On the Self-stabilization of Mobile Robots in Graphs

Lélia Blin, Maria Gradinariu Potop-Butucaru, Sébastien Tixeuil

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

On the Self-stabilization of Mobile Robots in Graphs

Lélia Blin[†] — Maria Gradinariu Potop-Butucaru[⋆] — Sébastien Tixeuil[‡]

[†] Université d'Evry, IBISC-CNRS, France
[⋆] Université Pierre et Marie Curie-Paris 6, LIP6-CNRS, INRIA REGAL, France

[‡] Université Paris Sud-XI, LRI-CNRS, INRIA Grand Large, France

N° ????

July 2007

Thème NUM

Rapport de recherche

ISSN 0249-6399 ISRN INRIA/RR--????--FR+ENG

On the Self-stabilization of Mobile Robots in Graphs

Lélia Blin[†] , Maria Gradinariu Potop-Butucaru[⋆] , Sébastien Tixeuil[‡]

[†] Université d'Evry, IBISC-CNRS, France
[⋆] Université Pierre et Marie Curie-Paris 6, LIP6-CNRS, INRIA REGAL, France
[‡] Université Paris Sud-XI, LRI-CNRS, INRIA Grand Large, France

Thème NUM — Systèmes numériques
Projet Grand Large

Rapport de recherche n° ???? — July 2007 — 20 pages

Abstract: Self-stabilization is a versatile technique to withstand any transient fault in a distributed system. Mobile robots (or agents) are one of the emerging trends in distributed computing as they mimic autonomous biologic entities. The contribution of this paper is threefold. First, we present a new model for studying mobile entities in networks subject to transient faults. Our model differs from the classical robot model because robots have constraints about the paths they are allowed to follow, and from the classical agent model because the number of agents remains fixed throughout the execution of the protocol. Second, in this model, we study the possibility of designing self-stabilizing algorithms when those algorithms are run by mobile robots (or agents) evolving on a graph. We concentrate on the core building blocks of robot and agents problems: naming and leader election. Not surprisingly, when no constraints are given on the network graph topology and local execution model, both problems are impossible to solve. Finally, using minimal hypothesis with respect to impossibility results, we provide deterministic and probabilistic solutions to both problems, and show equivalence of these problems by an algorithmic reduction mechanism.

Key-words: mobile agents, mobile robots, graphs, self-stabilization, leader election, naming

Unité de recherche INRIA Futurs
Parc Club Orsay Université, ZAC des Vignes,
4, rue Jacques Monod, 91893 ORSAY Cedex (France)
Téléphone : +33 1 72 92 59 00 — Télécopie : +33 1 60 19 66 08

Sur l'Auto-stabilisation de Robots Mobiles dans un Graphe

Résumé : L'auto-stabilisation est une technique générique pour tolérer toute défaillance transitoire dans un système distribué. Les robots (ou agents) mobiles constituent l'un des modèles émergents de l'informatique distribuée du fait de leur ressemblance avec les entités biologiques autonomes. La contribution de cet article est triple. D'abord, nous présentons un nouveau modèle pour l'étude des entités mobiles dans des réseaux sujets à des défaillances transitoires. Notre modèle diffère du modèle classique des robots car les robots ont des contraintes sur les chemins qu'ils peuvent emprunter, et du modèle classique des agents mobiles car le nombre d'agents reste fixe pendant toute la durée de l'exécution du protocole. Ensuite, dans ce modèle, nous étudions la possibilité de l'existence d'algorithmes auto-stabilisants quand ces algorithmes sont exécutés par des robots mobiles évoluant dans un graphe. Nous nous concentrons sur les briques de bases des systèmes à base d'agents et de robots : le nomage et l'élection d'un chef. Conformément à l'intuition, quand aucune contrainte n'est faite sur le réseau et les paramètres de l'exécution, les deux problèmes sont impossibles à résoudre. Enfin, quand les hypothèses minimales pour résoudre ces deux problèmes sont disponibles, nous proposons des solutions déterministes et probabilistes pour les deux problèmes, et montrons que ces deux problèmes sont équivalents au moyen d'une réduction algorithmique.

Mots-clés : agents mobiles, robots mobiles, graphes, auto-stabilisation, élection d'un chef, nomage

1 Introduction

A large panel of recent research in Distributed Computing focused on solving problems using mobile entities, often denoted by the term of *robots* or *agents*. Those entities typically evolve in the network (that comprises a fixed set of nodes forming a particular topology) to provide services, either to the user of the network or to its core components. With the advent of large-scale networks that involve a total number of components in the order of the million, two particular issues were stressed: *(i)* the resources used by the agents should be kept to a minimum given a particular problem (see *e.g.* [16]), and *(ii)* the fault and attack tolerance capabilities are of premium importance. Most of the works on fault and attack tolerance with mobile agents deals with *external* threats, *i.e.* the faulty part of the system or the attacker is not an agent itself. For example, several papers (*e.g.* [10]) investigate the black hole search problem, where mobile entities must cooperate to find a hostile node of the network that destroys every mobile entity traversing it. In an orthogonal manner, decontamination and graph searching papers (*e.g.* [15]) consider the chasing of hostile mobile entities that are harmful to the nodes but not to the agents.

In this paper, we consider the novel problem of dealing with faults and attacks that hit the mobile entities themselves, that is, the threat is *internal*. More precisely, we consider that an arbitrary transient fault or attack hits the system (both nodes and mobile entities), and devise algorithmic solutions to recover from those faults and attacks. The faults and attacks are *transient* in the sense that there exists a point from which they don't appear any more. In practice, it is sufficient that the faults and attacks are sporadic enough for the network to provide useful services most of the time. In this context, *self-stabilization* [11] is an elegant approach to forward recovery from transient faults and attacks as well as initializing a large-scale system. Informally, a self-stabilizing systems is able to recover from any transient fault or attack in finite time, without restricting the nature or the span of those faults and attacks.

Related works Mobile (software) agents on graphs were studied in the context of self-stabilization *e.g.* in [17, 18, 6, 12], but the implicit model is completely different from ours. In the aforementioned works, agents are software entities that are exchanged through messages between processes (that are located in the nodes of the network), and thus can be destroyed, duplicated, and created at will. In [17, 6], a single agent is assumed at a given time, and this agent is responsible for stabilizing a simultaneously running (classical *aka* non-stabilizing) distributed algorithm. In [18],

exactly n agents are supposed to traverse a n sized tree network infinitely often, by means of a *swap* primitive that swaps agents located at two neighboring nodes. In [12], the authors consider dynamic evolving networks and rely on random walks to ensure proper agent traversal; again, the purpose of the agent protocol is to ensure that a single agent stabilizes the system. By contrast, in this paper, we focus on the self-stabilization of the agents themselves, and our model keeps the number of agents fixed for the whole life of the network. In a similar way, in [7] is studied the resource allocation problem in a model where agents (the resource consumers) are mobile and the resources form a fixed network. In this model agents join and leave the resource network at will. However, contrary to the curent work the model proposed in [7] assumes that agents and nodes have unique identifiers.

Self-stabilizing mobile (hardware) robots in 2-dimensional space were recently studied in *e.g.* [20, 19, 14]. Here a fixed set of k mobile robots are able to move unconstrained, yet are not able to communicate other than by viewing the relative position of other robots. The presented algorithms are oblivious in the sense that between any two activations of a particular robot, the previous state of the robot is cleared. As such, those algorithms are inherently self-stabilizing, since any scheduling for execution will reset the state of a robot. In this model several problems have been studied under different assumptions on the environment (schedulers, fault-tolerance, robots visibility, accuracy of compasses): circle formation, pattern formation, gathering, leader election, scattering. However, the lack of digital communication between the robots somewhat limits the kind of problems that can be solved and a broad class of impossibility results have been obtained [20, 19, 14, 8, 1]. In this paper, we introduce non-oblivious robots (agents) in the context of self-stabilization and enable digital communication between robots (either because they are located at the same node, or by using so-called whiteboards) to solve more elaborate tasks (*e.g.* naming and leader election), yet we restrict the motion capabilities of the robots (only edges of a given graph can be followed).

A third related model in the area of self-stabilization is that of Population Protocols (see *e.g.* [4, 5, 3]). In this model, finite-state agents interact in pairs chosen by an adversary, with both agents updating their state according to a joint transition function. For each such transition function, the resulting population protocol is said to stably compute a predicate on the initial states of the agents if, after sufficiently many interactions in a fair execution, all agents converge to having the correct value of the predicate. It was proved that this model permits to compute problems that can be expressed through Presburger arithmetic. Our model permits as well to express joint transition functions between agents located at the same node, but also

(indirectly) between agents that are hosted by the same node at different moments through the whiteboard abstraction.

Our contribution The contribution of this paper is threefold. First, we present a new model for studying mobile entities in networks subject to transient fault. Our model differs from the classical robot model because robots have constraints about the paths they are allowed to follow, and from the classical agent model because the number of agents remains fixed throughout the execution of the protocol. Second, in this model, we study the possibility of designing self-stabilizing algorithms when those algorithms are run by mobile robots (or agents) evolving on a graph. We concentrate on the core building blocks of robot and agents problems: naming and leader election. Not surprisingly, when no constraints are given on the network graph topology and local execution model, both problems are impossible to solve. Finally, using minimal hypotheses with respect to impossibility results, we provide deterministic and probabilistic solutions to both problems, and show equivalence of these problems by an algorithmic reduction mechanism. From a theoretical perspective, our results complement the widely known possible *vs.* impossible problems in anonymous distributed systems (see *e.g.* [22, 23]). From a practical perspective, our symmetry breaking algorithm enables to solve other problems such as gathering (see [9]) that have known solutions when mobile entities have unique identifiers.

Outline In Section 2, we present the computing model that is introduced in this paper. Section 3 provides impossibility results that justify the assumptions made in subsequent sections. Section 4 presents a deterministic algorithm for naming in acyclic networks with half-duplex links, while Section 5 provides a probabilistic naming algorithm for general networks. Section 6 shows that the naming and leader election problem are equivalent (by reduction of one problem to the other). Concluding remarks are presented in Section 7.

2 Model

The network is modeled as a connected graph $G = (V, E)$, where V is a set of nodes, and E is a set of edges. We assume that nodes have local distinct labels for links, but no assumption is made about the labeling process. Nodes also maintain a so-called *whiteboard* which can store a fixed amount of information. We distinguish between *acyclic* networks (*i.e.* trees) and *cyclic* networks (*i.e.* that contain at least one cycle).

Agents (or *robots*) are entities that move between neighboring nodes in the network. A link is *full-duplex* if two agents located at neighboring nodes can exchange their position at the same time, crossing the same link in opposite directions. A link is *half-duplex* if only one direction can be used at a given time. We assume that k agents are present in the network at any time. Also, each agent is modeled by an automata whose state space is sufficient to hold a identifier that is unique in the network (*i.e.* the state space is at least k states). An agent may move from one node to one of the node's neighbors based on the following information: *(i)* the current state of the agent, *(ii)* the current state of other agents located at the same node, *(iii)* the local link labels of the current node (and possibly the label of the incoming link used by the agent to reach the node), and *(iv)* the memory stored at the node (the whiteboard).

A *configuration* of the system is the product of all agents locations and states and all whiteboards contents. The behavior of the system is essentially *asynchronous*, in the sense that there is no bound on the number of moves that an agent can make between any two moves of another agent. There is one notable exception: when two (or more) agents are at the same node, the execution order is decided by the host node. In the following we assume that no two agents located at the same node execute their actions concurrently. However, agents located at different nodes may execute their actions concurrently. So, in any configuration of the system, the scheduler may chose any subset of nodes that hold at least one agent: then, in one atomic step all chosen nodes execute the code of all their host agents. The pseudo-code of each node (unless otherwise stated) is formally presented as Algorithm 1.

```
foreach agent on node
  agent.execute()
end foreach
```

Algorithm 1: Pseudo-code at node i

We assume that the scheduler is *fair* in the sense that if a node holds at least one agent, it is eventually scheduled for execution by the scheduler. A *round* starting from configuration c is the minimum time until all nodes that hold at least one agent in c have been activated by the scheduler. We now define the naming and leader election problems that we focus on in this paper:

Definition 1 (Naming) *Let S be a system with k agents. The system S satisfies the naming specification if all k agents eventually have an unique identifier (no two agents in S share the same identifier).*

In the leader election problem agents have either the leader role or the follower role.

Definition 2 (Leader Election) *Let S be a system with k agents. The system S satisfies the leader election specification if an unique agent eventually has the leader role and all the agents agents have the follower role.*

Our goal is to withstand transient failures. For this purpose, we assume that every "changing" aspect of the network can be arbitrarily modified in the initial configuration of the system. These "changing" aspects include: *(i)* the agent states, *(ii)* the agent positions, *(iii)* the whiteboard states.

Definition 3 (Self-stabilizing Problem) *Let S be a system with k agents. The system S satisfies the naming or the leader election specification in a self-stabilizing way if Definition 1 respectively Definition 2 are verified in spite of an arbitrary initial configuration.*

3 Impossibility results

The results in this section are negative. We consider networks where agents have infinite memory, and nodes have whiteboards with infinite memory. Also, the scheduling is constrained in the sense that at every step, all nodes that hold at least one agent are scheduled for execution. With our assumptions, the initial memory of every agent is supposed to be identical, and the initial content of each whiteboard is supposed to be identical as well. Agents are anonymous and deterministic.

Theorem 1 *Deterministic naming or leader election of mobile agents is impossible in cyclic networks, even assuming synchronous scheduling, and infinite memory for each agent and whiteboard.*

Proof: Assume the topology of the network is a cycle. The proof idea follows the lines of impossibility results found in [2]. Intuitively, assume a cyclic network in a symmetric initial configuration with two agents. Assume the agents have both the same identifier (or leader) variable and all the whiteboards in the network are initialized with the same arbitrary values. Since agents execute the same deterministic

algorithm, there exists an execution of the system refereed in the following as e such that all configurations appearing in this execution are symmetrical with respect to the agents view. Since a configuration solving the naming problem is asymmetrical with respect to the agents view, and the fact that e does not contain asymmetrical configurations, there exists an execution of the system that never reaches a naming or leader election configuration.

In the following we construct a symmetric execution e with respect to the agents view. Without restraining the generality assume only two agents in the network placed such that agents have exactly the same view and the same initial state s_0. Since the two agents have the same view, start in identical states, and execute the same deterministic algorithm, they execute the same action. So, both agents will reach exactly the same state s_0 (in case the agents do not change their state) or $s_1 \neq s_0$. In the new state the agents have the same view of the network and the same state so they execute again the same action. Either, the two agents keep the same state or they both change to a new state s_2. The agents can repeat this game infinitely often. Overall, there is an infinite execution where the two agents pass exactly through the same states in the same time $(s_0)^*(s_1)^*(s_2)^* \ldots$ and the system never reaches an asymmetrical configuration. □

Theorem 2 *Deterministic naming or leader election of mobile agents is impossible in acyclic networks with full-duplex links, even assuming synchronous scheduling, and infinite memory for each agent and whiteboard.*

Proof: Consider a network consisting of two nodes u and v linked by edge e. Assume that there is an agent at each node. Initially all agents are in the same state, and all nodes whiteboards are in the same state. Also, the local labeling of edges is the same at each node. So, the network is completely symmetric, from an agent point of view. Now, each time an agent is scheduled for execution, it may update its own state, update the whiteboard, or (inclusive) move to the other node. Now assume that the scheduling of the agents is synchronous, this means that at every step, the state of each agent remains identical, the state of each whiteboard remains identical, and the relative position of each agent with respect to the view of the other agent remains the same. Overall, symmetry can not be broken by a deterministic agent algorithm, and naming or leader election can not be achieved. □

4 Self-stabilizing deterministic naming in acyclic networks with half-duplex links

In the following we propose a deterministic self-stabilizing algorithm for agents naming in acyclic networks with half-duplex links. In networks with k agents the algorithm uses $O(\log(k))$ bits memory per agent and $O(k\log(\Delta k))$ per node, where Δ is the maximum degree of the network.

Each agent has a state that includes a software identifier `id` ("software" meaning that this identifier can be corrupted), that is represented by some integer. Each node has a whiteboard (that can be corrupted as well) that can store up to k 2-tuples $\langle \texttt{id}, \texttt{edge} \rangle$. The `id` part of the 2-tuple denotes the integer identifier of an agent, and the `edge` part of the 2-tuple denotes a local edge identifier. This whiteboard is meant to represent the identifiers of the latest k agents with distinct identifiers that visited the node, along with the corresponding outgoing edge they took last time they visited the node. Each node provides to the agents that visit the node some helper functions to access the whiteboard:

- `edge`(i) returns the edge that is associated to i if i is present in the whiteboard, and *nil* if i is not in the whiteboard.
- `visit`(i, j) sets the edge j to be associated to agent i if i is in the whiteboard, or adds the entry (i, j) to the whiteboard if i is not present. If the whiteboard already contains k tuples with distinct identifiers, the least recently updated one is dropped from the whiteboard. After updating the whiteboard, the agent exits the node through port j.
- `new` returns a new identifier that does not exists in the whiteboard.

When arriving at a node, an agent checks whether the node has its identifier in its whiteboard. If it is not present, it adds its identifier and erases one of the identifiers if there are more than k identifiers on the node's whiteboard (assuming FIFO order). If it is present, then either there is another agent with the same identifier at the current node, or the agent is the only one with its identifier. In the first case, the first agent to be executed by the node simply picks up a new identifier, and initiates a Eulerian traversal. In the second case, the agent checks if the last outgoing edge followed by an agent with its identifier is the same as the current incoming edge. In the case it points to another edge (which means there is a discrepancy, whatever its cause), the node simply follows the path corresponding to its identifier, trying to confront the other agent with the same identifier (if such agent exists). Finally,

when an agent enters a node through the expected edge, it continues performing a Eulerian tour of the tree, *e.g.* by choosing the outgoing edge that is next in the Eulerian tour, *i.e.* edge $(j+1) \bmod \delta$, if j was the incoming edge. Formally, the algorithm that is executed by each agent is presented as Algorithm 2.

```
id: integer
execute() {
  if ( edge(id) == nil )
    visit( id, 0 ) // add self, will exit through port 0 by default
  else if ( exists agent j on node i such that j.id == id )
    id = new // Not present or somebody else has the same id
    visit( id, 0 ) // add self, will exit through port 0 by default
  else if ( edge(id) != incoming edge ) )
    visit( id, edge( id ) ) // Follow possible conflicting agent
  else
    visit( id, edge( id ) + 1 mod delta ) // continue Eulerian traversal
}
```

Algorithm 2: Deterministic agent code for tree networks

We prove self-stabilization by defining a predicate for *legitimate* configurations, then proving *(i)* every computation starting from a legitimate configuration is correct (see Appendix), and *(ii)* every computation starting from an arbitrary configuration eventually reaches a legitimate configuration.

Definition 4 (Legitimate configuration) *A legitimate configuration for Algorithm 2 satisfies the three following properties:* (i) *all agents have distinct "software" identifiers,* (ii) *all whiteboards tuples contain only actual agent identifiers, and* (iii) *all whiteboards tuples are consistent with actual agent locations.*

In the following we prove that starting from any configuration, the system converges to a legitimate configuration.

Lemma 1 *An agent with identifier i eventually visits every node infinitely often.*

Proof: Assume the contrary, *i.e.* there exists a time in the execution where at least one node u never gets visited by any agent with identifier i. In turn, this means that for every neighbor v of u, either v is never visited by an agent with identifier i

(and the argument can be repeated on the neighbors of v), or v is visited infinitely often but the agent never takes the edge toward u (shortened as e_u). The only way for an agent a with identifier i not to take e_u is *(i)* to exit through edge number 0 (and that edge is not e_u), *(ii)* to follow the path of a (supposedly) other agent with identifier i that did not take e_u, or *(iii)* to never take e_u by performing a Eulerian traversal of the graph (*i.e.* the agent never arrives by port $(e_u - 1 \bmod \delta)$). Cases *(i)* and *(ii)* can be executed only a finite number of times (since there are k agents), so this implies that the node is visited infinitely often by agents that properly execute the $(\texttt{edge}(i) + 1 \bmod \delta)$ rule and never exit through e_u. Since the network is acyclic, this last case is impossible. □

Lemma 2 *If two agents have the same identifier they eventually meet within $O(m)$ rounds, where m is the number of edges of the graph.*

Proof: Note that a node that keeps its identifier either follows the apparent path of a supposed other node, or performs a Eulerian traversal of the tree. Intuitively, the proof goes as follows. The apparent path may be either fake (it leads to a node that does not have identifier i in its whiteboard, or that does not contain another agent with identifier i) or real (the path leads to an agent whose identifier is i). If the path is fake, it is nevertheless finite, and the agent will perform only a finite number of steps to reach the end of the path and realize it is fake. When an agent realizes a path is fake, it executes the Eulerian traversal algorithm. When a node switches to the Eulerian traversal algorithm, its path becomes real. Now, if the path is real, the agent chases a real agent in an acyclic network, and the path information is correct. Since the network is acyclic and the links are half duplex, the two agents are bound to meet each other.

An agent follows a fake path when the information on whiteboards erroneously indicates the presence of another agent with the same identifier. In order to check and correct the information in the whiteboards an agents, in the worst case, has to visit every node in network. In order to perform the traversal, each edge is visited at most twice. Hence the complexity of the traversal is $O(m)$. □

Note that after an agent visited each node of the graph at least once, all whiteboards are coherent with the agent direction and identifier.

Lemma 3 (Convergence) *Starting from any arbitrary initial configuration with k agents, any computation eventually reaches a legitimate configuration in $O(km)$ rounds.*

Proof: First, we observe that no identifier that is initially present in the network on some agent is ever removed from the network. This is due to the fact that an agent only changes its identifier when observing that another agent at the same node has the exact same identifier. Since agents execute their code sequentially (activated by nodes), the first activated agent with a conflicting identifier change its identifier, and the other agent remain unchanged (unless there are more that two agents at the same node with the same identifier).

Now, we prove that starting from any initial configuration, the number of distinct identifiers only increases until it reaches k. Initially the number of distinct identifiers is at least 1. Suppose that there exists some integer j $(1 \leq j < k)$ such that there exists j distinct identifiers in the network. Now, after finite time, $O(jm)$, all j identifiers are present in each whiteboard in the network (see Lemma 2). Since $j < k$, there exist at least two agents with the same identifier. By the above argument, two agents with the same identifier are to meet within finite time, $O(m)$. When this is done, one of the agents will change its identifier to a new identifier. Since all j identifiers are in the whiteboards and are regularly refreshed, a new identifier (not in the existing j set) will be solicited by the agent, and the number of total identifiers in the network rises to $j+1$. By induction hypothesis, the number of distinct identifier eventually reaches k after $O(km)$ rounds.

When all agents have distinct identifiers, they all traverse all the network infinitely often (see Lemma 1). When each of them has traversed the network at least once, the paths to the agents are all correct with respect to their current position. As a consequence, the configuration is legitimate. □

Lemma 4 (Correctness) *Every computation of Algorithm 2 that starts from a legitimate configuration satisfies the Naming problem specification.*

Proof: Since all k agents have distinct identifiers, and all k identifiers are present in all whiteboards, the whiteboards do not contain any spurious identifier information. So, an agent arriving at a node always finds its own identifier, with proper incoming edge. As a result, agents never change identifier, whiteboards never drop existing identifiers, and edge information is kept accurate, so that every agent performs Eulerian traversal of the tree forever. □

5 Probabilistic naming in arbitrary networks

In this section we assume an weaker model where agents cannot communicate via whiteboard and the network is arbitrary with full-duplex links. Theorems 1 and 2

provide impossibility results related to deterministic naming in this model. In the following we show the possibility of probabilistic naming. The idea is to make every agent randomly move in the network. Anytime two agents that are located at the same node have the same identifier, each agent randomly chooses a new identifier. If there are several agents at the same node with distinct identifiers, the random walk is continued.

Each node provides to the agents that visit the node some helper functions:

- `random`(S) returns a random element from set S.
- `visit`(j) makes the agent exit the node through port j.

```
id: integer
execute() {
   if (there exists agent a such that a.id = id at node i)
       id := random( 1..k )
   else
       visit( random( 0..degree(i) )
}
```

Algorithm 3: Probabilistic agent code executed at node i for arbitrary networks

Algorithm 3 presents the core of our algorithm for probabilistic agent naming. In the presentation, a random value for the identifier is assumed to be between 1 and k, but an upper bound on k may also be used to boost stabilization time (*e.g.* k^2). The proof of correctness can be found in the appendix.

Definition 5 (Legitimate configuration) *A configuration is legitimate if and only if all agents have distinct identifiers.*

Lemma 5 (Convergence) *Starting from an arbitrary initial configuration, the network eventually reaches a legitimate configuration. The expected stabilization time is $O(kn^3)$ where k is the number of agents in the network.*

Proof: We first make the two following observations:

1. when two agents with two different identifiers meet at the same node in the network, their random walk is unaffected by the meeting;

2. when two agents with the same identifier meet at the same node in the network, they stop moving until at least one of them randomly picked a new identifier.

In an arbitrary initial configuration, a pair of agents (u, v) in the network may share the same identifier. We consider occurrences of meetings of two agents or more at the same node in the network. When random walks are unbiased, the meeting time between any two agents is $O(n^3)$, [21]. Here we do not consider the meeting time between agents with different identifiers. Instead we consider the first occurrence of a meeting involving two or more agents of the same color. When this occurs, the two agents draw a random coin and get a random identifier. With probability at least $\frac{1}{k}$, the drawn fresh random identifier is unique in the whole network. So, anytime two agents with the same identifier meet, there is a positive probability that they both get a unique identifier in the system. Anytime this happens, the number of agents who share their color with at least one other agent decreases by one. As a result, with probability one, a configuration where all agents have unique identifiers is reached, and remains thereafter. The stabilization time $O(kn^3)$. □

Lemma 6 (Correctness) *Every computation of Algorithm 3 that starts from a legitimate configuration satisfies the naming problem specification.*

Proof: Assume all identifiers are distinct for all agents, then the "if" clause is never falsified, so the identifier of the agent is never changed. As a result, the configuration remains legitimate. □

6 Naming and leader election

In this section, we consider the relationship between the aforementioned naming problem, and the leader election problem, where the network must eventually reach a configuration where exactly one agent is elected and all others are non-elected.

6.1 From naming to leader election

We first observe that given a naming of k robots in the network, it is easy to come up with a leader election protocol.

1. In our deterministic protocol, whiteboards are used to register all identifiers used in the system by the agents. When an agent arrives at some node, it checks from the whiteboard if its identifier is maximum in the whiteboard, and becomes elected if so. If its identifier is not maximum in the node's

whiteboard, the agent becomes non-elected. After stabilization of the naming algorithm, all whiteboards contain the exact identifiers used in the network, which means that all whiteboards contain the same identifiers in the system. So, if an agent has maximal identifier on one whiteboard, it has maximum identifier on all whiteboards. This guarantees the correctness of the leader election protocol.

2. In our probabilistic protocol, eventually all nodes have distinct identifiers in the network. If the exact value of k was used in the algorithm, then a node can simply checks its identifier against k to detect if it is the leader or not. If only an upper bound on k was used, then a more complicated process is required. The procedure is as follows: each node stores the identifiers of the last $k-1$ distinct agents it last saw; then if its identifier is maximum among those identifiers, it becomes elected, and remains non-elected otherwise. The leader status is updated anytime the list of the $k-1$ distinct encounters is modified. After stabilization of the naming algorithm, all identifiers are distinct, so a non-biased random walk is performed by each agent. Then each agents meets every other agent regularly within polynomial time. Overall, after polynomial time, every agent has met every other agent and stored their identifiers in its local memory. When every agent has all other agent identifiers in its local memory, the leader status remain correct and unchanged. The memory cost of the algorithm is $O(k \log(k))$ per agent and the time complexity is polynomial.

   An alternative to this algorithm is as follows. Each agent performs a random walk in the network (at each node the agent chooses with equal probability (1/node degree) the next edge to visit). In [13] it is proved that the expected time for a random walk to cover all nodes of a graph is $O(n \log(n))$. Each time an agent visits a node it marks in the node table its identifier if it is not present. If the agent identifier is the maximum in the table then the agent is the leader otherwise it keeps the follower status. The memory complexity of the algorithm is $O(k \log(k))$ per node and the expected time complexity is $O(kn \log(n))$.

6.2 From leader election to naming

Now consider the reverse problem of solving the naming problem given a leader in the group of agents. Our solution is presented as Algorithms 4 and 5. For simplicity, we assume that the leader agent is identified by a special symbol that is not in the domain on non-leader agents identifiers and which can be recognized by the node

as the leader mark. First we assume that each node, when activating agents, gives lower priority to the leader agent (*i.e.* the code of the leader agent, if present on the node, is executed last). The intuition of the algorithm is as follows: the leader agent simply performs a Eulerian traversal of the tree, and is not influenced by the other nodes. On its way during each traversal, the leader leaves in the `edge` variable of each traversed whiteboard the outgoing edge it used to exit last time it visited the node. In a legitimate situation, those `edge` variables constitute a tree pointing toward the current location of the leader. The rationale for the non-leader nodes is as follows: *(i)* when the leader is not present on the same node, the non-leader node simply follows the `edge` left by the leader, trying to reach it, and *(ii)* when the leader is present on the same node, the non-leader agent first checks against duplicate identifiers of non-leader agents located at the same leader-based node, and pick up a new fresh identifier if needed, then they take the same outgoing edge as the leader, in order to always remain at the same node as the leader agent. Since the network is acyclic and the links are half-duplex, every non-leader node eventually meets the leader, and once met, they never leave the leader. So, eventually, the leader agent collects all non-leader agents at its current location. When all agents are co-located at the same node and check that no duplicate identifiers exist, the naming process is finished.

```
id: integer
execute() {
  if ( leader )
    edge := edge + 1 mod delta // follow the Eulerian traversal
    visit( edge )
  else
    if ( leader is present on the same node )
      if (id is conflicting among present agents on the node)
        id := new // take fresh identifier
      visit( edge + 1 mod delta ) // take same exit as that of the leader
    else
      visit( edge ) // follow leader
}
```

Algorithm 4: Deterministic agent code for naming in tree networks

```
foreach non-leader agent on node
  agent.execute()
end foreach
leader.execute()
```

Algorithm 5: Deterministic node code for naming in tree networks

Definition 6 (Legitimate configuration) *A configuration of the network is legitimate if it satisfies the following properties:* (i) *all non-leader agents have distinct identifiers,* (ii) *all agents are located in the same node, and* (iii) *all* `edge` *whiteboards point toward the node that contain all agents.*

Lemma 7 (Correctness) *Starting from a legitimate configuration, the naming problem is solved.*

Proof: In a legitimate configuration, all agents have distinct identifiers. From the code of the algorithm, an agent may change its identifier only when discovering that it shares the same identifier with another agent. As a result, an agent never changes its identifier onwards, and the naming problem is solved. □

Lemma 8 (Convergence) *Starting from an arbitrary initial configuration, a legitimate configuration is eventually reached.*

Proof: We first prove that eventually, all `edge` whiteboards point toward the node that contain the leader. We observe that the leader behavior does not depend on the behavior of the non-leader agents. Second, when the leader leaves a node (whatever the initialization of the whiteboard of this node may be), the `edge` whiteboard of this node will always point toward the leader agent onwards (the network is acyclic, so the leader agent may only come back though this edge), and the next time the `edge` whiteboard is modified, it will advertise the current last taken edge by the leader agent. Our second observation is that whatever the initialization of the whiteboards, the leader agent always perform a Eulerian traversal of the network. As a result, all nodes are eventually visited by the leader node, and when all nodes have been visited at least once, all `edge` whiteboards are pointing toward the leader agent.

The second step of the proof is to show that any non-leader agent eventually meets the leader agent. Since the `edge` whiteboards all point toward the leader agent, and that non-leader agents that are not located on the same node as the

leader simply follow the `edge` whiteboards, they always move toward the leader agent. Since the network is acyclic and the links are half-duplex, the leader agent and any non-leader agent are bound to meet within $O(m)$ rounds. Now, when a non-leader agent meets the leader agent, their moving behavior remains the same hereafter (*i.e.* the leader and the non-leader agents follow exactly the same path at the same moment–when the node they are both located on is activated). So, eventually, within $O(km)$ rounds all agents are located at the same node at a given moment, and remain located at the same node hereafter (though the node they are located changes anytime it is scheduled for execution).

When all agents are gathered at the same node, non-leader nodes (that are executed in sequence when the node is activated) simply choose different identifiers. After one such node activation, all agents have distinct identifiers, are gathered at the same node, and all `edge` whiteboards are pointing to them. As a consequence, the configuration is legitimate. □

7 Concluding remarks

In this paper, we introduced the problem of self-stabilizing mobile robots in graphs, and presented deterministic and probabilistic solutions to the problems of naming, and leader election among robots. From a practical point of view, the main difference between the two solutions is that the deterministic solution uses a whiteboard (*i.e.* a local memory available at every node that the agents can use to communicate with others) while the probabilistic one does not. An interesting open question that is raised by this work is the trade-off between whiteboard availability and randomness capabilities. In addition, it would be of theoretical interest to prove that the computational power of our model is strictly greater (in terms of predicates that can be computed) than the Population Protocol model.

References

[1] N. Agmon and D. Peleg. Fault-tolerant gathering algorithms for autonomous mobile robots. In *Proc. 15th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2004)*, pages 1070–1078, New Orleans, LA, USA, January 2004.

[2] Dana Angluin. Local and global properties in networks of processors (extended abstract). In *STOC*, pages 82–93. ACM, 1980.

[3] Dana Angluin, James Aspnes, Zoë Diamadi, Michael J. Fischer, and René Peralta. Computation in networks of passively mobile finite-state sensors. In *PODC*, pages 290–299, 2004.

[4] Dana Angluin, James Aspnes, Zoë Diamadi, Michael J. Fischer, and René Peralta. Computation in networks of passively mobile finite-state sensors. *Distributed Computing*, pages 235–253, March 2006.

[5] Dana Angluin, James Aspnes, Michael J. Fischer, and Hong Jiang. Self-stabilizing population protocols. In *Principles of Distributed Systems; 9th International Conference, OPODIS 2005; Pisa, Italy; December 2005; Revised Selected Papers*, volume 3974 of *Lecture Notes in Computer Science*, pages 103–117, December 2005.

[6] J. Beauquier, T. Herault, and E. Schiller. Easy Stabilization with an Agent. *5th Workshop on Self-Stabilizing Systems (WSS)*, 2194:35–51.

[7] Ajoy Kumar Datta, Maria Gradinariu, and Michel Raynal. Stabilizing mobile philosophers. *Inf. Process. Lett.*, 95(1):299–306, 2005.

[8] Xavier Défago, Maria Gradinariu, Stéphane Messika, and Philippe Raipin Parvédy. Fault-tolerant and self-stabilizing mobile robots gathering. In *DISC*, pages 46–60, 2006.

[9] Anders Dessmark, Pierre Fraigniaud, Dariusz R. Kowalski, and Andrzej Pelc. Deterministic rendezvous in graphs. *Algorithmica*, 46(1):69–96, 2006.

[10] Stefan Dobrev, Paola Flocchini, Giuseppe Prencipe, and Nicola Santoro. Searching for a black hole in arbitrary networks: optimal mobile agent protocols. In *PODC*, pages 153–161, 2002.

[11] S. Dolev. *Self-stabilization*. MIT Press, March 2000.

[12] S. Dolev, E. Schiller, and J. Welch. Random walk for self-stabilizing group communication in ad-hoc networks. *Reliable Distributed Systems, 2002. Proceedings. 21st IEEE Symposium on*, pages 70–79, 2002.

[13] Uriel Feige. A tight upper bound on the cover time for random walks on graphs. *Random Struct. Algorithms*, 6(1):51–54, 1995.

[14] P. Flocchini, G. Prencipe, N. Santoro, and P. Widmayer. Distributed coordination of a set of autonomous mobile robots. *IVS, pages 480-485, 2000.*, 2000.

[15] Fedor V. Fomin, Pierre Fraigniaud, and Nicolas Nisse. Nondeterministic graph searching: From pathwidth to treewidth. In Joanna Jedrzejowicz and Andrzej Szepietowski, editors, *MFCS*, volume 3618 of *Lecture Notes in Computer Science*, pages 364–375. Springer, 2005.

[16] Pierre Fraigniaud, David Ilcinkas, Sergio Rajsbaum, and Sébastien Tixeuil. The reduced automata technique for graph exploration space lower bounds. In Oded Goldreich, Arnold L. Rosenberg, and Alan L. Selman, editors, *Essays in Memory of Shimon Even*, volume 3895 of *Lecture Notes in Computer Science*, pages 1–26. Springer, 2006.

[17] S. Ghosh. Agents, distributed algorithms, and stabilization. *Computing and Combinatorics (COCOON 2000), Springer LNCS*, pages 242–251, 2000.

[18] T. Herman and T. Masuzawa. Self-Stabilizing Agent Traversal. *WSS01 Proceedings of the Fifth International Workshop on Self-Stabilizing Systems, Springer LNCS*, 2194:152–166, 2001.

[19] G. Prencipe. Corda: Distributed coordination of a set of autonomous mobile robots. *Proc. ERSADS, pages 185–190, May 2001.*, 2001.

[20] I. Suzuki and M. Yamashita. Distributed anonymous mobile robots—formation and agreement problems. *Proceedings of the 3rd International Colloquium on Structural Information and Communication Complexity (SIROCCO '96), Siena, Italy, June 1996.*, 1996.

[21] Prasad Tetali and Peter Winkler. On a random walk problem arising in self-stabilizing token management. In *PODC*, pages 273–280, 1991.

[22] Masafumi Yamashita and Tsunehiko Kameda. Computing on anonymous networks: Part i-characterizing the solvable cases. *IEEE Trans. Parallel Distrib. Syst.*, 7(1):69–89, 1996.

[23] Masafumi Yamashita and Tsunehiko Kameda. Computing on anonymous networks: Part ii-decision and membership problems. *IEEE Trans. Parallel Distrib. Syst.*, 7(1):90–96, 1996.

Unité de recherche INRIA Futurs
Parc Club Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 ORSAY Cedex (France)

Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)
Unité de recherche INRIA Sophia Antipolis : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399