Self-stabilizing Neighborhood Unique Naming Under Unfair Scheduler

Maria Gradinariu Colette Johnen

Laboratoire de Recherche en Informatique, UMR CNRS 8623,
Université de Paris Sud, 091405 Orsay cedex, France
email: ({mariag,colette}@lri.fr)

Abstract

Robustness is one of the most important requirements of modern fault-tolerant distributed systems. The most inclusive approaches to fault-tolerance in distributed systems is self-stabilization. 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 the need for explicit exception handler of backward recovery.

Ensuring that neighboring vertex till distance 2 apart have distinct labels is an important problem in the graph theory with applications to problems such as job assignment, task scheduling or radio frequencies assignement.

We propose a self-stabilizing probabilistic solution for the neighborhood unique naming problem in uniform, anonymous networks with arbitrary topology. The solution stabilizes under the unfair distributed scheduler. Our solution is the first self-stabilizing labeling algorithm which additionally ensures that the processors at distance 2 have distinct labels. We prove that this solution needs in average only one trial per processor.

Moreover, we use our algorithm to transform the [6] maximal matching algorithm self-stabilizing only under the central scheduler such that it copes up with the distributed unfair scheduler.

Keywords: distributed algorithms, fault-tolerance

1 Introduction

Self-stabilization. Self-stabilization introduced by Dijkstra, [2], provides an uniform approach to fault-tolerance, [9]. More precisely, this technique guarantees that, regardless of the initial state, the system will eventually converge to the intended behavior. In this paper we are particular interested in uniform (every processor in the system executes the same algorithm) and anonymous systems (processors does not have a distinct identifier).

Neighborhood Unique Local Naming problem In [5] is defined the labeling graphs problem with conditions at distance 2. Neighborhood unique local naming problem, issued from this theoretical graph problem, ensures that in each neighborhood the vertex have distinct labels. In other words, there is no vertex with the same label as one of its neighbors and there is no vertex having neighbors named identically. This problem is a classical coloration problem with an additional restriction: the vertex at distance two have distinct labels. A practical application is the assigning radio frequencies to transmitters such that transmitters that are closed (distance 1 or 2 apart) to each other use different frequencies.

matching in a graph is a set of edges where no two edges are adjacent. The matching set M is called maximal if there is no other matching set M' such that $M \subset M'$. The main applications of this problem in distributed computing area are job assignment and task scheduling.

Related works The classical vertex coloration problem is a restriction of the Neighborhood Unique Naming problem. The vertex coloration was previously studied for planar and bipartite graphs (see [3, 14, 12, 13]). Using a well-known result from graph theory, Gosh and Karaata [3] provide an elegant solution for coloring acyclic planar graphs with exactly six colors, along with an identifier based solution for acyclic orientation of planar graphs. This makes their solution limited to systems whose communication graph is planar and processors have unique identifiers. Sur and Srimani [14] vertex coloring algorithm is only valid for bipartite graphs. A paper by Shukla *et al.* (see [13]) provides a randomized self-stabilizing solution to the two coloring problem for several classes of bipartite graphs, namely complete odd-degree bipartite graphs and tree graphs. In [4] the authors presents three coloration algorithms for the arbitrary networks. Their solutions use $O(D)$ colors, where D is the maximal degree of the network.

Nevertheless, all the previous presented algorithms are not solutions for the Neighborhood Unique NAming problem since it may be possible that vertex at distance 2 are labeled identically.

The Neighborhood Unique Naming has multiple applications such as: the acyclic orientation of general networks or finding the maximal matching sets. The first appication is trivial, therefore we focus on the maximal matching problem. There are several works treating the maximal matching problem. The faster known sequential algorithm is a wave algorithm due to Micali and Vazirani, [7]. This solution is not self-stabilizing.

A deterministic self-stabilizing solution for the maximal matching problem was provided by Huang and Hsu in [6]. This solution assumes that : each processor has a pointer which points to one of its neighbors or to void. The pointers implementation is not detailed. The protocol idea is to let any processor to choose the neighbor to match with. Their solution stabilizes only under a central scheduler.

Our contribution. We present the first self-stabilizing solution for the unique local naming problem which ensures that the processors at distance two apart have distinct labels. Our solution works on anonymous, uniform networks with any topology and it needs in average only one trial per processor under the unfair, distributed scheduler. That is, a processor randomly updates its local identifier one time in the average. Than, this algorithm is used to transform the [6] maximal matching algorithm such that it stabilizes under the unfair distributed scheduler. Note that in the transformed algorithm the randomization is used only for breaking the symmetry, once any processor of the network gets an unique local identifier, the neighborhood unique naming algorithm has no further influence (no action is performed), the maximal matching algorithm evolution is then deterministic. Our solution copes up with the most powerful scheduler — the distributed unfair scheduler. We deal with the unfair distributed scheduler by letting only the processors with a locally maximal idenfier to choose their match. Hence, we avoid the matching cycles generation (more details in Section 5).

Outline of the paper. The rest of the paper is organized as follows: the model for the probabilistic algorithms is presented in Section 2. Section 3 shows that it is impossible to design a deterministic algorithm for the unique local naming problem. In Section 4, we present a probabilistic algorithm solving the unique local naming problem and in Section 5 we construct in the top of

2 Model

Distributed Systems A distributed system is a set of state machines called processors. Each processor can communicate with some processors called neighbors. The communication among two neighbors is carried out using communication registers (called "shared variables" throughout this paper). A processor p communicates to its neighbors via some shared variables that its neighbors can read but cannot update. The system's communication graph is drawn by representing processors as nodes and the neighborhood relationship by edges between the nodes. We will use $\mathcal{N}_p$ to denote the set of neighbors of the processor p.

A processor p in a distributed system executes an algorithm which is a finite set of guarded actions of the form : $\langle label \rangle :: \langle guard \rangle \longrightarrow \langle statement \rangle$, where each guard is a boolean expression over the p variables and the shared variables of p's neighbors, and each statement is a deterministic or probabilistic update of the local and shared variables of p.

The state of a processor p is the value of its variables. A *configuration* of a distributed system is an instance of the processor states. A processor is *enabled* in a given configuration if at least one of the guards of its algorithm is *true*. During a computation step, one or more enabled processors perform the statement of an enabled action.

A distributed system can be modeled by a transition system. A transition system is a three-tuple $S = (\mathcal{C}, \mathcal{T}, \mathcal{I})$ where $\mathcal{C}$ is the collection of all configurations, $\mathcal{I}$ is a subset of $\mathcal{C}$ called the set of initial configurations, and $\mathcal{T}$ is a function $\mathcal{T}$ from $\mathcal{C}$ to the set of $\mathcal{C}$ subsets. A $\mathcal{C}$ subset of $\mathcal{T}(c)$ is called a c transition. An element of a c transition t, is called an output of t. In a probabilistic system, there is a probabilistic law on the output of a transition; in a deterministic system, each transition has only one output.

The computation step (c_1, c_2) belongs to the c_1 transition t iff $c_2 \in t$. All computation steps beginning at the configuration c_1 belong to a c_1 transition.

A *computation* of a distributed system DS is a *maximal* sequence of computations steps. *Maximality* means that the sequence is either infinite, or no processor is enabled in the terminal (final) configuration. All computations considered in this paper are assumed to be maximal. The computation set of a distributed system DS is denoted by $\mathcal{E}_{DS}$.

Scheduler In this model, a *scheduler* is a *predicate* over the system computations. In a computation, a transition (c_i, c_{i+1}) occurs due to the execution of a nonempty subset of the enabled processors in the configuration c_i. In every computation step, this subset is chosen by the scheduler. We refer to the following types of schedulers in this paper : *central scheduler* — in every computation step, only one of the enabled processors is chosen to execute its action; *synchronous scheduler* — in every computation step, all enabled processors are chosen concurrently by the scheduler; *distributed unfair scheduler* — during a computation step, a nonempty subset of the enabled processors is chosen by the scheduler.

A strategy is the set of computations that can be obtained under a specific scheduler choice. At the initial configuration, the scheduler "chooses" one set of enabled processors (it chooses a transition). For each output of the selected transition, the scheduler chooses a second transition, and so on. The strategy formal definition is based on the tree of computations. Let c be a system configuration. A *TS-tree* rooted in c, $\mathcal{T}ree(c)$, is the tree-representation of all computations beginning in c. Let n be a node in $\mathcal{T}ree(c)$ (i.e. a configuration), a *branch* rooted in n is the set of all $\mathcal{T}ree(c)$ computations starting in n with a computation step of the same n transition.

restriction of $\mathcal{T}ree(c)$ such that the degree of any $\mathcal{T}ree(c)$'s node (configuration) is at most 1. A strategy is defined as follows :

Definition 1 (Strategy) *Let DS be a distributed system, let D be a scheduler and let c be a TS configuration. We call a strategy of DS under D rooted in c a* sub-TS-tree *of degree 1 of $\mathcal{T}ree(c)$ such that any computation of the sub-tree verifies the scheduler D.*

Let st be a strategy of the distributed system DS, an *st-cone* $\mathcal{C}_h$ is the set of all possible st-computations with the same prefix h (for more details see [10]). The last configuration of h is denoted $last(h)$.

We have equiped a strategy with a probabilistic space (see [1] for more details). The measure of an st-cone $\mathcal{C}_h$ is the measure of the prefix h (i.e., the product of the probability of every computation step occurring in h). An st-cone $\mathcal{C}_{h'}$ is called a *sub-cone* of $\mathcal{C}_h$ if and only if $h' = [hx]$, where x is a computation factor.

Deterministic self-stabilization In order to define self-stabilization for a distributed system, we use two types of predicates : the legitimacy predicate — defined on the system configurations and denoted by $\mathcal{L}$ — and the problem specification — defined on the system computations and denoted by $\mathcal{PS}$.

Let $\mathcal{X}$ be a set and $Pred$ be a predicate defined on $\mathcal{X}$. The notation $x \vdash Pred$ means that the element x of $\mathcal{X}$ satisfies the predicate $Pred$.

Definition 2 (Deterministic self-stabilization) *Let DS be a distributed system. DS is self-stabilizing for a specification $\mathcal{PS}$ if and only if the following two properties hold :*
(1) convergence — all computations of DS reach a configuration that satisfies the legitimacy predicate. Formally, $\forall e \in \mathcal{E}_{DS} \ :: e = ((c_0, c_1)(c_1, c_2)\dots) \ : \exists n \geq 1, c_n \vdash \mathcal{L}$;
(2) correctness — all computations starting in configurations satisfying the legitimacy predicate satisfy the problem specification $\mathcal{PS}$. Formally, $\forall e \in \mathcal{E}_{DS} \ :: e = ((c_0, c_1)\ (c_1, c_2)\ \dots) \ : c_0 \vdash \mathcal{L} \Rightarrow e \vdash \mathcal{PS}$.

Probabilistic self-stabilization Let DS be a distributed system. A predicate P is closed for the computations of DS if and only if when P holds in a configuration c, P also holds in any configuration reachable from c.

Notation 1 *Let $\mathcal{DS}$ be a distributed system, D be a scheduler and st be a strategy of DS under D. Let CP be the set of all system configurations satisfying a closed predicate P (formally $\forall c \in CP, c \vdash P$). The set of st-computations that reach configurations of CP is denoted by $\mathcal{EP}$ and its probability by $Pr_{st}(\mathcal{EP})$.*

In this paper we study silent algorithms - those for which all computations are finite. Then the legitimate configurations satisfy the Problem Specification predicate. The probabilistic stabilization for this particular case of algorithms is restricted to the probabilistic convergence property : all computations reach a legitimate configuration.

Definition 3 (Probabilistic Stabilization) *A distributed system $\mathcal{DS}$ is* self-stabilizing *under a scheduler D for a specification PS if and only if there exists a closed legitimacy predicate $\mathcal{L}$ on configurations such that in any strategy st of $\mathcal{S}$ under D, the following conditions hold :*

Formally, $\forall st, Pr_{st}(\mathcal{EL}) = 1$.

- *All configurations satisfying* $\mathcal{L}$ *verifies the specification* PS.

Convergence of Probabilistic Stabilizing Systems Building on previous works on probabilistic automata (see [11, 15, 10, 8],), [1] presented a framework for proving self-stabilization of probabilistic distributed systems. In the following we recall a key property of the system called *local convergence* and denoted by LC.

Definition 4 (Local Convergence) *Let DS be a distributed system. Let st be a strategy of DS, $PR1$ and $PR2$ be two predicates on configurations of DS, where $PR1$ is a closed predicate. Let δ be a positive probability and N a positive integer. Let $\mathcal{C}_h$ be a st-cone with $last(h) \vdash PR1$ and let M denote the set of sub-cones $\mathcal{C}_{h'}$ of the cone $\mathcal{C}_h$ such that the following is true for every sub-cone $\mathcal{C}_{h'}$: $last(h') \vdash PR2$ and $length(h') - length(h) \leq N$. The cone $\mathcal{C}_h$ satisfies LC $(PR1, PR2, \delta, N)$ if and only if $Pr(\bigcup_{\mathcal{C}_{h'} \in M} \mathcal{C}_{h'}) \geq \delta$.*

Now, if in strategy st, there exist $\delta_{st} > 0$ and $N_{st} \geq 1$ such that any st-cone, $\mathcal{C}_h$ with $last(h) \vdash PR1$, satisfies $LC(PR1, PR2, \delta_{st}, N_{st})$, then the main theorem of [1] states that the probability of the set of st-computations reaching configurations satisfying $PR1 \wedge PR2$ is 1.

Theorem 1 *Let st be a strategy of the distributed system DS under a scheduler D. Let $PR1$ be a closed predicate on configurations such that $P_{st}(\mathcal{EPR}1) = 1$. Let $PR2$ be a closed predicate on configurations. Let us note $PR12$ the predicate $PR1 \wedge PR2$. If $\exists\ \delta_{st} > 0$ and $\exists\ n_{st} > 1$ such that st verifies the $LC(PR1, PR2, \delta_{st}, n_{st})$ property then $P_{st}(\mathcal{EPR}12) = 1$.*

3 Impossibility results

In the following, we show that it is impossible to provide a self-stabilizing deterministic solution to Neighbourhood Unique Naming problem under an unfair distributed scheduler on anonymous and uniform networks. We prove that even the unique local naming (neighboring processors have distinct identifiers) is impossible to achieve in a deterministic way under an unfair distributed scheduler.

Lemma 1 *There is no deterministic self-stabilizing algorithm solving the unique local naming problem in uniform and anonymous networks under unfair distributed scheduler.*

Proof: Let A be a deterministic, self-stabilizing algorithm that solves the unique local naming problem. We will study a distributed system whose topology is a ring of n processors. Let c_0 be a configuration of A where all processors are in the same state s_0. This configuration is illegitimate. In a legitimate configuration, there are two neighbors having different states. Since s_0 is an illegitimate configuration, at least one processor may execute an action. As all processors are in the same state then all processors are able to execute the same action.

After the computation step where all processors have performed the same action, the obtained configuration is c_1. In c_1, all processors are in the same state. This configuration is symmetrical (thus illegitimate).

We build an infinite computation where all reached configurations are symmetrical : this computation never reaches a legitimate configuration. □

Corollary 1 *There is no deterministic protocol solving the Neighbourhood Unique Naming problem under a unfair distributed scheduler.*

scheduler

In the previous section we prove that there is no deterministic algorithm solving the Neighborhood Unique Naming problem under an unfair distributed scheduler on anonymous and uniform networks. In the current section, we present a self-stabilizing probabilistic solution for this problem.

4.1 Neighborhood Unique Naming protocol description

The Algorithm 4.1 idea is very simple. Each processor has a variable, referred in the algorithm as *lid*, which indicates its local identifier and a $flag$ which signals the presence of at least two neighbors having the same *lid*. A processor which cannot execute $\mathcal{A}_2$, having at least two neighbors with the same local identifier, l, sets its $flag$ to l (Action $\mathcal{A}_1$). A processor having a neighbor with the same value of *lid* chooses randomly a new identifier from a bounded set of values (Action $\mathcal{A}_2$). The same action is executed by the processor, p, when it has a neighbor, q, which flag value is setted to p'*lid*. In this case, the processor q has at least two neighbors (p and another one) with the same *lid*.

We show that the bound $2n^2$ (for the local identifier values) is optimal in term of time complexity, when we deal with a distributed unfair scheduler.

Algorithm 4.1 Neighborhood Unique Naming algorithm on the processor i

---

Constant :
B : integer constant proven optimal for the value $2n^2$;

Shared Variable :
$lid.i$: a positive and no null integer bounded by B
$flag.i$: a positive integer bounded by B

Actions :
$\mathcal{A}_1$: $(\forall j \in \mathcal{N}.i\ (lid.i \neq lid.j) \wedge (lid.i \neq flag.j)) \wedge (\exists (j,k) \in \mathcal{N}.i\ (lid.j = lid.k) \wedge (flag.i \neq lid.j)) \wedge$
$(\nexists\ (t,l) \in \mathcal{N}.i,\ (lid.t = lid.l = flag.i)) \longrightarrow flag.i = lid.j$;

$\mathcal{A}_2$: $\exists j \in \mathcal{N}.i\ (lid.i = lid.j) \vee (lid.i = flag.j) \longrightarrow lid.i = random(1, \ldots, B)$;

4.2 Algorithm 4.1 analysis

In order to show the self-stabilization of Algorithm 4.1, we define the legitimate configurations for this algorithm as follows :

Definition 5 *Let p be a processor. Correct_lid(p) is the following predicate on configurations:*

- *$\forall q$ neighbor of p, $lid.p \neq lid.q$;*
- *$\forall q$ neighbor of p, $lid.p \neq flag.q$;*
- *$\forall r$ neighbor of a p's neighbor, $lid.p \neq lid.r$;*

Definition 6 *A legitimate configuration for Algorithm 4.1 is a configuration satisfying the predicate $\mathcal{LID}$: "every processor p verifies Correct_lid(p)".*

Proof: By Definition 6 once in a legitimate configuration no processor is able to execute an action. □

The analysis of system stabilization is reduced to the convergence proof.

Definition 7 *Let c be a configuration. We name m_c the number of processors p which does not verify Correct_lid(p).*

Note 1 *Let c be a configuration. In any computation from c there are at most consecutive $n - m_c$ computation steps in which Action $\mathcal{A}_2$ is not executed.*

Scena is the following scenario described informally by : "at each computation step, where at least one processor performs Action $\mathcal{A}_2$, (i) exactly one of these processors verifies the *Correct_lid* predicate after this computation step, (ii) the other processors keep their *lid* value".

Note 2 *Let c be a configuration. On any computation following the scenario Scena, the predicate Correct_lid is closed : once a processor p verifies Correct_lid(p), no action of p, of p's neighbor, or of a neighbor of a p's neighbor will change that.*

Lemma 3 *Let st be a strategy under an unfair distributed scheduler on a system executing the Algorithm 4.1. There exist $\epsilon > 0$ and $N > 0$ such that the strategy st is satisfying LC (true, $\mathcal{L}ID$, ϵ, N).*

Proof: Let $\mathcal{C}_h$ be a cone of the strategy st such that $last(h) = c$ does not verify $\mathcal{L}ID$.

We show that there is a positive probability ϵ and a positive integer N such that the cone $\mathcal{C}_h$ has a sub-cone, $\mathcal{C}_{h1}$, of probability $\epsilon_1 \geq \epsilon$ with $lenght(h1) \leq N$ where $last(h1)$ verifies $\mathcal{L}ID$.

Assume that $m_c \neq 0$. After at most $n - m_c$ computation steps a processor executes Action $\mathcal{A}_2$. We study the scenario: independently of scheduler choice, only one processor will change its *lid* value and all the other chosen processors to execute $\mathcal{A}_2$ will get the same *lid* value.

The probability that a processor keeps its *lid* value, after $\mathcal{A}_2$ is $\frac{1}{B}$.

Let p be the processor which will change its *lid* value from the configuration c. Let d_p^1 and d_p^2 the number of neighbors at distance 1 and 2 of p and let $(lid_j)_{j=1,d_p^1+d_p^2}$ be their local identifiers and let $(flag_k)_{k=1,d_p^1}$ be the flag values for the neighbors at distance 1. The probability that p chooses a value equal to lid_j or $flag_k$ is $\frac{1}{B}$. The probability that the new chosen value be different of all *lid* values of its neighbors at distance 1 or 2 and of *flag* values of its neighbors at distance 1 is $P_{st}((lid_p \neq lid_1) \wedge \ldots \wedge (lid_p \neq lid_{d_p^1+d_p^2}) \wedge (lid_p \neq flag_1) \wedge \ldots \wedge (lid_p \neq flag_{d_p^1}) = 1 - (\sum_{j=1}^{j=d_p^1+d_p^2} P_{st}(lid_p = lid_j) + \sum_{j=1}^{j=d_p^1} P_{st}(lid_p = flag_j)) = 1 - \frac{2d_p^1+d_p^2}{B} \leq 1 - \frac{2(n-1)}{B}$.

The probability of the computation step that we have defined is greater than $\frac{1}{B}^{(m-1)}(1-\frac{2(n-1)}{B})$. The number of processors whose neighbor has the same *lid* value has decreased by at least 1, hence in at most $m-1$ similar sequences of computation steps, a legitimate configuration is obtained. The size of this sequence is bounded by $n - m_c + 1 \leq n$.

The probability of the cone of computations having this configuration, the last in its history is: greater than $(\frac{1}{B})^{(m-1)(\frac{m}{2})}(1 - \frac{2n}{B})^{(m-1)}$.

Thus $N = n^2$ and $\epsilon = \left(\frac{1}{B}^{\frac{n}{2}}(1 - \frac{2n}{B})\right)^{n-1}$. □

Theorem 2 *Algorithm 4.1 is self-stabilizing under an unfair distributed scheduler for the unique local naming specification.*

of computations reaching a legitimate configuration has the probability 1. The result is proven by Lemma 3 and Theorem 1. □

4.3 Space and time complexity

In this section, we motivate the choice of the value $2n^2$ as bound for the local identifiers in Algorithm 4.1.

In the following lemma we show that a $2n^2$ bound is enough to guarantee that a processor performs in the average only one time the randomized Action $\mathcal{A}_2$.

Lemma 4 *When $B > 2n(n-1)$ where n is the system size, a processor performs in the average only one time the randomized Action $\mathcal{A}_2$.*

Proof: Let c be the initial configuration of Algorithm 4.1.

The probability that after the action $\mathcal{A}_2$, a processor has an unique local name is $1 - \frac{2d_p^1 + d_p^2}{B}$ where d_p^1 and d_p^2 are the number of neighbors at distance 1 and 2. Note that $2d_p^1 + d_p^2$ could be borned by $2(n-1)$. In the average $m_c(1 - \frac{2(n-1)}{B})$ processors have an unique local identifier after all processors that do not verify $Correct\_lid$ have performed one time $\mathcal{A}_2$.

$B > 2n(n-1)$ guarantees that in average, all processors have an unique local name after at most one Action $\mathcal{A}_2$. □

Lemma 5 *The space complexity for Algorithm 4.1 is $O(log(n))$ bits per processor.*

Proof: Each processor executing Algorithm 4.1 uses two shared variable bounded by $2n^2$. Hence, the space complexity for Algorithm 4.1 is $O(log(n))$. □

5 Application

In this section we transform, using the Neighborhood Unique Naming algorithm, the maximal matching algorithm designed by Huang *et al.*([6]) such that, the new transformed algorithm stabilizes under the distributed unfair scheduler. We use the unique naming at distance 1 in order to avoid the cycles generation (more details in Lemma 7) and at distance 2 in order to avoid the confusion between neighbors when a processor tries to match someone.

5.1 Maximal Matching algorithm description

Algorithm 5.1 uses as substratum the Neighborhood Unique Naming algorithm presented in the previous section (Algorithm 4.1). Note that Algorithm 4.1 is silent and once stabilized the neighbors at distance 1 and 2 have distinct idenfiers.

Each processor keeps a shared variable, referred to as local identifier (i.e. *lid* variable in the algorithm). Any processor has a distinct identifier in its neighborhood.

In order to construct the maximal matching, each processor maintains a shared variable *match*. The possible values for the *match* variable of an arbitrary processor p are :

- a p's neighbor local identifier—meaning that the current processor is matched with the referred neighbor (there is no ambiguity since the Neighborhood Unique Naming provides unique identifiers for the neighbors);
- 0 — meaning that the current processor is not matched.

Constants :
B : integer constant proven optimal for the value $2n^2$;

Shared Variable :
$lid.i$: a positive and no-null integer bounded by B;
$match.i$: a positive integer bounded by B

Actions :
$\mathcal{A}_1$: $(match.i = 0) \wedge (lid.i > max(lid.k, k \in \mathcal{N}.i \wedge match.k = 0) \wedge$
$(\exists j \in \mathcal{N}.i,\ match.j = 0) \wedge (\forall k \in \mathcal{N}.i, match.k \neq lid.i) \longrightarrow match.i = lid.j$
$\mathcal{A}_2$: $(match.i = 0) \wedge (\exists j \in \mathcal{N}.i,\ match.j = lid.i) \longrightarrow match.i = lid.j$
$\mathcal{A}_3$: $(match.i = lid.j) \wedge (match.j \neq 0) \wedge (match.j \neq lid.i) \longrightarrow match.i = 0$
$\mathcal{A}_4$: $(match.i \notin \{lid.j \mid j \in \mathcal{N}.i\} \bigcup \{0\}) \longrightarrow match.i = 0$

The detailed description of Algorithm 5.1 uses the matched processors notion defined as follows :

Definition 8 (Matched and Inactive processors) *Two neighbors (p, q) are matched iff $match.p = q$ and $match.q = p$. The set of matched processors is denoted $\mathcal{M}$. A processor p is inactive iff all its neighbors are matched and $match.p = 0$. The set of inactive processors is denoted $\mathcal{I}$. A processor p is unmatched iff $match.p = 0$.*

A processors p for which the following conditions hold will matches q (Action $\mathcal{A}_1$).
- p is unmatched ($match.p = 0$);
- the p's local identifier is the maximal value of local identifiers of p's unmatched neighbors;
- p has an unmatched neighbor q;
- there is no p neighbor matched with p.

An unmatched processor p having a neighbor q matched with p will match q (Action $\mathcal{A}_2$). A processor p matched with a neighbor which is not matched with p becomes an unmatched processor (Action $\mathcal{A}_3$). Finally, a processor for which the match variable indicates a local identifier different to all its neighbor identifiers becomes an unmatched processor (Action A_4).

5.2 The algorithm analysis

The analysis of the Algorithm 5.1 is done by prove that any computation starting in a configuration verifying the predicate $\mathcal{L}ID$ reaches a terminal configuration.

Definition 9 (Legitimate configuration) *A legitimate configuration satisfies the predicate $\mathcal{L}ID$ and all processors are inactive or matched.*

Lemma 6 *The set of configurations satisfying the predicate $\mathcal{L}ID$ is closed.*

Proof: Let c be a configuration verifying the predicate $\mathcal{L}ID$. Let c' be a configuration accessible from c. As c verifies $\mathcal{L}ID$ then the processors in c could execute any action. The execution of any action does not modify the value of the lid variable, hence any two neighbors still have different lid values. □

cessors $(p_k)_{k\in[1,m]}$ *form a matching cycle iff* $match.p_j = p_{j+1}, \forall j \in [1, m-1]$ *and* $p_1 = p_m$.
Let c be a configuration of Algorithm 5.1. c is cycle free if there is no matching cycle in c.

The previous solutions for the maximal matching problem ([6]) use weaker demons (i.e the central demon) to show the stabilization. This choice is motivated by possible matching cycles creation. The presence of a matching cycle could generate infinite executions (i.e. executions which do not reach legitimate configurations). We show that even under an unfair distributed scheduler the system executing Algorithm 5.1 does not create matching cycles.

Lemma 7 *From a configuration satisfying the predicate $\mathcal{LID}$, the Algorithm 5.1 does not create a matching cycle.*

Proof: Let c be a configuration of Algorithm 5.1 satisfying the predicate $\mathcal{LID}$. The only action of Algorithm 5.1 which could generate a cycle is $\mathcal{A}_1$. Let us consider a computation step from c where a processor p performs the action $\mathcal{A}_1$ to choose the processor q as "match". q is unmatched before the computation step. During this computation step, q can perform only the action $\mathcal{A}_2$ or $\mathcal{A}_4$. After the action $\mathcal{A}_2$, q is matched. After, the execution of the action $\mathcal{A}_4$, q is still unmatched. In all cases, q and p are not inside a matching cycle. □

Lemma 8 *Let c be a terminal configuration of Algorithm 5.1. Let p be a processor. If* $match.p \neq 0$ *then p is matched.*

Proof: If $match.p \neq 0$ and p is not matched, then p can perform the action $\mathcal{A}_3$ or $\mathcal{A}_4$; or p's match can perform the action $\mathcal{A}_2$. □

Lemma 9 *The terminal configurations are legitimate.*

Proof: Let c be a terminal configuration of Algorithm 5.1. Clearly c satisfies the predicate $\mathcal{LID}$

Let p be a processor. If $match.p = lid.q$ then p is matched (Lemma 8). If $match.p = 0$ then $\forall q \in \mathcal{N}_p$ we have $match.q \neq 0$ (otherwise a processor could perform the action $\mathcal{A}_1$). Therefore, if $match.p = 0$ then all p's neighbors are matched (Lemma 8). In c, all processors are matched or inactive. □

Note that since the neighboring nodes at distance 2 have distinct identifiers we avoid the confusion in choosing a matching.

Theorem 3 *In a legitimate configuration the states of match variables defines a maximal matching.*

Proof: The proof is done according to the definition of legitimate configuration. □

Theorem 4 *Any computation starting in a configuration satisfying the predicate $\mathcal{LID}$ is finite.*

Proof: Let e be a computation starting in a configuration satisfying the predicate $\mathcal{LID}$. During e, a processor can perform at most one time the action $\mathcal{A}_4$ and the action $\mathcal{A}_2$. Between two consecutive actions $\mathcal{A}_1$, a processor performs one and only one time the action $\mathcal{A}_3$.

Assume that e is an infinite computation. e has an infinite suffix e' where no processor performs the action $\mathcal{A}_4$ or the action $\mathcal{A}_2$. During e', the processors can perform only the actions $\mathcal{A}_1$ and $\mathcal{A}_3$.

Let cs be a computation step along e' where a processor p performs the action $\mathcal{A}_1$ to choose the processor q as "match". Before the computation step, q is unmatched. During the computation step and after that (along e'), q does not perform any action (it could only perform the action $\mathcal{A}_2$ or $\mathcal{A}_4$). Thus q stays unmatched and p cannot perform any action. Therefore, along e', p performs at most one action $\mathcal{A}_1$; and maybe one action $\mathcal{A}_3$.

Along e', a processor performs at most one action $\mathcal{A}_1$ and one action $\mathcal{A}_3$ (One can prove that along e' no action $\mathcal{A}_1$ is performed). Thus, e is a finite computation. □

We present the first algorithm for Neighborhood Unique Naming problem, self-stabilizing on anonymous and uniform systems. The main feature of our solution is that it guarantees that the neighboring processors at distance 1 and 2 have distinct identifiers. Our solution copes up with unfair distributed schedulers and is time optimal, more precisely we guarantee that the neighborhood unique naming is done in only one trial per processor in the average under any unfair scheduler. Moreover, it leads to a time optimal vertex coloration with no additional cost.

The presented solution is used as substratum in the modification of the [6] maximal matching algorithm such that the new algorithm stabilizes under the distributed unfair scheduler.

References

[1] J. Beauquier, M. Gradinariu, and C. Johnen. Randomized self-stabilizing optimal leader election under arbitrary scheduler on rings. Technical Report 1225, Laboratoire de Recherche en Informatique, September 1999.

[2] E. Dijkstra. Self stabilizing systems in spite of distributed control. *Communications of the ACM*, 17:643–644, 1974.

[3] S. Ghosh and Mehmet Hakan Karaata. A self-stabilizing algorithm for coloring planar graphs. *Distributed Computing*, pages 7:55–59, 1993.

[4] M. Gradinariu and Sébastien Tixeuil. Tight space uniform self-stabilizing l-mutual exclusion. Technical Report 1249, Laboratoire de Recherche en Informatique, Université de Paris Sud, March 2000.

[5] J. R. Griggs and R. K. Yeh. Labeling graphs with a condition at distance two. *SIAM, Journal of Discrete Mathematics*, pages 5:586–595, 1992.

[6] S. Hsu and S. Huang. A self-stabilizing algorithm for maximal matching. In *Information Processing Letters*, volume 43(2), pages 77–81, 1992.

[7] S. Micali and V. Vazirani. An algorithm for finding maximum matching in general graphs. In *21st IEEE Annual Symposium on Foundations of Computer Science*, 1980.

[8] A. Pogosyants, R. Segala, and N. Lynch. Verification of the randomized consensus algorithm of aspen and herlihy: a case study. In *Distributed Computing(13)*, pages 155–186, 2000.

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

[10] R. Segala. *Modeling and Verification of Randomized Distributed Real-Time Systems.* PhD thesis, MIT, Departament of Electrical Engineering and Computer Science, 1995.

[11] R. Segala and N. Lynch. Probabilistic simulations for probabilistic processes. In Springer-Verlag, editor, *CONCUR '94, Concurrency Theory, 5th International Conference , LNCS:836*, Uppsala, Sweden, August 1994.

[12] S. Shukla, D. Rosenkrantz, and S. Ravi. Developing self-stabilizing coloring algorithms via systematic randomization. In *Proceedings of the International Workshop on Parallel Processing*, pages 668–673, Bangalore, India, 1994. Tata-McGrawhill, New Delhi.

for anonymous networks. In *Proceedings of the Second Workshop on Self-stabilizing Systems*, pages 7.1–7.15, 1995.

[14] S. Sur and Pradip K. Srimani. A self-stabilizing algorithm for coloring bipartite graphs. *Information Sciences*, 69:219–227, 1993.

[15] S. H. Wu, S. A. Smolka, and E. W. Stark. Composition and behaviors of probabilistic i/o automata. In *concur94*, pages 513–528, 994.