A Local Enumeration Protocol in Spite of Corrupted Data

Brahim Hamid and Mohamed Mosbah
LaBRI-
Université Bordeaux-1,
351, cours de la libération,
33405 Talence, France,
Email: {hamid,mosbah}@labri.fr

***Abstract*— We present a novel self-stabilizing version of Mazurkiewicz enumeration algorithm [1]. The initial version is based on local rules to enumerate nodes on an anonymous network. [2] presented the first self-stabilizing version of this algorithm which tolerates transient failures with an extension of messages complexity. Our version is based on local detection and correction of transient failures. Therefore, it ensures the fault-tolerance property without adding messages or reduces the messages' number of other version. In addition, we have developed an interface based on the Visidia platform to simulate faults through a graphical user interface. The implementation of the presented algorithm in this platform shows its dynamic execution and validates its correction. The asynchronous message passing version of the presented protocol shows the transformation of distributed algorithms encoded in local computations model, as a high model, to message passing model, as a weaker model. The transformation is given in spite of corrupted information with simple changes.**

I. INTRODUCTION

Distributed computing systems are becoming larger and larger, heterogeneous and complex. Since the applications running on these systems require the cooperation of many components, they are prone to failures and errors of many different types, leading to inconsistent executions. Hence, a desirable feature of a computation in a distributed system is fault-tolerance. A particularly suitable approach to deal with such a feature is to design self-stabilizing algorithms [3], [4]. Since it has many applications in dynamic networks, distributed naming or enumeration of a graph has gained much attention recently. In fact, many distributed applications are based on names to share resources, to uniquely identify entities, to refer to location, and so on. Theoretically, enumeration may be used to draw and to quantify a border between possibility and impossibility of some tasks in distributed computing. The naming problem belongs to the family of coordination problems. In such a problem, a set of processes of size N should decide on distinct values from set of size N.

This paper is based on "A Local Self-Stabilizing Enumeration Algorithm," by B. hamid and M. Mosbah, which appeared in the Proceedings of the Distributed Applications and Interoperable Systems, 6th IFIP WG 6.1 International Conference, DAIS 2006, Bologna, Italy, June 14-16, 2006. © 2006 Springer.

In this paper, we propose a general approach based on graph relabeling systems and its extension to message passing system to preserve the global enumeration of a graph using only local information.

The concept of self-stabilization [5] is introduced to design a system which tolerates transient failures. Informally, self-stabilizing algorithms ensure that after any failure, the system will automatically recover to reach a correct configuration in a finite time. In general, self-stabilizing algorithms are constructed in such a way that a given process will continue to function correctly in spite of intermittent faults. The stabilizing algorithms are optimistic, they guarantee a return to a correct behavior within a finite time after all faulty behaviors cease. Self-stabilizing algorithms protect against transient failures, since they can automatically repair any fault in the system. The term *fault* refers to failure.

A distributed system is represented as a connected, undirected graph G denoted by (V, E). Nodes V represent processes and edges E represent bidirectional communication links. In this work, we use a model based on local computations, and particularly graph relabeling systems, to encode distributed algorithms. Local computations in graphs are powerful models which provide general tools for studying distributed algorithms. In the following, the notion of relabeling sequences corresponds to a *sequential* execution. For a local computations we can define equivalent parallel rewritings, when two consecutive relabeling steps concerning non-overlapping balls may be applied in any order. They also can be applied concurrently. Protocols presented in this study use forms of computations near to *cellular automata* or transition state models [5]. At each step of the computation, labels are modified for a given ball composed of some node and its neighbors. The modification is given according to rules depending only on the labels of nodes composing this ball. In this work we are interested in formalizing and studying enumeration problem using means of local computations. The study of such a problem in high model allows us to deduce some properties on weaker models.

Moreover, we extend the model to deal with self-stabilization in asynchronous message passing systems. In the considered networks, processes communicate and synchronize by sending and receiving messages through the links. There is no assumption about the relative speed of processes or message transfer delay, the networks are *asynchronous*. Each node communicates only with its neighbors. The links are reliable and the process can fail and recover in a finite time. The failures that are tolerated in such a system are the transient failures of processes.

An anonymous network is a network where all nodes execute the same algorithm without a unique identity for each node. In general, the task solved by an enumeration algorithm is the assignement of a different *name* to each node of an anonymous network. So, such an algorithm may be used as a preprocessing task of many algorithms based on the identities. As it is well-known, many problems have no solutions in anonymous networks. The motivations of this work are on the one hand to design an enumeration protocol in anonymous networks using only local computations [1]. On the other hand, we show the adaptation of our developed framework [6] to enumeration algorithm in the presence of transient failures.

We are interested in the study of the Mazurkiewicz's enumeration algorithm [1] based on local computations. Mazurkiewicz's algorithm is a distributed algorithm to enumerate nodes in an anonymous minimal-covering graph when its size is known. A distributed enumeration algorithm on a graph G is a distributed algorithm such that the result of any computation is a labeling of the nodes that is a bijection from $V(G)$ to $1, 2, ..., |V(G)|$. [2] proposed a version of self-stabilizing enumeration algorithm with a final stage in which each node computes locally the set of final names from the final mailbox. Before this stage the node can choose a name which is greater than the size of the graph.

Many self-stabilizing algorithms have been already designed [7], [8]. However, most of these works propose global solutions which require to involve the entire system. As networks grow fast, detecting and correcting errors globally is no longer feasible. The solutions that deal locally with detection and correction are rather essential because they are scalable and can be deployed even for large and evolving networks. Moreover, it is useful to have the correct (non faulty) parts of the network operating normally while recovering locally the faulty components. Few general approaches providing local solutions to self-stabilization have been proposed in [9]–[11].

In [6], we consider the problem of designing algorithms encoded by local computations in a distributed system with transient failures. The developed formal framework allows to design and prove fault-tolerant distributed algorithms. We introduce correction rules which can be applied by faulty nodes or by their neighbors in order to self repair the incorrect states. Such rules have higher priorities than the main rules of the algorithm which ensure that the failures are repaired before continuing the computation. Of course, we deal only with predefined faulty local configurations. A tool called *Visidia* [12], validating the local computations model has been implemented. The distributed system of Visidia is based on asynchronous message passing model. However, it has been assumed that components of such a system do not fail. In this work, we show a simulation of fault-tolerant enumeration algorithm using this platform.

The rest of the paper is organized as follows. Models of distributed systems including some element of graph theory, graph relabeling systems and message passing system are presented in Section 2. We start the Section 3 with our framework to design self-stabilizing systems, then we describe our solution to encode self-stabilizing enumeration algorithm. Section 4 shows its proofs of correctness and its complexity analysis and Section 5 shows an implementation of our protocol on the Visidia platform. Since previous protocol is based on local information, we present in Section 6 an asynchronous message passing version of this protocol. A short analysis is also given. Finally Section 7 concludes the paper with discussions about future works.

II. Preliminaries

A. Graphs

We introduce some terminologies and definitions about graphs. A graph is a collection of points called the nodes of the graph where some of them are connected by lines (adjacent) called the edges.

In each class of graphs we can use the following structure (V, E) to design a graph G, where V is the set of nodes and $E \subseteq V^2$ is the set of edges. We use sometimes the notations V_G, E_G to denote respectively V, E where $G = (V, E)$. Let $(u, v) \in E$, we say that u is a neighbor of v. A path p in G is a sequence $(v_0, ..., v_l)$ of nodes such that for each $i < l$, $(v_i, v_{i+1}) \in E$. The integer l is called the length of p. Then, a path $p = (v_0, v_1, ..., v_k)$ of non-zero length through the graph such that $v_0 = v_k$, is called a cycle of the graph. A path $P = (v_0, ..., v_l)$ is simple if it does not contain cycles. The set of neighbors of node u is denoted by $N(u) = \{v \in V/(u, v) \in E\}$. A ball center on u with radius l is the set $B_l(u) = \{u\} \cup \{v_j \in V/$ there exists a path $(v_0, ..., v_j)$ in G with $v_0 = u$ and $j \leq l\}$. We will say that $v \in V_G$ is an $l-$neighbor of u if v is in $B_l(u)$.

We say that the graph $G' = (V', E')$ is a subgraph of the graph $G = (V, E)$, if $V' \subseteq V$ and $E' \subseteq E$. A graph is connected if for any pair of its nodes, there exists a path between them. Let $V' \subseteq V$, we denote by $G|V'$ the restriction of G to the subgraph $(V', \{(u, v) \in E \cap V'^2\})$. We also denote by $G \setminus V'$ the graph $G|(V \setminus V')$. When V' is a singleton $V = \{v\}$, by abuse, we will use the notations $G|v$ and $G \setminus v$.

A graph morphism φ between two graphs $G = (V, E)$ and $G' = (V', E')$ is a surjective mapping $\varphi : V \rightarrow V'$ such that $\forall$ $(u, v) \in E$: $(\varphi(u), \varphi(v)) \in E'$ and $\forall$ $(u', v') \in E'$, $\exists(u, v) \in E$ such that $u' = \varphi(u)$ and $v' = \varphi(v)$.

The cardinality of set $V(G)$ (which is also the size of the corresponding network) is denoted by $\mid V(G) \mid$, and we assume that $\mid V(G) \mid= \mathcal{N}$. For any set A, we write 2^A (resp. A^n) to denote the set of all finite subsets of A

(resp. the set of all n-tuples, for $n \in \mathcal{N}$ of element of A), where $\mathcal{N}$ is the set of all positives integers.

The set $N_G(v)$ (resp.$N_h(w)$) is composed of all the neighbors of v in the graph G (resp. the neighbors of w in the graph H). We say that a graph G is a *covering* of a graph H if there exists a surjective homomorphism φ from G onto H such that for every node v of $V(G)$ the restriction of φ to $N_G(v)$ is a bijection onto $N_H(\varphi(v))$. In particular, $(\varphi(v), \varphi(u)) \in E(H)$ implies $(v, u) \in E(G)$. The covering is *proper* if G and H are not isomorphic. It is called connected if G (and thus also H) is connected. A graph G is called *minimal-covering* if every covering from G to some H is a bijection.

B. Graph Relabeling Systems (GRS) to Encode Distributed Algorithms

In [13], the authors proposed a powerful model to encode distributed algorithms. This model is based on the use of graph relabeling systems. It offers general tools to encode distributed algorithms, to prove their correctness and to understand their powers.

Let Σ be an alphabet. A *labeled graph* is a couple (G, L) where G is a graph and L is a *labeling function* i.e. a mapping from V_G to Σ. In the sequel, we assume that all the used labeling functions share the same alphabet. The labeled graph (H, L') is a sublabeled graph of (G, L) , if H is a subgraph of G and L' is the restriction of L to V_H. A labeled graph morphism φ: $(G, L) \longrightarrow (G', L')$ is a graph morphism from G to G' which preserves the labeling, that is, for any $x \in V_G$, $L'(\varphi(x)) = L(x)$. A labeled subgraph (G', L') of (G, L) is an *occurrence* of (H, L_H) in (G, L) if there exists an isomorphism φ between H and G' such that $\forall x \in V_H$, $L_H(x) = L'(\varphi(x))$.

A graph relabeling rule is a triple $R = (H, L', L_H)$ such that (H, L') and (H, L_H) are two labeled graphs. The labeled graph (H, L') is the *precondition part* of R and the labeled graph (H, L_H) is its *relabeling part*. Given a labeled graph $(G,\ L_i)$, an algorithm $\mathcal{R}$ is a finite set of relabeling rules. An *$\mathcal{R}$-computing step* is a couple of label functions (L_i, L_{i+1}) such that there exists a sublabeled graph (G', L') of (G, L_i) and a rule $R = (H, L', L_H) \in \mathcal{R}$ satisfying : (G', L') is an occurrence of (H, L_H) in (G, L_i), $\forall x \in V_G \setminus V_H$, $L_{i+1}(x) = L_i(x)$ and $\forall x \in V_H$, $L_{i+1}(x) = L_H(x)$. In other words, the labeling L_{i+1} is obtained from L_i by modifying all the labels of the elements of G' according to the labeling L_H. Such a computing step will be denoted by $L_i \xrightarrow[R]{} L_{i+1}$.

We say that the *computations in G are local* if a node $u \in V_G$ can only act on the set of its $l-$neighbors in a ball of a fixed radius l . Formally, this is equivalent to fix the diameter of the graphs H of the relabeling rules $R \in \mathcal{R}$ to $2l$. Any computation in G is a relabeling sequence $(L_0, L_1, \cdots, L_n)$ such that $\forall\, 0 \le i < n$, (L_i, L_{i+1}) is a computing step. We use the notation $L \xrightarrow[\mathcal{R},p]{} L'$ to say that there exists a local computations of length p starting from the label L to the label L' i.e. there exists a computation $(L_0, L_1,\ \cdots L_n)$ such that $L_0 = L, L_n = L'$.

A computation in G is finished if no relabeling rule can be applied on the current labeling. In this case, the corresponding labeled graph (G, L') is a result of the algorithm. We denote by $RES_{\mathcal{R}}(G)$ the set of all the results of the algorithm $\mathcal{R}(G)$. The execution time of a computation is the length of the corresponding relabeling sequence. Thus, a measure of the time complexity of a distributed algorithm, or of a graph relabeling system, will be the execution time of the longest computation of the algorithm. A measure of a space complexity of a distributed algorithm is the size of the label of a node of the graph. Obviously, the used concrete memory is the size of all the labels in the graph.

In this work, we consider only relabeling in ball of radius 1. That is, each one of them, may change its label according to rules depending only on labels of all nodes composing this ball. We assume that each node distinguishes its neighbors and knows their labels. In the sequel we use the set $B(v)$ to denote the set of (locally) ordered immediate neighbors of v which is an input data.

C. Example of a Spanning tree Computation

Consider the following graph relabeling system used to compute a distributed spanning tree of a given graph $G = (V, E)$. Every process $v \in V$ has for label the quadruple :

- $B(v)$: the set of immediate neighbors of v which is an initial data,
- $St(v)$: the state of v that can have only 2 values: A to mean that v has been included in the tree or N when it is not yet in the tree,
- $father(v)$ is the father of v in the spanning-tree.
- $Sons(v)$: the ordered list of its sons in the tree.

When starting the algorithm, we choose a node v_0 as *the root* of the tree. The initial label function L_0 is defined by :

- $St(v_0) = A,\ father(v_0) = \perp,\ Sons(v_0) = \emptyset$;
- $\forall v \neq v_0,\ St(v) = N, Sons(v) = \emptyset, father(v) = \perp$.

Here, when $father(v) = \perp$ this means that v has no defined father.

The algorithm is described by a relabeling system $\mathcal{R}$ with one rule $R1$ defined on $\Sigma = \{2^{\mathcal{N}} \times \{N, A\} \times V_G \cup \{\perp\} \times 2^{\mathcal{N}}\}$:

Distributed Computation of a Spanning tree in Local Computations Model

```
R1 : Spanning rule acting on 2 nodes $v,\ u$
        Precondition :
        – $St(u) = N$
        – $\exists\ v \in B(u),\ St(v) = A$
        Relabeling :
        – $St(u) := A$
        – $father(u) := v$
        – $u$ is added to $Sons(v)$.
```

At any step of the computation, when a N-labeled node u finds a neighbor v with $St(v) = A$, this node u may decide to include itself in the tree by changing $St(u)$ to

A. Moreover, $father(u)$ is set to v and the node v adds u in its sons list. So at each computation step, the number of N-labeled nodes decreases of 1.

The computation finishes when all the nodes v are such that $St(v) = A$. And obviously we have a spanning tree of G rooted at v_0 defined by the third components (or the fourth components) of the labels of the nodes.

Property 2.1: If v the elected node of the graph $G = (V, E)$ initiates the execution of the Spanning Tree Module, eventually after the application of $\#V$ rules all the nodes u of G are labeled $Stage(u) = A$ and a spanning tree T rooted at v of G is built.

D. Message Passing Model

Contrary to the local computations model where communications are abstracted, in a message passing system, processors communicate by sending messages through bidirectional communication links. The network is also represented by an undirected graph in which each node represents a processor, and each edge is present between two nodes if their corresponding processors may communicate. An algorithm in such a system consists of a local program at each node. The program encodes the local actions that processor may make. They include modification of local variables, send messages to and receive messages from each of its neighbors in the corresponding graph topology.

More formally, each processor p_i is modeled using a set of state. Since the processor is identified with a particular node in the graph, edges incident to p_i are labeled using arbitrary set of numbers in the set $1 \cdots r$ where r is the degree of p_i. Processor p_i 's action takes as input a value of some states of p_i, messages sent by the neighbors. Then, it produces as output a new value of some states and also at most one message for each link between $1 \cdots r$. Messages previously sent by p_i and not yet delivered cannot influence p_i current step. At any time, a *configuration* or a global state of the system is a vector C composed of the states of all the processors at this time. An *initial configuration* is a vector of the initial states of all the processors.

To model a computation in a message passing system, we introduce two kinds of events. The computation event, representing a computation step of a processor p_i in which it applies action to its current accessible states. The other event is denoted by the delivery event responsible of the deliverance of message m from processor p_i to processor p_j.

So an execution shows the behavior of a system, which is a set of sequence of configurations and events. Such sequences depend of the task assigned to the system being modeled. Thus, they must satisfy some properties according to the tasks.

In the message passing model, a system is said to be asynchronous if there is no fixed upper bound on the duration of both computation and delivery events. In spite of many distributed applications use usual upper bounds on message delays and processor step times, it often suitable to design algorithms that are independent of any particular timing parameters, namely asynchronous algorithms.

In the asynchronous message passing model, an execution is said to, be *admissible* if each processor may take an infinite number of computations events, and every message sent is eventually delivered. Such requirements model the fact that processors do not fail and communication links are also reliable. It does not imply that the processors programs must contain an infinite loop. In this point of view, the termination of an algorithm is reached when actions not change the processor states after a certain point.

Complexity measures in the message passing model are related to the number of messages and the amount of time as the local computations model. We will focused on the worst-case. Such measures depend on the notion of the algorithm terminating or when algorithm reaches son specific configuration. So, we assume that each processor's state includes a subset of states to denote *terminated* states. After the reach of such states, actions of the program maps terminated states only to terminated states. Thus, we say that the algorithm has terminated when all processors are in the terminated states and no all messages are delivered.

The message complexity of an algorithm encoded in the message passing model is the maximum of the total number of message sent during all admissible executions of this algorithm. For the time complexity, we adopt the common approach assuming that the maximum message delay in any execution is one unit of time. Note such a measure is not used to prove the correctness of such algorithms. Then, to calculate the time complexity of an algorithm it suffices to take the running time until the termination.

Distributed computation of a spanning tree in message passing model

```
var  $father$ : integer init $0$;
     $root, in\text{-}tree$ : boolean init $false$;
     $Sons$ : set of integer init $emptyset$;
     $i, q$ : integer;

I : {For the initiator $r$ only, execute once:}
    $in\text{-}tree := true$;
    $root := true$;
    for ($i := 1$ to $deg(r)$) do  send<tok> via port $i$

T : {A message <tok> has arrived at $v_0$ from port $q$}
      if ($not\ in\text{-}tree$)
            $father := q$;
            $in\text{-}tree := true$;
            send<son> via port $q$;
            for ($i := 1$ to $deg(v_0)$) do  send<tok> via port $i$

S : {A message <son> has arrived at $v_0$ from port $q$}
      $Sons := Sons \cup \{q\}$
```

Property 2.2: Given a graph $G = (V, E)$ and en elected node v_0. The execution of the spanning tree algorithm involves, in the worst case, the send of $\Delta \times \#V$ **tok** messages and $\#V - 1$ **son** messages and uses at most $(\Delta + 1) \times \#V - 1$ time, where Δ is the arity of G.

III. The Local Stabilizing Enumeration Algorithm

A. Self-stabilizing Graph Relabeling Systems

We quote from *E.Dijkstra* [5] the following definition: *'a system is a self-stabilizing system if independently of the starting configuration the system will reach a legitimate configuration after a finite time'*. An algorithm is therefore called self-stabilizing if it eventually starts to behave correctly regardless of the initial configuration.

A *local configuration* of a process is composed by its state, the states of its neighbors and the states of its communication links. In this work we use the notion of local illegitimate configurations encoded by local computations [6]. For a labeled graph (G, λ), we say that a local configuration $f = (B_f, \lambda_f)$ is illegitimate for (G, λ), if there is no subgraph in (G, λ) which is isomorphic to f. In other words, there is no ball (neither sub-ball) of radius 1 in G which has the same labeling as f. Such labels are not used when the system runs in a correct manner.

Transient failures cause processes to change their states yielding illegitimate local configurations and therefore an illegitimate global configuration. A self-stabilizing system will be able to destroy such a fault by eventually stabilizing into a correct global configuration without restarting the system. A local stabilizing graph relabeling system is a triple $\Re = (L, \mathcal{P}, \mathcal{F})$ where L is a set of labels, $\mathcal{P}$ a finite set of relabeling rules and $\mathcal{F}$ is a set of illegitimate local configurations [6]. Let $\mathcal{G}_{\mathcal{L}}$ be the set of labeled graphs (G, λ) and $h : \mathcal{G}_{\mathcal{L}} \longrightarrow \mathbb{N}$ be an application associating to each labeled graph (G, λ), the number of its illegitimate configurations at this stage of the computation. Therefore, we denote by $h(G, \lambda, \mathcal{F})$ the current number of illegitimate configurations of the labeled graph (G, λ). A local stabilizing graph relabeling system must satisfy the two following properties:

- $Closure$: $\forall\ (G, \lambda) \in \mathcal{G}_{\mathcal{L}}$ such that $h(G, \lambda, \mathcal{F}) = 0$, $\forall\ (G, \lambda')\ /(G, \lambda) \xrightarrow[\Re]{*} (G, \lambda') : h(G, \lambda', \mathcal{F}) = 0$.
- $Convergence$: $\forall\ (G, \lambda) \in \mathcal{G}_{\mathcal{L}}, \exists$ an integer k, such that $(G, \lambda) \xrightarrow[\Re]{k} (G, \lambda')$ and $h(G, \lambda', \mathcal{F}) = 0$.

As for self-stabilizing algorithms, the closure property stipulates the correctness of the relabeling system. A computation beginning in a correct state remains correct until the terminal state. The convergence however provides the ability of the relabeling system to recover automatically within a finite time (finite sequence of relabeling steps).

In [6] we state the following result:

Theorem 3.1: if $\Re = (L, I, P, \mathcal{F})$ is a graph relabeling system with illegitimate configurations $\mathcal{F}$, then it can be transformed into an equivalent local stabilizing graph relabeling system $\Re_s = (L, P_s, \mathcal{F})$.

The set P_s is composed of set P and some correction rules to detect and eliminate each illegitimate configuration of $\mathcal{F}$. The correction rules have higher priority than the rules in P.

Definitions 3.2:

- A configuration (G, L') is reached from the configuration (G, L) during the execution of the algorithm $\mathcal{R}$ iff $\exists\, p \geq 0$ such that $L \xrightarrow[\mathcal{R},p]{} L'$,
- Let $\mathcal{CR}$ (resp. $\mathcal{CR}_s$) be the set of all configurations reached during the execution of $\mathcal{R}$ (resp. $\mathcal{R}_s$).
- An algorithm $\mathcal{R}$ stabilizes to another algorithm $\mathcal{R}_s$ if the following hold: In each relabeling chain induced by the execution of $\mathcal{R}$, there is a finite suffix after which all the configurations reached are in $\mathcal{CR}_s$.
- The stabilization time is the length of the relabeling sequence corresponding to this suffix. So a measure of the stabilization time complexity of a distributed algorithm, or of a graph relabeling system, will be the stabilization time of the longest suffix of the algorithm.

B. The Mazurkiewicz's Enumeration Algorithm

An enumeration algorithm on a graph G is a distributed algorithm such that the result of any computation is a labeling of the nodes that is a bijection from $V(G)$ to $1, 2, ..., |V(G)|$. First, we give a description of the initial enumeration algorithm [1]. Every node attempts to get its own name, which shall be an integer between 1 and $|V(G)|$. A node chooses a name and broadcasts it with its neighbor-hood (i.e. the list of the name of its neighbors) all over the network. If a node u discovers the existence of another node v with the same name, then it compares its local view, i.e. the labeled ball of center u, with the local view of its rival v. If the local view v is "Stronger", then u chooses another name. Each new name is broadcast with the local view again over the network. At the end of the computation it is not guaranteed that every node has a unique name, unless the graph is non ambiguous. However, all nodes with the same name will have the same local view.

The crucial property of the algorithm is based on a total order on local views such that the "Strength" of the local view of any node cannot decrease during the computation. To describe this local view we use the following notation: if v has degree d and its neighbors have names $n_1, n_2, ...n_d$ with $n_1 \geq .... \geq n_d$, then $LV(v)$, the local view, is the $d\text{-}tuple(n_1, n_2, ...n_d)$. Let $\mathcal{LV}$ be the set of such ordered tuples. The alphabetic order defines a total order $\preceq$ on $\mathcal{LV}$. The nodes v are labeled by triples of the form (n, LV, GV) representing during the computation :

- $n(v) \in \mathbb{N}$ is the name of the node v,
- $LV(v) \in \mathcal{LV}$ is the latest view of v,
- $GV(v) \subset \mathbb{N} \times \mathcal{LV}$ is the mailbox of v and contains all the information received at this step of the computation. We call this set the global view of the v.

We define the list $sub(LV, n, n')$: the copy of LV where any occurrence of n is replaced by n' if n exists or adds n to LV otherwise. Let $LV \in \mathcal{LV}$ and $(n, n') \in \mathbb{N}^2$, if $n < n'$ then $LV \prec sub(LV, n, n')$. The initial labels of each node are $(0, \phi, \phi)$. Each node v has labels: $(n(v), LV(v), GV(v))$ and the labels obtained after applying a rule are $(n'(v), LV'(v), GV'(v))$. Let v_0 be a node which is center of ball $B(v_0)$ and let $\{v_1, v_1 \cdots, v_d\}$ be the set of its neighbors. Let $(n(v_i), LV(v_i), GV(v_i))$ the triple associated to the node v_i with $0 \leq i \leq d$. We call the ball of the center v and we write $B(v)$ the set composed of v and its neighbors. Now we present Mazurkiewicz's enumeration algorithm:

Mazurkiewicz's Enumeration Algorithm

R1 : **Transmitting rule**

<u>*Precondition :*</u>

– $\exists\ v_i \in B(v_0), GV(v_i) \neq GV(v_0)$

<u>*Relabeling :*</u>

– $\forall\ v_i \in B(v_0), GV'(v_i) := \bigcup_{v_j \in B(v_0)} GV(v_j)$

R2 : **Renaming rule**

<u>*Precondition :*</u>

– $\forall\ v_i \in B(v_0), GV(v_i) = GV(v_0)$

– $n(v_0) = 0$ or $n(v_0) > 0$ and ($\exists\ LV_1 | (n(v_0), LV_1) \in GV(v_0)$ and $LV(v_0) \prec LV_1$)

<u>*Relabeling :*</u>

– $n'(v_0) := 1 + max\{n \mid (n, LV) \in GV(v_0)\}$

– $\forall\ v_i \in B(v_0) \backslash \{v_0\}, LV'(v_i) := sub(LV(v_i), n(v_0), n'(v_0))$

– $\forall\ v_i \in B(v_0), GV'(v_i) := GV(v_i) \bigcup_{v_j \in B(v_0)} \{(n'(v_j), LV'(v_j))\}$

C. A Local Stabilizing Enumeration Algorithm

We present in the sequel a new enumeration algorithm encoded by local stabilizing relabeling systems. This protocol is optimal compared to the version presented in [2]. We will refer to this protocol as $\mathcal{ENL}$ algorithm.

In a correct behavior, when a name of v_0 is already chosen by another node v_i, v_0 (resp. v_i) will receive this information and change its name if the local view of v_0 (resp. v_i) contains the older modifications (see Figure 1).

In a corrupted behavior, when a name of v_0 is corrupted, it detects this corruption, changes its name to -1 and initializes its states, then one of its neighbors v_i detects this change, corrects some of the state of v_0. After that, v_0 chooses another number to rename itself as shown in Figure 2.

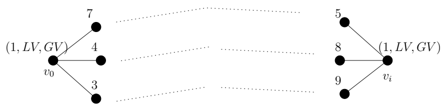


Figure 1. Example of a safe running

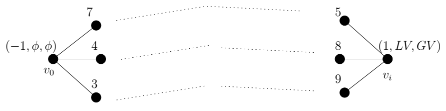


Figure 2. Example of a run starting from corrupted state

Function $\delta_L(n)$ gives the number of occurrences of a name n in the list LV. We start by defining some illegitimate configurations to construct a set $\mathcal{F}$, then we improve the system by adding the correction rules to detect and to eliminate these configurations. The node v_0 is said to be corrupted or in the illegitimate configuration, if one of its components is changed using extra relabeling. This relabeling does not correspond to those of the previous rules. We can define the following predicates to denote theses behaviors.

1) Corruption of the name: $(n(v_0), LV) \notin GV(v_0)$ or $n(v_0) > \mid V(G) \mid$.
2) Corruption of the local view:$\exists\ n_1 \in LV(v_0) \mid \neg\exists\ v_i \in B(v_0) \backslash \{v_0\} : n(v_i) = n_1$ or $n_1 > \mid V(G) \mid$.
3) Corruption of the global view:$\exists\ (n_1, LV_1) \in GV(v_0)$, $\delta_{LV_1}(n(v_0)) \geq 2$ or $n_1 > \mid V(G) \mid$.

The function $choose\_unused(GV)$ chooses one unused name in the set of global view $GV \subset \mathbb{N} \times \mathcal{LV}$. We use the list $subset(GV, (n, LV), (n', LV'))$ to denote the copy of GV where any occurrence of (n, LV) is replaced by (n', LV') if $(n, LV) \in GV$ or adds (n', LV') to GV. For the present system, we deal with the following set $\mathcal{F} = \{f_1, f_2, f_3\}$ encoding the previous behaviors' predicates. Therefore, the correction rules are:

Detection and Correction Rules

Rc1 : **Corruption of the name**

<u>*Precondition :*</u>

– $(n(v_0), LV) \notin GV(v_0)$ or $n(v_0) > \mid V(G) \mid$

<u>*Relabeling :*</u>

– $(n'(v_0), LV'(v_0), GV'(v_0)) := (-1, \phi, \phi)$

– $\forall\ v_i \in B(v_0) \backslash \{v_0\}, LV'(v_i) := LV(v_i) \backslash \{n(v_0)\}$

– $\forall\ v_i \in B(v_0), GV'(v_i) := GV(v_i) \backslash \{(n(v_0), LV(v_0))\}$

Rc2 : **Corruption of the local view**

<u>*Precondition :*</u>

– $\exists\ n_1 \in LV(v_0) \mid \neg\exists\ v_i \in B(v_0) \backslash \{v_0\} : n(v_i) = n_1$ or $n_1 > \mid V(G) \mid$

<u>*Relabeling :*</u>

– $LV'(v_0) := LV(V_0) \backslash \{n_1\}$

– $GV'(v_i) := subset(GV(v_i), (n(v_0), LV(v_0)), (n(v_0), LV'(v_0)))$

– $n'(v_i) = -2$

Rc3 : **Choose of the name**

<u>*Precondition :*</u>

– $n(v_0) = -2$

<u>*Relabeling :*</u>

– $n'(v_0) := choose\_unused(GV(v_0))$

– $\forall\ v_i \in B(v_0) \backslash \{v_0\}$, $LV'(v_i) := sub(LV(v_i), n(v_0), n'(v_0))$

– $\forall\ v_i \in B(v_0)$, $GV'(v_i) := GV(v_i) \bigcup_{v_j \in B(v_0)} \{(n'(v_j), LV'(v_j))\}$

Rc4 : **Corruption of the global view**

<u>*Precondition :*</u>

– $\exists\ (n_1, LV_1) \in GV(v_0)$, $\delta_{LV_1}(n(v_0)) \geq 2$ or $n_1 > \mid V(G) \mid$

<u>*Relabeling :*</u>

– $GV'(v_0) := GV(v_0) \backslash \{(n_1, LV_1)\}$

IV. Proof of Correctness and Complexity Analysis

Now, we show the correctness of the $\mathcal{ENL}$ algorithm using a scheme based on the properties satisfied by our protocol according to the starting configuration. First, we deal with a configuration without illegal configurations. Then, we prove that for a computation which starts from a configuration with corruptions, our protocol reaches a configuration without illegal configuration in a finite time. Finally, we show that the protocol satisfies its task in spite of corrupted data. The analysis is closed with time complexity measures.

A. Proof of Correctness

We define the relabeling system $\Re_s = (L, P_s, \mathcal{F})$, where $L = \{\{\mathcal{N} \cup \{0, -1, -2\}\} \times 2^{\mathcal{N}} \times 2^{\mathcal{N} \times 2^{\mathcal{N}}}\}$ and $P_s = \{R1, R2, Rc1, Rc2, Rc3, Rc4\}$ such that $Rc_j > R_i$. We now state the main results.

Lemma 4.1: The system $\Re_s = (L, P_s, \mathcal{F})$ satisfies the *closure* property.

Proof. We prove this Lemma by induction on the size of relabeling sequences. Let (G, λ) any labeled graph where $h(G, \lambda, \mathcal{F}) = 0$. Let (G, λ_k) a labeled graph obtained from (G, λ) by applying k rules only from the set $P = \{R1, R2\}$. From the definition of correction rules, they are applied when some illegitimate configurations are introduced. When $k = 0$ the Lemma is $true$. We suppose that the lemma remains $true$ after applying k rules. Now we show that the lemma remains true after the application of $k+1$ rules. From the induction hypothesis, $h(G, \lambda_k, \mathcal{F}) = 0$. At this step, the only possible application rules are $R1$ and $R2$. By definition, such rules do not introduce illegitimate configurations, then $h(G, \lambda_{k+1}, \mathcal{F}) = 0$. Therefore, all the labeled graphs (G, λ') obtained from (G, λ) verify the property. Formally, if $h(G, \lambda, \mathcal{F}) = 0$ then $\forall\ (G, \lambda')$, $(G, \lambda) \xrightarrow[\Re]{*} (G, \lambda') : h(G, \lambda', \mathcal{F}) = 0$.

□

The proof of termination presented in [1] is based on the fact that no state can occur twice in the same run of the protocol. In the self-stabilization context when a system is subject to corruption, this fact is not automatically satisfied. Recall that [1] proposed an extension of its algorithm to deal with any graph (including ambiguous graphs). Therefore, this extension may be used to treat the case of corruption [2]. Another way consists to treat locally the corruptions, then the corrections' actions are executed during the execution of the enumeration algorithm. Our solution satisfies the last way, its correction is shown in the following Lemma.

Lemma 4.2: The system $\Re_s = (L, P_s, \mathcal{F})$ satisfies the *convergence* property.

Proof. We study the case of each illegitimate configuration.

1) $\exists\ v_0 \in V(G)$ labeled such that:$(n(v_0), LV) \notin GV(v_0)$ or $n(v_0) >\mid V(G) \mid$. The corrupted node v_0 applies rule $RC1$ to change its name to -1, then $RC2$ is executed by one of its neighbors v_i, after that v_0 executes $RC3$ to choose an unused number to rename itself. Let v_i one of its neighbors. The new state of v_i, after the application of rule $RC2$, is such that the name (resp. the local view) of v_0 does not appear in the local view (resp. in the global view) of v_i. Then, the *Transmitting rule* allows to diffuse this novel state in all the graph.
2) $\exists\ v_0 \in V(G)$ labeled such that: $\exists\ n_1 \in LV(v_0) \mid \neg\exists\ v_i \in B(v_0)\backslash\{v_0\} : n(v_i) = n_1$ or $n_1 >\mid V(G) \mid$. Correction rule $RC2$ detects and eliminates such configuration.
3) $\exists\ v_0 \in V(G)$ labeled such as: $\exists\ (n_1, LV_1) \in GV(v_0)$, $\delta_{LV_1}(n(v_0)) \geq 2$ or $n_1 >\mid V(G) \mid$. Also, with the same reasoning, rule $RC4$ is applied to detect and eliminate this corrupted configuration.

The size of the relabeling sequence required, to eliminate all the illegitimate configurations and also to terminate the execution of the algorithm, is given in the section related to the complexity analysis. Formally, we have the following properties:

- The application of a correction rule decreases $h(G, \lambda, \mathcal{F})$ and induces the execution of the rule $R1$.
- The application of a rule in P does not increase $h(G, \lambda, \mathcal{F})$.
- $\forall\ (G, \lambda) \in \mathcal{G}_{\mathcal{L}}, \exists$ an integer k, $(G, \lambda) \xrightarrow[\Re]{k} (G, \lambda') : h(G, \lambda', \mathcal{F}) = 0$

□

Lemma 4.3: The system $\Re_s = (L, P_s, \mathcal{F})$ encodes an enumeration algorithm.

Proof. To show that the result is an enumeration, we use the same properties as those used in [1], [2]. Let G be a graph, to explain how this protocol denoted by $\mathcal{P}$ works, we introduce some notations. We denote by σ the state of the network which is composed of the local configurations of all the nodes. If $\sigma(v) = (n, LV, GV)$, then, n, LV, GV refer to the name, the local view and the mailbox of v at state σ. Thus, $\gamma(\sigma(v))$ is the name of the node v at state σ. Elements of mailbox are called messages; message m is sent by na if $m = (na, LV)$ for some LV. Name na is *known* to node v at state σ, if $\sigma(v) = (n, LV, GV)$ for some n, LV, GV and GV contains a message sent by na. We denote by $\sigma_k = (n_k(v), LV_k(v), GV_k(v))$ the label of each node $v \in V(G)$ after the k^{th} step of the computation of $\mathcal{P}$. Protocol $\mathcal{P}$ satisfies the following properties whose proofs can follow the scheme of the one used in [1].

(I1) $\forall\ k \geq 0,\ v \in V(G)$: $LV_k(v) = LV_k(B(v)\backslash\{v\}) - \{0, -1, -2\}$,
(I2) $\forall\ k \geq 0,\ v \in V(G)$: $u, w \in B(v)$: $n_k(u) = n_k(w) \Rightarrow u = w$,
(I3) $\forall\ k \geq 0, v \in V(G)$: $n_k(v) \leq n_{k+1}(v)$ and $LV_k(v) \preceq LV_{k+1}(v)$ and $GV_k(v) \subseteq GV_{k+1}(v)$,
(I4) $\forall\ k \geq 0,\ v \in V(G)$, $\forall\ (na, LV) \in GV_k(v)$: $\exists\ u \in V(G)$, $n_k(u) = na$,
(I5) $\forall\ k \geq 0, v \in V(G), n_k(v) \neq 0$: $\forall\ na, na', na \leq na'$, v known $na' \Rightarrow v$ known na,
(I6) Protocol $\mathcal{P}$ is terminating for any graph,
(I7) The result of any run of $\mathcal{P}$ for any graph G is locally bijective,
(I8) $\mathcal{P}$ is an enumeration protocol for any unambiguous graph.

□

From the proofs of the three previous lemmas, we can state:

Corollary 4.4: Starting from any labeled graphs, the system $\Re_s = (L, P_s, \mathcal{F})$ terminates.

Corollary 4.5: The relabeling system $\Re_s$ is local stabilizing. It encodes a self-stabilizing enumeration algorithm for any unambiguous graph.

B. Complexity Analysis

In this section, we consider the model of distributed system described in Section 2.B and we compute the complexity of our protocol in terms of relabeling rules or steps. The number of steps when the system does not contain any failure component is denoted by $\mathcal{M}$, the number of steps with failures is denoted by Σ. The complexity of the Mazurkiewicz enumeration algorithm is $\mathcal{M} = \theta(\mid V(G) \mid^3)$. The time of stabilization of the algorithm proposed in [2] is $\theta(t\times \mid V(G) \mid^2)$ steps, where t is the sum of the number of nodes and the highest name initially known. We use the following properties to give the stabilization time of our protocol:

1) Each node applies one rule in the set of correction rules to correct itself.
2) The application of the correction rules does not add illegitimate configurations. The application of the rule $RC1$ provokes the application of the rules $RC2$, $RC3$ and so $R1$.
3) In the worst case, for f_1 corruptions of names, f_2 corruptions of local view, f_3 corruptions of global view, the nodes apply $3f_1 + f_2 + f_3$ correction rules. Therefore, the stabilization time is $\theta(5\times \mid V(G) \mid^3)$, when rule $R1$ is applied $\mid V(G) \mid^2$ times.

The worst case corresponds to the case of f_1. Let $f = 3f_1$, then the time of stabilization is $\theta(9f \mid V(G) \mid^2)$. So, the time for the enumeration algorithm subject to f corruptions is $\Sigma = \theta(9f \mid V(G) \mid^2 + (\mid V(G) \mid -f)^3)$. Starting from a configuration without corruptions, we obtain $\theta(\mid V(G) \mid^3)$. See that in [2], the parameter t is unbounded, and in our version f is bounded by $\mid V(G) \mid$.

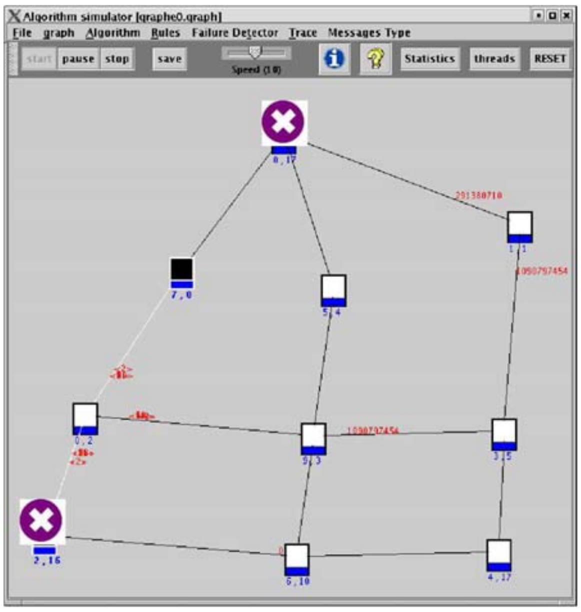
V. An Implementation on the Visidia Tool

Visidia [12] is a tool to implement, to simulate, to test and to visualize distributed algorithms. It is motivated by the important theoretical results on the use of graph relabeling systems to encode distributed algorithms and to prove their correctness. Visidia provides a library together with an easy interface to implement distributed algorithms described by means of local computations. The distributed system of Visidia is based on asynchronous message passing model. However, it has been assumed that components of such a system do not fail. The threads representing the processes of the computation are created on the same machine.

A stage of computation [14] in Visidia is carried out after some synchronization, which can be achieved by using probabilistic procedures [15]. The processes are simulated by Java threads. The high level primitives including the synchronization procedures allows the user to implement local computations.

There are three types of local computations. To implement these local computations in an asynchronous message passing system, a randomized synchronization procedure is associated to each type, which are given in the following:

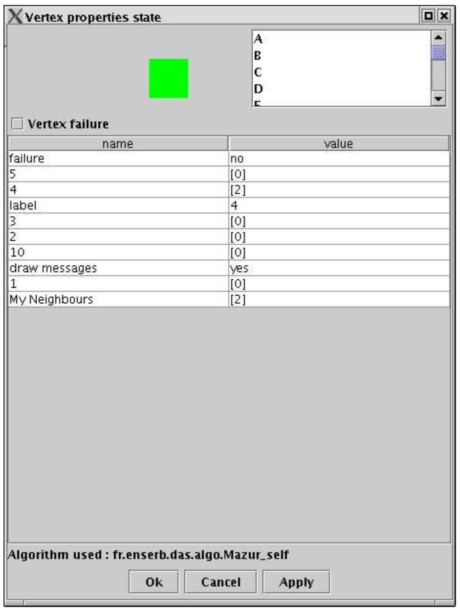
1) *Rendez-vous* (RV): in a computation step, the labels attached to nodes of K_2 (the complete graph with 2 nodes) are modified according to some rules depending on the labels appearing on K_2.
2) *Local Computation 1* (LC1): in a computation step, the label attached to the center of a ball is modified according to some rules depending on the labels of the ball, labels of the leaves are not modified.
3) *Local Computation 2* (LC2): in a computation step, the labels attached to the center and to the leaves of a ball may be modified according to some rules depending on the labels of the ball.



Algorithm simulator [graphe0.graph]
File graph Algorithm Rules Failure Detector Trace Messages Type
start pause stop save Speed (10) Statistics threads RESET
0,17
291380710
1,1
1050797454
7,0
5,4
0,2
9,3
1050797454
3,5
2,16
6,10
4,17

Figure 3. The beginning of the simulation with transient failures

To simulate transient failures [16], the user can simulate the faulty of a process with the graphical user interface before the beginning of the simulation or during the simulation. For the enumeration algorithm, we show an execution by starting the algorithm with a randomized value of the name (labels). In the following figures, we present an execution of a local self-stabilizing enumeration algorithm on Visidia starting with faulty values as shown in Figure 3. For a graph of 10 nodes, each node is represented by two labels $(number, name)$ where $number$ is a number used to indicate and distinguish the nodes on the graphical interface and $name$ is a value used by the enumeration algorithm. The maximum chosen value doesn't upper to 10, there are two nodes numbered 0 and 2 with incorrect names respectively 17 and 16. In Figure 4 we show the local view of the node 7, its


Vertex properties state

A
B
C
D

Vertex failure

name	value
failure	no
5	[0]
4	[2]
label	4
3	[0]
2	[0]
10	[0]
draw messages	yes
1	[0]
My Neighbours	[2]

Algorithm used : fr.enserb.das.algo.Mazur_self

Figure 4. The beginning of the simulation with transient failures

number is 4 and has a neighbor 8 named 2. In Figure 5, the faulty node 2 (resp. 0) correct himself to 8 (resp. to 7). Then, the local view of the node 7 contains the correct named neighbor 0 (see Figure 6). Finally, Figure 7 and Figure 8 show the end of the execution of the enumeration algorithm. In the first one, the graph is totally named and in the second one we show the final local view of the node 7.

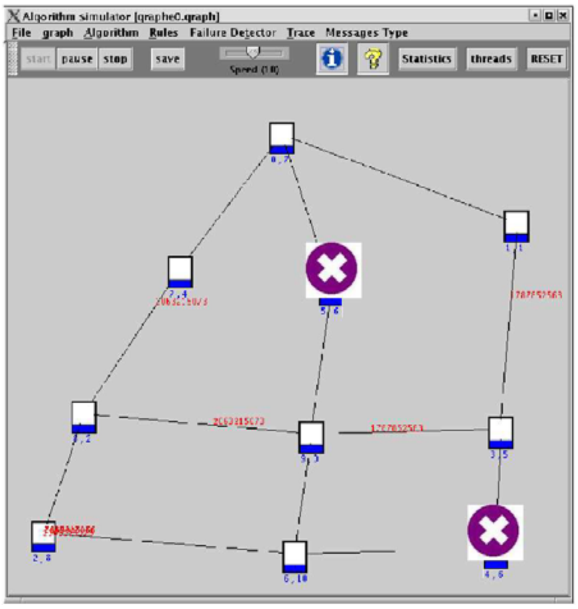
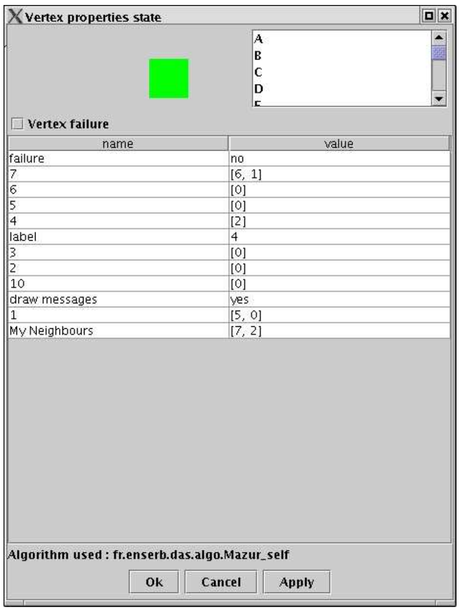

Algorithm simulator [graphe0.graph]
File graph Algorithm Rules Failure Detector Trace Messages Type
start pause stop save Speed (10) Statistics threads RESET

Figure 5. Local correction of the transient failures

VI. Message Passing Model

In this section we present a self-stabilizing enumeration protocol in message passing model. We use a model as presented in [17] and the extension presented in [18] to design self-stabilizing algorithms. This is based on the use of time-out mechanism to deal with the corruption of the local states. In the following we refer to this protocol as $\mathcal{ENM}$ algorithm. An important difference between the local computations model and message passing model is that in the last one, processor cannot individually checks whether the system is in an illegal local or global configuration. As well known that a self-stabilizing system is able to reach a desirable configuration starting from any arbitrary state. Starting from a configuration without any message in the channels, no processor may distinguish between legal terminal state and illegal state. So, the most


Vertex properties state

A
B
C
D

Vertex failure

name	value
failure	no
7	[6, 1]
6	[0]
5	[0]
4	[2]
label	4
3	[0]
2	[0]
10	[0]
draw messages	yes
1	[5, 0]
My Neighbours	[7, 2]

Algorithm used : fr.enserb.das.algo.Mazur_self

Figure 6. Local correction of the transient failures

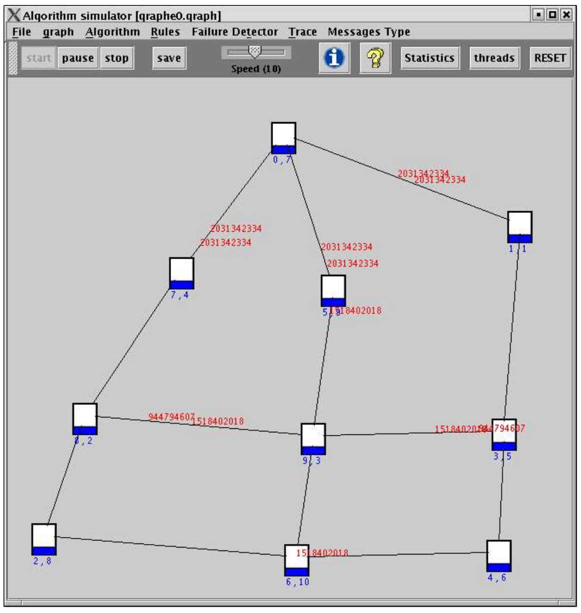
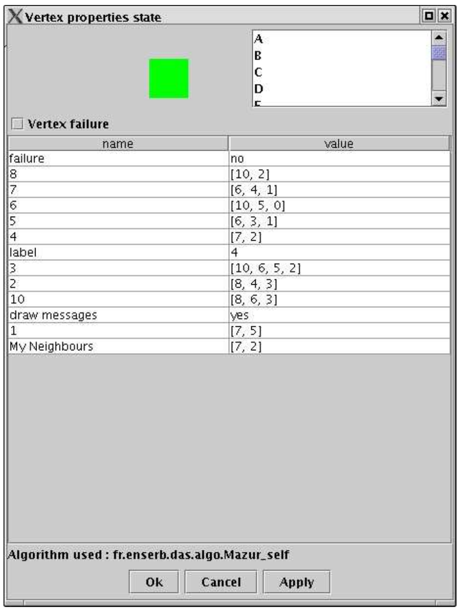

Algorithm simulator [graphe0.graph]
File graph Algorithm Rules Failure Detector Trace Messages Type
start pause stop save Speed (10) Statistics threads RESET
0 , 7
1 , 1
7 , 4
5 , 9
8 , 2
9 , 3
3 , 5
2 , 8
6 , 10
4 , 6

Figure 7. The end of enumeration and the local views



name	value
failure	no
8	[10, 2]
7	[6, 4, 1]
6	[10, 5, 0]
5	[6, 3, 1]
4	[7, 2]
label	4
3	[10, 6, 5, 2]
2	[8, 4, 3]
10	[8, 6, 3]
draw messages	yes
1	[7, 5]
My Neighbours	[7, 2]

Algorithm used : fr.enserb.das.algo.Mazur_self

Figure 8. The end of enumeration and the local views

works about self-stabilizing in a message passing model are not terminating. It means that, periodically processors exchange information to check the legality of the local then the global configuration or they assumed to receive information about the state of the neighbors infinitely often [19]. In this work we adopt the same concept of *silency* introduced in [19] as the ability of a system to converge to a global silent state. In such a state, processors still exchange messages to check and confirm the legality of this states and the the values of the local variables of processors are not altered. We can imagine unreliable failure detectors [20] to announce corruption of some process. Implementation of such a service is not explained in this work because it's another field of research.

A. The Protocol

The algorithm is based on those of [21] which is a simple translation of the Mazurkiewicz's algorithm in the message passing model. So, we use the same variables (name, local view, global view) to encode the states of processes. In the following we use the three type of messages:

1) $request$: asking for a local variables of neighbors,
2) $response$: responding to a request messages and containing the request information such as the value of the own name, local view and global view,
3) $new\_knowledge$: a message containing for each neighbor a novel affectation of the name and the update version of the local view of each of them. Such a message is transmitted to all the neighbors of the process that previously requested information of them.

For a better readability, we use the following procedures:

- We denote by $deg(v)$ the size of the set of outgoing port numbers of v.
- $renaming\_condition$: is the renaming condition as presented in the local computations model. When $name = 0$ or the received knowledge contains an element named as the node but with "stronger" set of known names.
- $all\_responded$: store the received knowledge to be used to renaming and to build a new knowledge. Also, it detects when all responses arrived from neighbors.
- $maxname()$: is the greater name known to neighbors,
- $new\_know(maxn)$: unifies the knowledge of all neighbors,
- $actualizes\_knowledge()$: it updates its current knowledge, including LV and GV, according to the received knowledge.

In the following, we assume that the hide of sent data (packet), which is the type of the message, is detected by the receiver at the reception of packet without added operation.

In a normal behavior, each node has the following variables: $name$, LV, GV as presented in the previous sections and $stage$ to denote the stage of the computation in which the processor is. So, we distinguish three stages: $\perp$, $waitR$ for wait response and $waitK$ for wait new knowledge. Starting from the normal configuration such that $\forall v$, $stage(v) = \perp$, $name(v) = 0$, $LV = \emptyset$, $GV = \emptyset$, each node broadcasts request to all its neighbors about their local variables and then waits for their responses (the $stage$ variable is set to $waitR$). At the reception of such a message at v_0 from a neighbor v_i, v_0 responses sending the current values of its local variables. If the stage of v_0 is $\perp$ then it will be changed to $waitK$. The reception of the responses allows to build knowledge about neighbors and then about all the nodes after the distribution of such knowledge. So, after the reception of a response from all the neighbors (the $stage$ of such a process is $waitR$), the processor checks its local name and those of the received knowledge. Thus, if the renaming condition is satisfied (as presented in the local computations model) it changes its name, then it updates its current knowledge according to the received knowledge. After that the new knowledge are transmitted to neighbors and changes its $stage$ to $\perp$. Finally, at the reception of knowledge at v_0 such that $stage(v_0) = waitK$, it modifies its local variables to take into account the received values and changes its $stage$ to $\perp$. At any step, processor set it stage to $terminate$ if its local variables contains a node named with a number which is the size of the graph to be enumerated.

Corruption of information causes abnormal configuration. Thus we deal with the same illegal configuration as described above. Therefore, using the same mechanism as [18], [22], we propose a self-stabilizing enumeration protocol in message passing model. The notion of *timeout* to check the apparition of abnormal configuration. Such a notion allows to destroy the termination caused when communication links are empty and the result of the computation is not correct.

As presented in the sections focused on the local computations model, we use the same predicates to denote the set of illegal configurations. We avoid some technical aspects to present correction rules associated to each corruption. The application of each correction rule generates exchange of message **corrupt** to unify knowledge of all the neighbors according to each kind of corruption.

Self-stabilizing Enumeration Algorithm in Message Passing Model

```
var $state \in \{\bot, waitR, waitK, terminate\}$;
    $name$ : integer;
    $LV$ : set of integer;
    $GV$ : set of knowledge;
    $i, p, q$ : integer;

I : {Init rule at each node $v_0$:}
    for ($i := 1$ to $deg(v_0)$) do send<reqR> via port $i$
    $stage := waitR$;

QR : {A message <reqR> has arrived at $v_0$ from port $q$}
    send<respR,name,LV,GV> via port $q$;
    $stage := waitK$;

RR : {A message <respR> has arrived at $v_0$ from port $q$}
    if ($all\_responded$ and $stage = waitR$)
        if ($name = 0$ or $renaming\_condition$)
            $maxn = maxname()$;
            $new\_know(maxn)$;
            $name := maxn + 1$;
            for ($i := 1$ to $deg(v_0)$) do send<newK,GV> via port $i$;
            $stage = \bot$;

RK : {A message <newK> has arrived at $v_0$ from port $q$}
    if ($stage = waitK$)
        $actualizes\_knowledge()$;
        $stage = \bot$;

CK : {for each node $v_0$, at the reach of the timeout:}
    if ($f_1$)
        $RC1()$;
    if ($f_2$)
        $RC2()$;
    if ($f_3$)
        $RC4()$;

RC1 : {A message <corrupt> has arrived at $v_0$ from port $q$}
    $actualizes\_knowledge\_bis()$;
    $stage = \bot$;
```

B. Proof of Correctness ans Analysis

Now, we show the correctness of the $\mathcal{ENM}$ algorithm using the same scheme as the proofs given in the local computations model. First, we prove that $\mathcal{ENM}$ algorithm is self-stabilizing. Then, we prove that the $\mathcal{ENM}$ satisfies its task. The proofs are based on the same techniques as such used in the local computations model. The analysis is closed with some complexity analysis.

Starting from a configuration without illegal configuration, the **CK** action doesn't change the states of any node. So, the *closure* property is satisfied. Starting from a configuration with illegal configuration, after the reach of some *time out*, node affected by the corruption detects such a configuration, applies adequate correction rule. The number of steps needed to correct each kind of corruption is the same as such used in the local computations model. Then, following each of the possible illegal configuration, the algorithm detects and corrects himself. Since the protocol, as described in [21], satisfies its task, we claim that,

Corollary 6.1: The $\mathcal{ENM}$ algorithm is local stabilizing. It encodes a self-stabilizing enumeration algorithm for any unambiguous graph.

Corollary 6.2: In a correct behavior, the cost of the $\mathcal{ENM}$ algorithm is in $O(| \ V \ |^3)$ messages. In the presence of f corruptions, the cost of the $\mathcal{ENM}$ is extended by the send of $c \times f$ **corrupt** messages, for some constant c. Thus, the algorithm is delayed according to the used *time out*.

VII. Conclusion

In this paper, we have presented a method to encode self-stabilizing enumeration algorithm with local computations. We have adapted the method given in [6] to create an easy self-stabilizing Mazurkiewicz's enumeration algorithm. This kind of algorithm can be used to implement a system which tolerates transient failures. The method is based on defining a set of illegitimate configurations and adding correction rules to the initial graph rewriting systems. The resulting protocol encoded by local computations is able to detect and correct transient failures by applying correction rules.

This protocol is easy to understand and its translation from the initial algorithm requires little changes. The proof is decomposed into two steps. First the proof of self-stabilization which is based on our developed framework. Second the proof that this protocol does its expected task which is based on the same as [1]. For the complexity study, we show that our protocol is better than [2]. In this work, we had also shown that self-stabilization meets global detection of termination such as [22].

The simulation phase allows us to prove and to show the convergence of our protocol in the presence of transient failures. We show this by starting the execution of the algorithm with faulty labels. Therefore, the system detects and corrects these transient failures by applying correction rules.

The enumeration protocol presented in this work in the message passing model allows to enforce the relation between local computations as high model and message passing model as a weaker model to deal with self-stabilization. This is an illustration of the use of intuitions of high model to encode easily protocols in weaker models as message passing model. Our work may be used on the study of solvable problems. Now we describe some of the key developments.

In the future, we start with measurements and analysis of the message passing version of the presented protocol. Then we will interested to study other algorithms including consensus, election using the same way as the enumeration to draw most exact border between possible and impossible tasks in distributed computing especially in unrelibale networks. So, both unreliability of processors and communication links will be considered. We are also interested to perform some exeperiemtal measurements as a pre-processing tasks to understand and to improve protocols behind theirs implementations. There are many applications that are based on the knowledge of a distinct identifier of each processor of a network. We will study the case of renaming problem as presented in [24], including crash failures, space of names and applications.

ACKNOWLEDGMENT

We thank anonymous reviewers of the DAIS06 confernce for their comments.

REFERENCES

[1] A. Mazurkiewicz, "Distributed enumeration," *Inf. Processing Letters*, vol. 61, no. 5, pp. 233–239, 1997.

[2] E. Godard, "A self-stabilizing enumeration algorithm," *Inf. Process. Lett.*, vol. 82, no. 6, pp. 299–305, 2002.

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

[4] M. Schneider, "Self-stabilization," *ACM Computing Surveys*, vol. 25, no. 1, pp. 45–67, 1993.

[5] E. Dijkstra, "Self stabilizing systems in spite of distributed control," *Communications of the ACM*, vol. 17, no. 11, pp. 643–644, 1974.

[6] B. Hamid and M. Mosbah, "An automatic approach to self-stabilization," in *6th ACIS International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing (SNPD2005), Baltimore, USA*. IEEE Computer Society, May 2005, pp. 123–128.

[7] A. Cournier, A. Datta, F. Petit, and V. Villain, "Self-stabilizing pif algorithms in arbitrary network," *21th International Conference on Distributed Computing Systems (ICDCS 2001)*, pp. 91–98, April 2001.

[8] S. Ghosh, A. Gupta, T. Herman, and S. V. Pemmaraju, "Fault-containing self-stabilizing algorithms," in *PODC '96: Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*. ACM Press, 1996, pp. 45–54.

[9] B. Awerbuch, B. Patt-Shamir, and G. Varghese, "Self-stabilization by local checking and correction (extended abstract)," in *Proceedings of the 32nd annual symposium on Foundations of computer science*. IEEE Computer Society Press, 1991, pp. 268–277.

[10] Y. Afek and S. Dolev, "Local stabilizer," in *ISTCS '97: Proceedings of the Fifth Israel Symposium on the Theory of Computing Systems (ISTCS '97)*. IEEE Computer Society, 1997, p. 74.

[11] Y. Afek, S. Kutten, and M. Yung, "The local detection paradigm and its applications to self-stabilization," *Theor. Comput. Sci.*, vol. 186, no. 1-2, pp. 199–229, 1997.

[12] M. Mosbah and A. Sellami, "Visidia: A tool for the VIsialization and SImulation of DIstributed Algorithms(2003)," http://www.labri.fr/visidia/.

[13] I. Litovsky, Y. M´etivier, and E. Sopena, "Graph relabeling systems and distributed algorithms," in *Handbook of graph grammars and computing by graph transformation*, W. S. Publishing, Ed., vol. Vol. III, Eds. H. Ehrig, H.J. Kreowski, U. Montanari and G. Rozenberg, 1999, pp. 1–56.

[14] M. Bauderon, Y. M´etivier, M. Mosbah, and A. Sellami, "Graph relabeling systems : A tool for encoding, proving, studying and visualizing distributed algorithms," *Electronic Notes in Theoretical Computer Science*, vol. 51, 2001.

[15] Y. M´etivier, N. Saheb, and A. Zemmari, "Randomized rendezvous," in *Colloquium on mathematics and computer science: algorithms, trees, combinatorics and probabilities*, ser. Trends in mathematics, Birkhauser, Ed., 2000, pp. 183–194.

[16] B. Hamid and M. Mosbah, "Visualization of self-stabilizing distributed algorithms," in *9th International conference information visualization IV 2005, London, UK*. IEEE Computer Society, 2005, pp. 550–555.

[17] G. Tel, *Introduction to distributed algorithms*, 2nd ed. Cambridge University Press, 2000.

[18] M. G. Gouda and N. Multari, "Stabilizing communication protocols," *IEEE Trans. Comput.*, vol. 40, no. 4, pp. 448–458, 1991.

[19] S. Dolev, M. G. Gouda, and M. Schneider, "Memory requirements for silent stabilization," in *PODC '96: Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*. New York, NY, USA: ACM Press, 1996, pp. 27–34.

[20] T. Chandra and S. Toueg, "Unreliable failure detectors for reliable distributed system," *Journal of the ACM*, vol. 43, no. 2, pp. 225–267, July 1996.

[21] P. Dembinski, "Enumeration protocol in estelle: an exercise in stepwise development," in *FORTE XI / PSTV XVIII '98: Proceedings of the FIP TC6 WG6.1 Joint International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols (FORTE XI) and Protocol Specification, Testing and Verification (PSTV XVIII)*. Deventer, The Netherlands, The Netherlands: Kluwer, B.V., 1998, pp. 147–162.

[22] A. Arora and M. Nesterenko, "Unifying stabilization and termination in message-passing systems," in *ICDCS '01: Proceedings of the The 21st International Conference on Distributed Computing Systems*. Washington, DC, USA: IEEE Computer Society, 2001, p. 99.

[23] K. Wada and W. Chen, "Optimal fault-tolerant routings with small routing tables for k-connected graphs." *J. Discrete Algorithms*, vol. 2, no. 4, pp. 517–530, 2004.

[24] H. Attiya, A. Bar-Noy, D. Dolev, D. Peleg, and R. Reischuk, "Renaming in an asynchronous environment," *J. ACM*, vol. 37, no. 3, pp. 524–548, 1990.

B. Hamid He received his Master degree in computer science from the University of Jules Vernes in 2003. He is currently an PhD Student at the University of Bordeaux-1, France. His current research interests include models and algorithms and simulation platform for distributed computing systems, failure detection and self-stabilization.

Dr. M. Mosbah Has received his PhD degree from the university of Bordeaux 1 (France) in 1993. He has been an associate professor between 1994 and 2002. He is a full professor in computer science since 2003 at ENSEIRB (National Graduate School of Electronics, Computer Science and Telecommunications of Bordeaux), France. His research interests include distributed algorithms, mobile agent algorithms, ad hoc networks and graph algorithms. He participated to several national and European research projects. Currently, he is the responsible of the graduate studies at the University of Bordeaux