for every processor v, we have $G_i(v) = F :\ i = \{1..5\}$. Following Lemma 4, in a terminal configuration there is not a node v such that $Ch_v = NF$.
Case 1. $Ch_v = F$.
$G_1(v) = F$ 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. $Ch_v = T$.
$(G_2(v) = F) \equiv (\forall z \in N_v^+ : w_v > SR_z) \wedge (Cl_v \leq k)$. $G_1(v) = F$ 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. □

6 Robustness

On a configuration that satisfies $\mathcal{SP}$, the clusterhead of any node performs its task correctly, because it is not an ordinary node. Thus, the hierarchical structure is kept up. Let us remind the definition of $\mathcal{SP}$: $\mathcal{SP} \equiv \forall v \in V : (Clusterhead_v \in N_v \cup \{v\}) \wedge (Ch_{Clusterhead_v} \neq F)$.

Let v a processor. We define $\mathcal{SP}_v$ as the safety predicate $\mathcal{SP}$ on v.

Lemma 6 *$\mathcal{SP}_v$ is closed.*

Proof: Assume that we have a computation step $c_1 \xrightarrow{cs} c_2$, we will prove that if $\mathcal{SP}_v$ is verified in c_1, then in c_2, $\mathcal{SP}_v$ is verified.
We will prove by contrary. Assume that in c_2, $(Clusterhead_v \notin \{N_v \cup v\}) \vee (Ch_{Clusterhead_v} = F)$. Thus, in cs there are two possibilities.

Case 1. v changed its clusterhead during the execution cs. Notice that the rules R_4 and R_5 do not change the value of clusterhead of v. If v performs R_1 or R_3 in cs then $\mathcal{SP}_v$ is always verified because after doing R_1 or R_3, $(Clusterhead_v = v) \wedge (Ch_v \neq F)$. Thus, v performed R_2 during the execution of cs. We denote z the clusterhead selected by v in cs. In c_1, $Ch_z = T$ and in c_2, $Ch_z = F$. In cs, z cannot perform R_2. Thus, there is a contrary because R_2 is the only rule that changes the Ch_z value to F.

Case 2. v did not change its clusterhead during the execution of cs. Denote z the clusterhead of v. In c_1, $\mathcal{SP}_v$ is verified implies that $Ch_z \neq F$. In c_2, $\mathcal{SP}_(v)$ is not verified implies that $Ch_z = F$. Thus, during the execution cs, z performed R_2. But z can perform R_2 only when $G_{22}(z)$ is verified, that implies $Clusterhead_v \neq z$ in cs. That is a contrary. □

Theorem 3 *$\mathcal{SP}$ is closed.*

Proof: The theorem follows directly from Lemma 6. □

We denote z the clusterhead of node v. The safety predicate $\mathcal{SP}$ ensures that z is a neighbor of v and z is not an ordinary node. Thus, the safety predicate $\mathcal{SP}$ is only violated in cases of a z's removal (or a crash of z), a failure of link between v and z. Therefore, the safety predicate $\mathcal{SP}$ is preserved in the following cases:

1. Change of node's weight (illustrated in Figure 1).
2. Crash of ordinary nodes.
3. Joining of subnetworks that verify $\mathcal{SP}$ (illustrated in Figure 2).
4. Failures of link between two ordinary nodes or between two clusterhead nodes.

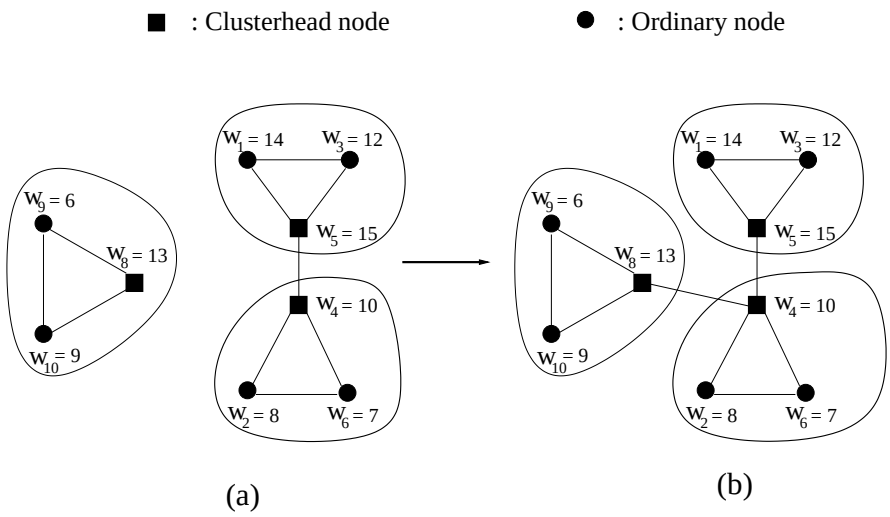

Figure 2: The joining of two stabilized subnetworks.

7 Time complexity

We consider synchronous computation, in which every process performs its code simultaneously. Thus, all enabled process perform a rule in a computation step.

Theorem 4 *The system verifies $\mathcal{SP}$ in one round under the synchronous schedule.*

Proof: Assume that we have a computation step $c_1 \xrightarrow{cs} c_2$. There are two possibilities:

Case 1. In c_1, $G_i(v) = F$, $\forall i \in \{1..3\}$. We denote $z = Clusterhead_v$ in c_1.

1. If $Ch_v = T$. Since $G_{12}(v)$ and $G_{31}(v)$ are not verified, that implies $z = v$, thus $\mathcal{SP}_v$ is verified in c_1.

2. If $Ch_v = F$. Since $G_{21}(v)$ is not verified, that implies $z \in N_v^+$, thus $\mathcal{SP}_v$ is verified in c_1.

3. If $Ch_v = NF$. Since $G_{11}(v)$ and $G_{22}(v)$ are not verified, that implies $z = v$, thus $\mathcal{SP}_v$ is verified in c_1.

Thus, in c_1, $\mathcal{SP}_v$ is verified. Since $\mathcal{SP}_v$ is closed (Lemma 6), then in c_2, $\mathcal{SP}_v$ is verified.

Case 2. In c_1, $\exists i \in \{1..3\} : G_i(v) = T$.

1. If $G_1(v) = T$. v will performs $R_1(v)$ in cs. After performing $R_1(v)$, $(Clusterhead_v = v) \wedge (Ch_v = T)$, thus $\mathcal{SP}_v$ is verified in c_2.

2. If $G_3(v) = T$. v will performs $R_3(v)$ in cs. After performing $R_3(v)$, $(Clusterhead_v = v) \wedge (Ch_v = NF)$, thus $\mathcal{SP}_v$ is verified in c_2.

3. If $G_2(v) = T$. v will performs $R_2(v)$ in cs. We denote z' the clusterhead selected by v in cs. Using the same argument in case 2 of Lemma 6: z' could not perform R_2 in cs. Therefore, $\mathcal{SP}_v$ is verified in c_2.

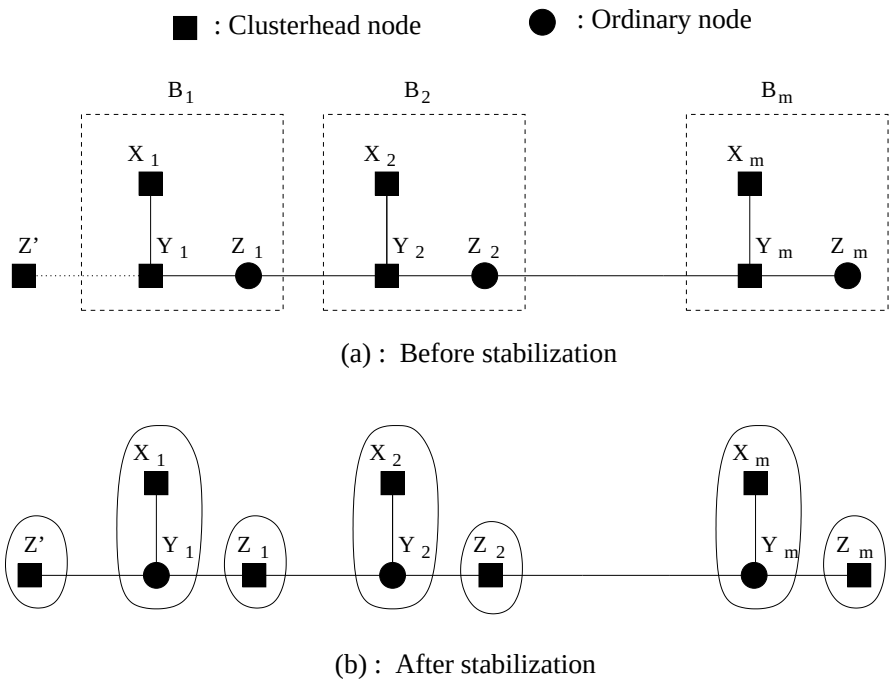

Figure 3: Stabilization time.

□

The stabilization time is the maximum number of rounds needed to reach a stabilized state from an arbitrary initial one. Figure 3 presents a scenario to measure stabilization time in the case $k = 1,\ h = 0$. Notice that this example can be generalized at any value of k and the initial configuration is the worst one. We have a configuration C composed by m blocs as depicted in Figure 3(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 a neighbor of Y_1. The largest convergence time under any weight-based clustering algorithm happens with this initial configuration. We denote N the number of nodes in the system S, $N = m(k+2)+1$. Following Algorithm 1, from the initial configuration, each bloc B_i will one after another takes two rounds to reconstruct. Thus, $2m+1$ rounds are needed to converge under the synchronous schedule. The stabilization time is $O(2N/(k+2))$.

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 multi-hop wireless networks. In *VTC'99: Proceedings of the IEEE* 50*th 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. Self-stabilizing 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. Frodigh, P. Johansson, and P. Larsson. Wireless ad hoc networking: The art of networking without a network. In *Ericsson Review, No. 4*, 2000.

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

[14] 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.

[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 L. Nguyen. Self-stabilizing clustering algorithm for ad hoc networks. *Technical Report no. 1357, L.R.I, Université de Paris Sud*, 1429, 2006.

[18] 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.

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

[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.