Easy Stabilization with an Agent

Joffroy Beauquier[1], Thomas Hérault[1], and Elad Schiller[2]

[1] Laboratoire de Recherche en Informatique
Bâtiment 490, Université Paris-Sud, 91 405 Orsay, France
{jb,herault}@lri.fr

[2] Department of Computer Science, Ben-Gurion University of the Negev
Beer-Sheva 84105, Israel
schiller@cs.bgu.ac.il

Abstract. The paper presents a technique for achieving stabilization in distributed systems. This technique, called agent-stabilization, uses an external tool, the agent, that can be considered as a special message created by a lower layer. Basically, an agent performs a traversal of the network and if necessary, modifies the local states of the nodes, yielding stabilization.

1 Introduction

Fault tolerance and robustness are important properties of distributed systems. Frequently, the correctness of a distributed system is demonstrated by limiting the set of possible failures. The correctness is established by assuming a predefined initial state and considering every possible execution that involves the assumed set of failures.

The requirements of some distributed systems may include complex failures, such as a corruption of memory and communication channels. The self-stabilization methodology, introduced by Dijkstra [3], deals with systems that can be corrupted by an unknown set of failures. The system self-stabilizing property is established by assuming that any initial state is possible, and considering only executions without transient failures. Despite a transient failure, in a finite time, a self-stabilizing system will recover a correct behavior.

Self-stabilizing systems are generally hard to design, and still harder to prove [4]. In this paper, we are proposing to investigate a specific technique for dealing with the corruption problem. The basic idea appears in a paper of Ghosh [5] and consists of the use of mobile agents for failure tolerance. A mobile agent can be considered as an external tool used for making an application self-stabilizing. The notion that we propose here is different from Ghosh's notion. A mobile agent can be considered (in an abstract way, independently of its implementation) as a message of a special type, that is created by a lower layer (the outside). We will assume that some code corruption is possible and a mobile agent (or more simply an agent) can carry two types of data: a code, that can be installed at some nodes, and some information (referred to as contained in the briefcase), that can be read or modified by the processors. Code and briefcase are specific to the application controlled by the agent.

The outside can create a single agent and install it on an arbitrary node (the initiator). As long as the agent has not been destroyed, no other agent can be created. An agent has a finite lifetime. It can be destroyed either by its initiator, or by any site if it is supposed to have traveled too long. After an agent has been destroyed, the outside will eventually create a new one (with possibly another initiator). Once created, an agent moves from a processor to a neighbor of this processor, following a non-corruptible rule. More precisely, each node has permanently in its ROM some code, which is the only non-corruptible information and does not depend on any particular application. Moreover, it is simple and compact.

When an agent reaches a node (by creation or by transmission), the code carried by the agent is installed. The code is in two parts, agent rules and application (or algorithm) rules. Agent rules are the only applicable rules when an agent is present, and, on the contrary, application rules are the only applicable rule when it is not. Agent code execution is always finite and ends by a call to the agent circulation code, responsible for transmitting the agent to another site. The ability of the agent to overwrite a corrupted application code (or simply to patch the code in a less severe fault model), motivates the inclusion of code in the agent. Because it is supposed to travel, an agent (code and briefcase) can be corrupted.

In our agent model, there are two crucial properties, non-interference and stabilization. The non-interference property expresses that if no failure (corruption) occurred after the creation of the previous agent, after the destruction of this agent, the controlled application must not behave differently whether or not a new agent is present in the network. The property involves that the application cannot use the agents for solving its problem and that the agents cannot perform resets in a predetermined global configuration. The stabilization property states that if a failure occurred, after a finite number of agent creations, the application behaves accordingly to its specification.

Agent-stabilization has some advantages over classic self-stabilization. With self-stabilization, even if the stabilization time can be computed, it says nothing about the effective time needed for stabilization, which strongly depends on the communication delays and, consequently on the global traffic in the network. At the contrary, if agents are implemented in a way that guarantees small communication delays (as high priority messages for instance), an efficient bound on agent-stabilization time can be given, because such a bound depends only on the time of the network traversal by the fast agent, plus the time between two successive agent creations, that can be tailored to the actual need. A second advantage is that agent-stabilization is more powerful than self-stabilization. Any self-stabilizing solution is agent-stabilizing, with agents doing nothing, but some impossibility results from self-stabilization theory can be bypassed with agents, as we will show in the sequel.

2 The Specification

The system is a set P of n communicating entities that we call *processors*, and a relation *neighbors* $\subset P \times P$. To each processor p in P is associated some internal variables, an algorithm and a communication register r_p (r_p can have a composite type). Each processor $p \in P$ has a read and write access to its register r_p and a read access to its neighbor's communication registers. The local state of a processor is the set of the values of the internal variables and of its register. A vector whose components are the local states of every processor of the system is the configuration of the system.

The algorithm is defined as a set of guarded rules, of the form $Guard_i \longrightarrow Action_i$ where $Guard_i$ is a predicate on the local variables, the communication register and the communication registers of the processor's neighbors, and $Action_i$ is a list of instructions which could modify the communication register and/or the local variables of the processor. A rule is said to be enabled if its guard is true and disabled if its guard is false.

The system can evolve from a configuration to another by applying at some processor p an atomic step. For defining the computation rules of a processor, we need now to look more precisely at what is held in the communication register. It holds two parts : an algorithm-specific part including all information that the processor has to communicate to its neighbors and an agent-specific part. The agent-specific part of the communication register is composed of 6 fields : Agent Code, Algorithm Code, Briefcase, Next, Prev and Present.

The fields Agent Code and Algorithm Code hold two finite sets of guarded rules, depending on the algorithm ; the Briefcase is a field of constant size depending on the algorithm ; the fields Next and Prev are either $\perp$ or pointers to the neighbors of the processor (they point respectively to the previous processor which had the agent and to the next which will have it), the field Present is a boolean and indicates whether or not the agent is present at the processor.

An atomic step for a processor p is the execution by p of one of the following rules :

Agent step: if $Present_p$ is true, select and compute one of the enabled rules held by the Agent Code part of the register, if $Next_p$ is not $\perp$, install the agent in $Next_p$ from p ; set $Present_p$ to false.

Algorithm step: if $Present_p$ is false, select and compute one of the enabled rules held by the Algorithm Code part of the register ;

Agent installation: spontaneously install the agent in p from $\perp$;

Installing an agent in p from q is performing the following algorithm :

1. Set $Present_p$ to true;
2. Set $Prev_p$ to q;
3. If q is $\perp$, instantiate Algorithm Code with a fresh copy in Agent Code;
4. If q is not $\perp$, copy the values of $Briefcase_q$, $AlgorithmCode_q$ and $AgentCode_q$ from q to p.

An execution with initial configuration C is an alternate sequence of configurations and atomic steps, each configuration following an atomic step being obtained by the computation of the atomic step from the previous configuration, C being the first configuration of the sequence. We will consider here only fair executions. A fair execution is an execution such that if a guard of a processor P is continuously enabled, this processor will eventually perform an atomic step.

System executions satisfy two axioms that we describe here :

Uniqueness of creation. The atomic step *Agent Installation* can be performed only on a configuration where for all processor p, $Present_p$ is false ;

Liveness of creation. In each execution there is infinitely many configurations in which an agent is present;

For the sake of clarity, we define a special alternate sequence of configurations and atomic steps, which is not an execution with respect to the two axioms, but that we call an agent-free execution and which is used to define the agent-stabilizing notion below. An agent-free execution is an alternated sequence of configurations and atomic steps, such that the initial configuration of the sequence satisfies $\neg Present_p$ for all p and only algorithm steps are computed in this sequence. The set of agent-free executions starting at L (where L is a set of configurations) is the collection of agent-free executions such that the initial configuration is in L. We define also the agent-free projection of an execution : it is the alternate sequence of configurations and atomic steps, where the agent-part of the registers is masked and agent-steps are ignored.

Finally, we define the type of failures that we will consider by defining initial configurations. There is no constraint on an initial configuration, except the fact that agent circulation rules must be correct (non corruptible).

Let $\mathcal{S}$ be a specification (that is a predicate on executions). A system is agent-stabilizing for $\mathcal{S}$ iff there exists a subset $\mathcal{L}$ of the set of configurations satisfying :

Convergence. For every configuration C, for every execution starting at C, the system reaches a configuration L in $\mathcal{L}$;

Correctness. For very configuration L in $\mathcal{L}$, every execution with L as initial configuration satisfies the specification $\mathcal{S}$;

Independence. For very configuration L in $\mathcal{L}$, every agent-free execution with L as initial configuration satisfies the specification $\mathcal{S}$;

Non-interference. For every configuration L in $\mathcal{L}$, the agent-free projection of every execution with L as initial configuration satisfies the specification $\mathcal{S}$;

Finite time to live. Every execution has the property that, after a bounded number of agent-steps, an agent is always removed from the system.

Convergence and correctness parts of this definition meet the traditional definition of a self-stabilizing algorithm. The independence part ensures that the algorithm will not rely on the agent to give a service : e.g. an agent-stabilizing system yielding a token circulation cannot define the token as the agent. Thus, an agent is indeed a tool to gain convergence, not to ensure correctness. Moreover, the non-interference property enforces the independence property by ensuring that an agent cannot perturb the behavior of a fault-free system. Finally, the finite time to live property ensures that an agent will not stay in the system forever and that an agent is a sporadic tool, used only from time to time.

3 General Properties of Agent-Stabilizing Systems

The study of the convergence property of an agent-stabilizing system is as difficult as the convergence property of a self-stabilizing system, because executions can start from any initial configuration. Moreover, the traditional fault model of self-stabilization does not include the corruption of the code. Here, we accept some code corruption (algorithm code corruption), making the study still harder.

We will prove in this part that without loss of generality, for proving the convergence property of an agent-stabilizing system, the set of initial configurations can be restricted to those in which code is correct and there is exactly one agent, just "correctly" installed at some processor p.

Theorem 1. *Assume the finite time to live property. Then, for proving convergence of an agent-stabilizing system, it is sufficient to consider as initial configurations, those following the installation of a unique agent.*

Sketch of Proof. Let C be an arbitrary configuration with n processors and p agents at processors $P_1, \ldots, P_p$ (thus, $Present_i$ is true for $i \in \{P_j, 1 \leq j \leq p)$. Let $\mathcal{E} = C\, s_1\, C_1 \ldots$ be an arbitrary execution with initial configuration C. $\mathcal{E}$ is fair, thus there are infinitely many agent steps in the execution $\mathcal{E}$. The property "finite time to live" is assumed, thus the execution reaches necessarily a configuration in which an agent is removed from the system, leading to a configuration C', with one agent less than in C.

The axiom "uniqueness of creation" involve that while there is at least one agent in the system (thus one processor verifying "$Present_p$ is true"), the agent installation step cannot be performed on any other processor in the system. By recursion on the number of agents in the system, the system reaches a configuration C^p where there is no agents in the system (potentially, $C^p = C$ if there was no agent in the initial configuration).

The execution $\mathcal{E}$ also satisfies the axiom "liveness of creation" ; there is a finite number of configurations between C and C^p, thus at least one agent will be present in a configuration A after C^p. The only atomic step which could install an agent is the "Agent installation" step, thus there is always a configuration A reachable from C where a single agent is just installed in the system. □

4 Agent Circulation Rules

One of the main goals of the agent is to traverse the network. We will present here three algorithms for achieving a complete graph traversal. Two of them (depth first circulation and switch algorithm) hold for arbitrary graphs, whereas the third is an anonymous ring traversal. We will also show that, in any agent-system using one of these algorithms as agent circulation rules, it is easy to implement the finite time to live property.

Depth First Circulation.
The algorithm assumes that every processor of the system is tagged with a color black or white. Assume that agent has just been created. The briefcase carries the direction of the traversal (Up or Down). The processor visited by the agent for the first time (with a Down Briefcase) flips its color, chooses as a parent the previous processor, chooses a neighbor tagged with a color different from its new color and sends the agent to this neighbor, with the briefcase Down. When the agent reaches a dead end (i.e. is sent down to a processor with no neighbors with the right color), it is sent back by this processor with Up in the Briefcase. When a processor receives an agent with Up in the Briefcase, if it is a dead end, it sends the agent Up, if not, it selects a new neighbor to color and sends the agent down.

Moreover, the briefcase holds a *hop* counter, namely the number of steps since the installation. This counter is incremented each time the agent moves. If the agent makes more than $4|E|$ steps (a bound greater than 2 times the number of edges of the depth first spanning tree), the agent is destroyed (next processor is $\perp$). More formally, in figure 1 the description of the traversal algorithm is given. For this, we define :

$$\theta(\Gamma_i, c) = \begin{cases} \perp & \text{if } \forall p \in \Gamma_i, color_p \neq c \\ p \text{ s.t. } color_p = c \text{ and } p \in \Gamma_i & \text{else} \end{cases} \quad \text{and also} \quad \begin{array}{l} \overline{black} = white \\ \overline{white} = black \end{array}$$

We claim that this algorithm produces a complete traversal of the system even in the case of transient failures because, as we will show in theorem 3, at each execution of rule 1, the number of processors of the same color strictly increases, until reaching the total number of processors in the system.

Theorem 2. *Every fair agent-system using the depth first circulation algorithm of figure 1 as agent circulation algorithm satisfies the finite time to live property.*

The proof is straightforward and uses the fairness of the system for showing that every agent increments its *hop* counter until the bound is reached.

Remark that the bound is large enough to allow a complete depth first traversal of the system : it is 2 times the number of edges, and in a depth first traversal, each edge of the spanning tree is used 2 times (one time for going down to the leaves, one time for going up to the root). We will now prove that this algorithm produces a complete graph traversal when there is a single agent in the system.

Let $p, q \in P$. We say that $p \sim q$ iff $color_p = color_q$ and $(p, q) \in neighbours$. In the sequel, the term "connected components" refers to the connected components of the graph of the relation $\sim$.

Lemma 1. *Let C be a configuration with m connected components $(M_1, \ldots M_m)$ and let p be a processor with an agent just correctly installed. Let M_j be the connected component of p.*

```
**Local:** $\langle parent_i \in \Gamma_i \bigcup \{\perp\}$ — $color_i \in \{black, white\}\rangle$
**Briefcase:** $\langle Briefcase_i \in (\{down, up\}, hop), 1 \le hop \le 2.|E|\rangle$
**Constant rule:** At each rule, we add the following statement:
        if $hop > 2|E|$ then $Next_i \leftarrow \perp$.

 – Agent rules :
    1. $Prev_i = \perp \vee \big((Briefcase_i = (down, hop)) \wedge (\theta(\Gamma_i, color_{Prev_i}) \neq \perp)\big) \longrightarrow$
       $\begin{cases} parent_i \leftarrow Prev_i \\ Next_i \leftarrow \theta(\Gamma_i, color_i) \\ color_i \leftarrow \overline{color_i} \\ \text{if } Prev_i = \perp, hop = 0 \\ Briefcase_i \leftarrow (down, hop + 1) \end{cases}$
       (* going down *)
    2. $(Prev_i \neq \perp) \wedge (Briefcase_i = (down, hop)) \wedge (\theta(\Gamma_i, color_{Prev_i}) = \perp) \longrightarrow$
       $\begin{cases} parent_i \leftarrow Prev_i \\ Next_i \leftarrow Prev_i \\ Briefcase_i \leftarrow (up, hop + 1) \\ color_i \leftarrow \overline{color_i} \end{cases}$
       (* reaching a dead-end *)
    3. $(Prev_i \neq \perp) \wedge (Briefcase_i = (up, hop)) \wedge (\theta(\Gamma_i, \overline{color_i}) \neq \perp) \longrightarrow$
       $\begin{cases} Next_i \leftarrow \theta(\Gamma_i, \overline{color_i}) \\ Briefcase_i \leftarrow (down, hop + 1) \end{cases}$
       (* done one branch, go down for others *)
    4. $(Prev_i \neq \perp) \wedge (Briefcase_i = (up, hop)) \wedge (\theta(\Gamma_i, \overline{color_i}) = \perp) \longrightarrow$
       $\begin{cases} Next_i \leftarrow parent_i \\ Briefcase_i \leftarrow (up, hop + 1) \end{cases}$
       (* completely done one node, going up*)
```

Fig. 1. Depth first agent traversal algorithm for anonymous networks.

In every execution $\mathcal{E} = C\, s_1 \ldots C' \ldots$ there is a configuration such that the agent is in p, has visited all processors in M_j and every processor in M_j has flipped its color.

Theorem 3. *Consider the same configuration C, as in lemma 1.*
In every execution $\mathcal{E} = C\, s_1 \ldots C' \ldots$ there is a configuration such that there is no agent, one agent has visited all processors of the system and all processors in C' have the same color.

Sketch of Proof. Lemma 1 states that from an initial configuration C, the system reaches a configuration C_1, with all processors of M_j (the connected component of p) visited and having flipped their color. If there is only one connected component in C, then the theorem is true (rule 4 set $Next$ to $\perp$ in p and when p makes an atomic step, it removes the agent). If there are $m > 1$ connected components in C, then there are $1 \le n < m$ connected components in C_1 (because all processors of M_j flipped their color, thus merged with the neighbor connected component). Thus, lemma 1 applies again in a configuration C'_1 with the agent removed from p (rule 4) and installed in a processor p' (liveness of the agent).

Recursion on the number of connected components in C, proves the theorem. □

The Switch Algorithm. The switch algorithm is another anonymous graph traversal algorithm. The algorithm presented in [12] is defined for semi-uniform Eulerian networks, but as we will show now, it works for uniform Eulerian networks with the agent-stabilizing system assumptions. This algorithm is an improvement of the depth first circulation because here circulation is decided purely locally, without communicating with the neighbors of the processor which holds the agent. Moreover, in the depth first token circulation, the agent performs a complete graph traversal only after all connected components merged into one. Here, the algorithm converges with only one agent installation. This has a price : an agent circulation round takes $2 \times n \cdot |E|$ agent steps, even after stabilization, while an agent circulation round with the depth first circulation takes only $2|E'|$ agent steps (E' being the set of covering edges of the depth first tree).

The idea of the algorithm is the following. Suppose that each node has ordered its (outgoing) edges and let $PointsTo$ be a pointer on some of these edges. When the token is at some node, it is passed to the neighbor pointed by $PointsTo$, then $PointsTo$ is set to the next edge in the list (which is managed circularly).

In [12], it is proved that, starting with a unique token, this algorithm eventually performs an Eulerian traversal of the network.

We propose here a slight modification of this algorithm so that it performs a complete traversal of an anonymous Eulerian agent-system. The initiator (with $Prev = \perp$) will set the Briefcase to 0 and every time the agent performs an agent step, Briefcase will be incremented by 1. If a processor holds the agent and if Briefcase is greater than $2 \times n \cdot |E|$ (where E is the set of edges of the system and n is the number of processors in the system), then this processor sets $Next$ to $\perp$, destroying the agent. The agent rules are presented more formally in figure 2. We will now prove that this circulation algorithm satisfies the finite time to live property.

Theorem 4. *Every fair agent-system using the switch circulation algorithm of figure 2 as an agent circulation algorithm satisfies the Finite Time to Live property.*

The proof is straightforward and uses the fairness of the system for showing that the Briefcase counter is incremented until reaching the bound.

Theorem 5. *The switch circulation algorithm is a circulation algorithm, which performs a complete graph traversal from a configuration in which an agent installation follows a configuration without agents.*

We use the convergence property of the switch, proven in [12], and the convergence time, which is $n \cdot |E|$ to state that after $n \cdot |E|$ agent steps, every edge, thus every node of the system has been visited by the agent.

```
Local: $\langle PointsTo_i \in \Gamma_i$
Briefcase: $\langle 0 \le Briefcase \le 2 \times n \cdot |E| + 1\rangle$
Operator: $Inc(PointsTo_i)$ increments $PointsTo_i$ by 1 modulo $Degree_i$.

  – Agent rules :
      1. $Prev = \perp \longrightarrow \begin{cases} Inc(PointsTo_i) \\ Next_i = PointsTo_i \\ Briefcase_i = 0 \end{cases}$
      2. $(Prev \neq \perp) \wedge (Briefcase < 2 \times n \cdot |E|) \longrightarrow$
         $\begin{cases} Inc(PointsTo_i) \\ Next_i = PointsTo_i \\ Briefcase_i = Briefcase_i + 1 \end{cases}$
      3. $(Prev \neq \perp) \wedge (Briefcase \ge 2 \times n \cdot |E|) \longrightarrow \{ Next_i = \perp$
```

Fig. 2. Switch circulation algorithm

Anonymous Oriented Ring Circulation.
In such a topology circulation is trivial : the agent rule consists of sending the agent to the successor and counting in the Briefcase the number of processors already seen. If this number is greater than or equal to the number of processors in the system, the agent is destroyed. With this algorithm, the finite time to live property is obvious, as is the correctness property of the agent circulation : starting from a configuration with only one agent, just installed at some processor p, this agent visits all the processors in the system. Furthermore, every new agent is eventually destroyed by its initiator.

5 Examples Illustrating the Power of Agents

We will first present general transformations which, given a self-stabilizing solution for semi-uniform networks or for networks with distinct identifiers, automatically transform them into agent-stabilizing solutions for the same problem, but for completely anonymous networks (uniform networks). The basic idea is that a generic agent algorithm, with an empty set of algorithm rules, either distinguishes a particular processor or gives distinct identifiers in an anonymous networks. By adding the rules of the self-stabilizing solution as algorithm rules one gets an agent stabilizing solution for anonymous network.

Let us briefly describe the ideas of the transformation from uniform to semi-uniform. Note that an algorithm that always declares the processor in which the agent is created as the leader, does not satisfies the non-interference property. Thus, the agent must perform a complete traversal (using the circulation rules) for checking whether or not there is only one distinguished processor. If several distinguished processors (resulting from a corruption) are found, all but the first are canceled and if no one is found, the initiator becomes distinguished.

Note that there is no algorithm rule in this solution, then it can be combined with any self-stabilizing solution for semi-uniform networks. The main advantage of automatic transformations is that there are automatic. But they very seldom yield the most efficient solution. In the second part, we will give another illustration of the power of agents, by giving efficient specific solutions to some classical problems.

5.1 General Transformers

Leader Election within a Uniform Network. This transformer allows a self-stabilizing system designed for semi-uniform networks to work for uniform networks. This is achieved by solving the Leader election problem.

Starting at the initiator, the agent performs a complete graph traversal using the algorithm in figure 1. We proved that this algorithm eventually performs a complete graph traversal, and it is obvious that after the first traversal, no processor has $prev|_p = \perp$ Then we can ensure that there is no processor with $parent_p = \perp$ after the first agent traversal, by adding $parent_p = Prev$ to the action part of rule 4 (the rule which is used when every neighbor of this node has been visited).

The algorithm is simple : the agent carries in the briefcase the information "did I meet a leader yet" (boolean value). If this information is true when the agent meets a leader, the leader is destroyed, if the agent meets the initiator (which is the only processor which has $parent_i = \perp$ for every agent installation after the first complete traversal) and this information is false, this processor becomes the leader.

Lemma 2. *Consider a configuration C of Depth first agent traversal, in which all processors have the same $color_i$ and an agent has just been installed at processor p. The agent performs a complete graph traversal and rule 4 is executed exactly once by each processor before the agent is removed from the system.*

Theorem 6. *The leader election algorithm is agent-stabilizing.*

Sketch of Proof. The agent has a finite time to live, as it was proven in theorem 2. It is then legitimate to use the result of theorem 1 and to consider for the convergence property a configuration with one agent in the system. This agent will eventually traverse every node and come back to the initiator (theorem 3). According to algorithm rules, there will remain one and only one leader in the system, thus convergence is verified. Consider a legitimate configuration. The algorithm rules are empty (thus the agent-system satisfies independence), and when an agent is installed, it will perform a complete graph traversal, visiting every node and executing rule 4 at each node exactly once (lemma 2 applies, the configuration is legitimate, thus every node is of the same color). Thus, it will visit the leader and note it in the briefcase. Then, the initiator will not be designated as the leader and the correctness property is satisfied. Moreover, the leader stays the leader, thus non-interference property is also verified. □

Network Naming. The naming problem is to assign to each processor of the network a unique identity. We consider that there is a set of identifiers (at least as large as the network size), and that identities should be chosen within this set. The agent carries in the briefcase a boolean tag for each element of this set. When the agent reaches a processor p, if p uses an identifier already used, its identifier is set to the first available identifier, and this identifier is tagged in the briefcase ; if p uses an identifier which is not yet used, this identifier is tagged in the briefcase. The agent performs a complete graph traversal according to figure 1 algorithm and actions are executed when the guard of rule 4 is true.

Theorem 7. *The network naming algorithm is agent-stabilizing.*

Sketch of Proof. As it was said before, the Finite time to live property is inherited from the agent circulation algorithm. Like for leader election, independence is obvious since the algorithm has no rule. For proving the correctness property, we have to show that the set of legitimate configurations is closed. Let us define legitimate configurations as configurations such that $\forall p, q \in P, tag_p \neq tag_q$. In a token circulation, rule 4 of depth first traversal applies at most once (theorem 2). Thus, the agent will never meet a processor with a tag equal to an already used tag. Thus, the rule will never be applied and the configuration will not change with respect to the tags. Let us consider an arbitrary configuration C' with an agent just installed at processor p. If this configuration is not legitimate, then there exists a processor p and a processor q tagged by the same name. Theorem 3 states that in every execution with initial configuration C', there is a configuration C'' reached after the agent visited every node. When this agent has been installed, $Briefcase$ was set to nil ; it first reached either p or q. If it reached p, tag_p is an element of $Briefcase$, and when it reaches q, tag_q is set to a no yet used tag. □

5.2 Local Mutual Exclusion within a Uniform Network

With the previous algorithm, we can easily transform every self-stabilizing algorithm designed for networks with identities to an agent-stabilizing solution for anonymous networks. Note that we assume that the agent can know the different Ids already seen in the network (briefcase must have $n \log(n)$ bits). Obviously, in specific cases, better solutions can be found. We propose now a solution for solving Local Mutual Exclusion within a uniform anonymous network. Local Mutual Exclusion is a generalization of the dining philosophers problem [7], and can be defined as "having a notion of privilege on processors, two neighbors cannot be privileged at the same time".

The solution is based on the self-stabilizing local mutual exclusion algorithm proposed in [1]. Every processor has a variable b in range $[0; n^2 - 1]$. Assuming that neighbors have different b values, then a cyclic comparison modulo n^2 of the b values yields an acyclic orientation of edges (from greater to lesser). If we consider that every sink (i.e. a node with only ingoing edges) is privileged, then two neighbors never can be privileged simultaneously. To pass the privilege, b is set to the maximum of the neighbors value plus one (modulo n^2), and in [1] it is shown that this rule provides a cyclic selection of every processor in the system.

It is also shown that this solution is based upon the fact that a processor has a value different from the values of its neighbors, and identifiers are used to implement that. An agent can be easily used in an anonymous network, for creating asymmetry between a processor and its neighbors.

```
◇ Algorithm rules
    1. $b < min^{[n^2]}_{q \in \Gamma_p}(b|_q) \longrightarrow b \leftarrow max^{[n^2]}_{q \in \Gamma_p}(b|_q + 1)$
◇ Agent rules
    1. Every rules of figure 1, composed with
    2. $\exists q \in \Gamma_p$ s.t. $b|_q = b \longrightarrow b \leftarrow max^{[n^2]}_{q \in \Gamma_p}(b|_q + 1)$
```

Fig. 3. Anonymous uniform Local mutual exclusion agent-stabilizing algorithm.

Theorem 8. *The local mutual exclusion algorithm of figure 3 is agent-stabilizing.*

Sketch of Proof. This algorithm uses the depth first circulation algorithm, thus the Finite Time to live property is satisfied. Then, according to theorem 1, we can consider for proving convergence a configuration with a single agent just installed at some node. Let C be such a configuration. Either, $\forall p \in P, \forall q \in \Gamma_p, b|_p \neq b|_q$ and it was proven in [1] that the algorithm converges or $\exists p \in P, q \in \Gamma_p, b|_p = b|_q$. In this case, the agent will eventually visit p and q (theorem 3), either p first or q first. If the first visited processor is p (resp. q), agent rule 2 will be enabled, thus the system will reach a configuration in which $\exists p \in P, q \in \Gamma_p, b|_p = b|_q$ is false. That will remain false in the sequel of the execution. Then convergence is proven.

Starting from a legitimate configuration, the agent does not modify the b bit of any processor (a legitimate configuration satisfies $\forall p \in P, \forall q \in \Gamma_p, b|_p \neq b|_q$). Correctness of the algorithm has been proven in [1]. Thus every execution from a legitimate initial configuration is correct. In such an execution, the agent does not modify the behavior of the algorithm, then non interference is satisfied. Finally, the independence property is also satisfied since every agent-free execution with a legitimate initial configuration is correct. □

5.3 Token Circulation on a Bidirectional Uniform Oriented Ring

As it was shown in [5] a simple non self-stabilizing algorithm for achieving token circulation on a ring with a leader can be made easily agent-stabilizing, assuming that the agent is always installed at the leader. We will prove that this assumption can be removed. Furthermore, we propose a token circulation algorithm on a bidirectional uniform oriented ring which is agent-stabilizing.

```
  **Predicates:**
  $sensitive(p) \equiv b|_{p+1} = b|_p \neq b|_{p-1}$
  $HasToken(p) \equiv b|_{p-1} = 1 \wedge b|_p = 0$,
  $Context(p) = Tokens$ if $HasToken(p)$, $NoTokens$ else.
◇ **Algorithm rules**
  (a) $sensitive(p) \wedge b|_p = 1 \longrightarrow b|_p = 0$
  (b) $sensitive(p) \wedge b|_p = 0 \longrightarrow b|_p = 1$
◇ **Patch 1 Rules**
  $color|_p \neq Seen \wedge sensitive(p) \wedge b|_p = 1 \longrightarrow b|_p = 0$
  $color|_p \neq Seen \wedge sensitive(p) \wedge b|_p = 0 \longrightarrow b|_p = 1, color|_p = Seen$
◇ **Patch 2 Rules** $\emptyset$
◇ **Agent rules** (Rules of ring circulation, in conjunction with) :
   1. $Prev = \perp \longrightarrow$
      $color|_p = NotSeen, Briefcase.context = Context(p)$, install Patch 1.
   2. $Prev \neq \perp \wedge Briefcase.hop = n \longrightarrow$
      if $Briefcase.context = NoTokens$ and $color|_p = NotSeen$,
              create a new token
      elsif $Briefcase.context = Tokens$ and $color|_p = Seen$,
              install Patch 2, $Briefcase.hop = 0$
      else install Algorithm.
   3. $Prev \neq \perp \wedge Briefcase.context = Tokens \longrightarrow$
      if $HasToken(p)$ destroy the token
   4. $Prev \neq \perp \wedge Briefcase.context = NoTokens \longrightarrow$
      $Briefcase.context = Context(p)$
```

Fig. 4. Anonymous uniform token circulation agent-stabilizing algorithm

The idea is the following : Let b be a boolean variable stored at each node. Let us say that processor p has the token in configuration C if in C, $b|_p = 0$ and $b|_{p-1} = 1$ (the ring is oriented). Let us say that a processor is sensitive if it has the same b value as its successor and a b value different from its predecessor (then we allow the token to be or not to be sensitive). The algorithm performs the following rules : If a processor p is sensitive, it is allowed to flip its bit. Let us denote a configuration by a word, like 01...10, where a digit represents the value of the b bit of a processor. A legitimate configuration is of the form $1^p 0^{n-p}$.

This algorithm is not self-stabilizing. The solution that we propose here to stabilize uses agents. The first idea is simple : an agent traverses the ring in the same direction as the tokens, carrying in the briefcase the information "did I see a token until now". If so, every other token can be destroyed by setting the b bit to 1. When the agent comes back to the Initiator, if there is at least one token in the system (Briefcase = Tokens), then the agent simply disappears, if not a token is created at the Initiator (flipping the bit) and then the agent disappears.

The circulation algorithm used here is the oriented anonymous ring circulation. It is detected that the agent has performed a complete traversal by checking if $Briefcase$ has the value n when the agent is back to the Initiator. This solution works only if tokens do not move. Then the next problem to solve is to ensure that the agent will meet the supplementary tokens, if any. For that, the initiator receives a patch from the agent : instead of letting every token pass through him while the agent is inside the ring, it allows only one token to pass and freezes the others.

The last problem is: if the agent saw one or more tokens while another token passed through the initiator, when the agent will come back to the initiator there will be two tokens in the system. But this situation can be detected by the initiator. One solution could be not to let the agent overtake a token, another not to let a token pass through the initiator. But both would fail with the non-interference requirement. A solution matching the requirements is to give to the initiator the responsibility to destroy the supplementary token when it reaches it. Note that, if a bounded convergence time is mandatory (related to the speed of the agent), it could be better to let the initiator freeze one token while the agent performs a second round for deleting the other. The solution is presented more formally in figure 4.

Theorem 9. *The agent-algorithm of figure 4 is agent-stabilizing*

Sketch of Proof. Let us first prove the independence property : a legitimate configuration is defined by 1^p0^{n-p}. In such a configuration, if p is greater than 1, the system can choose between algorithm rules a and b and reaches either the configuration $1^{p+1}0^{n-p-1}$ (the token moves one step backward) or the configuration $1^{p-1}0^{n-p+1}$ (the token does not move). If p is 1, then the system is only allowed to reach the configuration 110^{n-2} (the token moves backward), and if p is $n-1$ the system is only allowed to reach the configuration $1^{n-2}00$. In every reachable configuration the number of tokens is one and every processor of the system eventually has the token. So, starting from a legitimate configuration if there are no agent steps, the behavior is correct.

We prove then the non-interference property : if there is exactly one token in the system, either it will go through p and the agent will never meet this token, $Briefcase.context$ is $NoTokens$, but the agent is silently destroyed, or the agent meets the token, $Briefcase.context$ is $Tokens$ when the agent reaches back p and the agent is silently destroyed : the configuration remains legitimate. Then, starting from a legitimate configuration, the agent has no effect on the behavior of the system (the token is neither moved nor blocked). This proves also the correctness property of the algorithm.

The finite time to live property is a consequence of the circulation algorithm (an agent performs at most 2 rounds of the ring). For the convergence property, we consider an initial configuration with one agent installed at processor p. If there are no tokens in the system, when the agent reaches back to p, $Briefcase.context$ is $NoTokens$, thus rule 3 applies and the following configuration is legitimate ; if there are more than one token in the system, either the agent meets the supplementary tokens and then they are destroyed, or these tokens are frozen (they cannot move to the next processor) by the first patched algorithm rule. At most one token could pass through the Initiator during the first round. If the agent has met a token and a token has pass through the initiator, the agent will make another round, destroying one token and no token could pass through the initiator within this round. Thus, the agent has to meet the tokens, and every supplementary token is destroyed. Thus, the configuration following the configuration after the agent came back to the initiator for the second time if necessary is legitimate. □

6 Conclusions

In this paper, we have formally defined the notion of agent (in the context of stabilizing systems), and agent-stabilizing systems. We proposed a set of axioms for having a powerful tool for dealing with failure tolerance. We illustrated the use and the power of this tool on two static problems, which provide a general transformation scheme and two dynamic classical problems.

The agent is a special message that circulates in the system, checks its consistency and mends it if faults are detected. It is created by a lower layer which was not described here, but whose properties are well defined (uniqueness and liveness of creation). We also defined a set of constraints (properties), that the system must satisfy to be considered as agent-stabilizing. In addition to the classical correctness and convergence properties, we added independence and non-interference notions. These notions ensure that the system will not rely on the agent in the (normal) case without failure, and that the presence of the agent in the normal case is not disturbing the system.

We provided a general transformation for self-stabilizing algorithms designed for non-anonymous networks into agent-stabilizing algorithms for anonymous networks. Finally, We have shown that the agent can also be used to transform non-stabilizing algorithms, like the token circulation on anonymous ring with two states per processor, into stabilizing ones.

References

1. J.Beauquier, A.K.Datta, M.Gradinariu, and F.Magniette. Self-stabilizing local mutual exclusion and daemon refinement. In DISC 2000 Distributed Computing 14th International Symposium, Springer-Verlag LNCS:1914, 2000.
2. J.Beauquier, C.Genolini, and S. Kutten. Optimal reactive k-stabilization: the case of mutual exclusion. In PODC'99 Proceedings of the Eighteenth Annual ACM Symposium on Principles of Distributed Computing, pages 209-218, 1999.
3. E.W.Dijkstra. Self stabilizing systems in spite of distributed control. Communications of the Association of the Computing Machinery, 17:643-644, 1974.
4. M.G.Gouda. The triumph and tribulation of system stabilization. In WDAG'95 Distributed Algorithms 9th International Workshop Proceedings, Springer-Verlag LNCS:972, pages 1-18, 1995.
5. S.Ghosh. Agents, distributed algorithms, and stabilization. In Computing and Combinatorics (COCOON 2000), Springer-Verlag LNCS:1858, pages 242-251, 2000.

6. S.Ghosh and A.Gupta. An exercise in fault-containment: self-stabilizing leader election. Information Processing Letters, 59:281-288, 1996.
7. S.T.Huang. The Fuzzy Philosophers. In Parallel and Distributed Processing (IPDPS Workshops 2000), Springer-Verlag LNCS:1800, pages 130-136, 2000.
8. S.Kwek. On a Simple Depth-First Search Strategy for Exploring Unknown Graphs. In WADS97 Proc. 5th Worksh. Algorithms and Data Structures, Springer-Verlag LNCS:1272, 1997.
9. D.Kotz, R.Gray, D.Rus, S.Nog and G.Cybenko, Mobile Agents for Mobile Computing. In Technical Report PCS-TR96-285, May 1996, Computer Science Department, Dartmouth College
10. M.Schneider. Self-stabilization. ACM Computing Surveys, 25:45-67, 1993.
11. G.Tel. Introduction to distributed algorithms. Cambridge University Press, 1994.
12. S.Tixeuil, Auto-stabilisation Efficace. Thèse de doctorat, LRI, January 2000