

Snap-Stabilizing Tasks in Anonymous Networks

Emmanuel Godard

the date of receipt and acceptance should be inserted later

Abstract We consider snap-stabilizing algorithms in anonymous networks. Self-stabilizing algorithms are well known fault tolerant algorithms : a self-stabilizing algorithm will eventually recover from arbitrary transient faults. On the other hand, an algorithm is snap-stabilizing if it can withstand arbitrary initial values and immediately satisfy its safety requirement. It is a subset of self-stabilizing algorithms. Distributed tasks that are solvable with self-stabilizing algorithms in anonymous networks have already been characterized by Boldi and Vigna in [BV02b].

In this paper, we show how the more demanding snap-stabilizing algorithms can be handled with standard tools for (not stabilizing) algorithms in anonymous networks. We give a characterization of which tasks are solvable by snap-stabilizing algorithms in anonymous networks. We also present a snap-stabilizing version of Mazurkiewicz' enumeration algorithm.

This work exposes, from a task-equivalence point of view, the complete correspondence in anonymous networks between self or snap-stabilizing tasks and distributed tasks with various termination detection requirements.

1 Introduction

In the world of fault-tolerance, distributed tasks that admits self-stabilizing solutions have been long studied [Dol00]. An algorithm is self-stabilizing if, starting from arbitrary initial values in the registers used by the algorithm, it can eventually stabilize to a correct final value. In particular, when looking at some computed values, the algorithm can output incorrect values as long as it eventually outputs correct ones.

In contrast, an algorithm is snap-stabilizing if it can withstand arbitrary initial values and output only correct values [BDPV99]. Snap-stabilizing tasks

form a subset of self-stabilizing tasks where the algorithm is required to retain computed values until it is “sure” that they are correct. Snap-stabilizing algorithms have really interesting properties, they can withstand arbitrary transient failures, while at the same time, improving on self-stabilizing algorithms about a key point : the stabilization moment is not unknown : when a response is given, it is correct.

We present here the first characterization of snap-stabilizing tasks on anonymous networks. Not only we are reusing techniques borrowed from the study of the non-stabilizing tasks in anonymous networks and show they apply also here, but we complete the correspondence between self/snap-stabilizing tasks and termination detection.

How does snap-stabilizing tasks differ from self-stabilizing tasks has not been considered so far in anonymous networks to the best of our knowledge. Here we show that, on anonymous networks, there are tasks that admit self-stabilizing solutions but that have no snap-stabilizing ones. We show that the difference between self and snap stabilization is actually the same one gets with non-stabilizing tasks when considering implicit vs explicit termination. This result completes the understanding of the computability power of fault-tolerant and non fault-tolerant algorithms.

1.1 Our Result

We give the first characterization of the computability of snap-stabilization. In order to show that it complements known results about self-stabilizing and non self-stabilizing tasks in anonymous networks, we recall the previous equivalence established by Boldi and Vigna. Solving a task means solving a given specification linking inputs labels to output labels for a given set of graphs. Informally an algorithm has implicit termination if it is allowed to write numerous times a (tentative) solution in the dedicated OUT register. An algorithm has explicit termination when it is possible to write in OUT only once. Whenever the OUT register is defined, this means that (locally) the algorithm has terminated its computation.

Theorem 1 (Boldi and Vigna [BV01,BV02b]) *A task is solvable on a family of anonymous networks by a self-stabilizing algorithm if and only if it is solvable with implicit termination.*

The “only if” part being obvious, the merit of [BV02b] is to show that there is a universal algorithm to solve tasks (that are at all solvable) by a self-stabilizing algorithm on anonymous networks, and that the condition for solvability (informally speaking: stability of the specification by lifting) is exactly the one required by implicit termination. In other words, once a task is solvable with implicit termination, it admits a reliable self-stabilizing solution without any additional condition.

Theorem 2 (this paper) *A task is solvable on a family of anonymous networks by a snap-stabilizing algorithm if and only if it is solvable with explicit termination.*

As in the Boldi and Vigna result, the “only if” part is immediate. We therefore focus on establishing the “if” part. So the main contribution of this paper is a universal snap-stabilizing algorithm that solves the task at hand if this task satisfies the condition for being solvable by an algorithm with explicit termination.

This condition is given in Theorem 3. It is the same as the one given in [CGM08] for solvability with explicit termination. We first prove our results for terminating tasks in the asynchronous model, then we show how to extend the technique for long lived tasks in the synchronous model (for simplicity of exposition).

The roadmap is the following. Section 2 introduces the model of computation and the definition of snap-stabilizing algorithms. Section 3 introduces the algebraic tools that are necessary to express the condition in Theorem 3. Section 4 describes a universal snap-stabilizing algorithm based upon Mazurkiewicz enumeration algorithm [Maz97].

1.2 Related Work

Given a distributed task, the condition for it being solvable by an algorithm with explicit termination was first given in [BV01]. The presentation we will use in this paper is the one given in [CGM08]. Instead of the View algorithm of [YK96, BV01], we use Mazurkiewicz’ algorithm [Maz97]. A variation of Mazurkiewicz’ algorithm was proved to be self-stabilizing in [God02], in the Mazurkiewicz model, a model that offers strong synchronization between neighbours. We present here a version for the cellular model.

Snap-stabilizing algorithms were introduced in [BDPV99]. A more recent exposition can be found in [CDD⁺16]. In [CDV09, CDD⁺16], a general transformation technique is given to obtain simple snap-stabilizing algorithms from self-stabilizing ones. The authors expose a snap-stabilizing transformer for non-anonymous networks which implies that, in networks with identities, the tasks that are solvable by snap-stabilizing algorithms are exactly the ones that are solvable by self-stabilizing algorithms. In this paper, we prove the task equivalence between snap-stabilization and explicit termination in anonymous networks and show that this implies that the expressivity of snap-stabilizing algorithms is different of self-stabilizing algorithms in the anonymous context.

In [AD14], a probabilistic correction condition is proposed for snap-stabilizing algorithms. A Las Vegas algorithm is an algorithm whose termination is not guaranteed but whose outputs is always correct. The condition of [AD14] defines, in a sound way, what is a Las Vegas stabilizing algorithm that is robust to arbitrary corruption of the initial memory.

Anonymous networks are networks where nodes do not have a name that is unique. It has seen many works since the seminal work of Angluin [Ang80]. There have been two main universal algorithms proposed to solve problems in this setting. The first one has been proposed by Yamashita and

Kameda in [YK96]. Its universality has been extended by Boldi and Vigna in [BV01] (explicit termination) and [BV02b] (implicit termination). It computes the (possibly infinite) universal cover of the underlying graph. The second one computes a minimal base of the underlying graph. It was presented by Mazurkiewicz [Maz97] to solve enumeration. Its universality has been extended in [GM02a]. Its extension to numerous other models has been done by Chalopin in [Cha06], its application to the Election problem in the message passing model has been done in [CGM12]. Boldi and Vigna have also shown how to derive a minimal base (in a finite time) from the universal covering [BV02b]. One of the main advantage of Mazurkiewicz' algorithm is that it is always stabilizing, contrary to the View algorithm of [BV02b] where it is necessary to know or derive an estimate of the size to make it stabilizing. On the distributed computability side, the first complete characterization of tasks that admits self-stabilizing algorithms has been given in [BV02b]. Here, we use a mix of different techniques from the second approach, some of which were first introduced in [CGM08].

There is an unpublished version of Mazurkiewicz' algorithm in the communication model of this paper but without transient faults in [Cha06, chap. 4], where the model is coined the "cellular model".

2 Definitions and Notations

2.1 Basic Definition for Computability

A network is represented by a graph or digraph G where vertices corresponds to nodes and edges or arcs corresponds to (possibly asymmetric) communication links. The set of vertices is denoted by $V(G)$. We consider a fixed set of labels Λ . Labels are used to represent the local states of parts of the communication network.

So we consider labelled graphs in the general sense. Nodes can be labelled (internal state of the nodes), arcs can be labelled (messages in transit, port numbering). We will use \mathbf{G} to denote a (di)graph with all its associated labels. Since the input labels can be encoded in the labels, we consider all labelled graphs as the possible inputs for distributed algorithms. The set of all labelled graphs is denoted \mathcal{G} . Given a labelled graph $\mathbf{G} = (G, \lambda)$, where G is the underlying graph and $\lambda : V(G) \mapsto \Lambda$ is the labelling function, we will conveniently note (\mathbf{G}, λ') the graph G labelled by $\lambda \times \lambda'$.

Given a network $\mathbf{G} \in \mathcal{G}$ and a vertex v in \mathbf{G} , we assume that the state of a node v during the execution of any algorithm is of the form $(\lambda(v), \text{MEM}(v), \text{OUT}(v))$. This tuple of registers has the following semantics. $\lambda(v)$ is a read-only part of the state, $\text{MEM}(v)$ is the internal memory of v , $\text{OUT}(v)$ will contain the output value, i.e. the result of the computation at node v . When the register OUT is not defined, it contains the value \perp .

A distributed algorithm is an algorithm that is replicated on every node and operates on the local state of the node v by way of communication with

the neighbours of v . The communication here is done in the *locally shared variables* model of Dijkstra, that is also called the *cellular* model [Cha06]. A distributed algorithm is a set of rules (pairs of precondition and command) that describe how a node has to change its current state (the command) according to its own state and the state of *all its neighbors* (the precondition or guard). We say that a rule R is activable at a node v if the neighborhood of v satisfies the precondition of R . In this case, the vertex v is also said to be activable. If a rule R is activable in v , an atomic move for v consists of reading the states of all its neighbors, computing a new value of its state according to the command of R , and writing this value to the register MEM and/or OUT. If more than one rule is activable at a node, one is chosen non-deterministically. Of course, it is possible to have priorities for rules, and to discard this non-determinism.

A daemon is a distributed adversary that chooses at each step a set of activated nodes among the activable ones. If only one node can be chosen at a time, this is called the *central daemon*. If any set can occur, this is called the *asynchronous daemon*. If the sets of activated nodes is exactly the set of activable nodes this is called the *synchronous daemon*. Given a daemon, an execution, or run, is a sequence of atomic moves of activated nodes. We consider here the asynchronous daemon (whose executions contain the synchronous daemon execution).

A vertex-relabelling relation is a relation between labelled graphs where the underlying graphs are identical. The evolution of the global system can be seen as a sequence of relabelling steps where only the state part of the labels of the graphs is modified, according to the application of rules prescribed by the algorithm at a set of locations that depends of the kind of daemon that is considered. Under a given execution ρ , the evolution of the global configuration of the network \mathbf{G} is described by the sequence of labelled graphs $\mathbf{G}, (\mathbf{G}, \text{MEM}_1 \times \text{OUT}_1), (\mathbf{G}, \text{MEM}_2 \times \text{OUT}_2), \dots$; this is usually abbreviated to $\mathbf{G}_0, \mathbf{G}_1, \mathbf{G}_2, \dots$ for convenience.

If the sequence is finite, that is if there is a step $t \in \mathbb{N}$ where no rule is applicable, or if there is an infinite suffix starting from step $t \in \mathbb{N}$ where the registers OUT are not modified, we say that the execution has stabilized and denote by \mathbf{G}^f the graph labelled with OUT_t , $\mathbf{G}^f = (\mathbf{G}, \text{OUT}_t)$. It is the *terminal state* of the computation.

A terminating problem is a distributed problem for which it is expected that the nodes have final values. For example, the Election problem is a terminating problem that should be compared with the Mutual Exclusion problem where nodes have to solve indefinitely the problem of entering the critical section one node at a time. We formally define now what is a terminating distributed problem.

Definition 1 A *terminating task* is a couple (S, \mathcal{F}) where $\mathcal{F} \subset \mathcal{G}$ is a family of labelled graphs and S is a vertex-relabelling relation on \mathcal{G} .

The *specification* S is a general way to describe our distributed problem in terms of relation between inputs and outputs. This description is independent of the *domain* \mathcal{F} where we want to solve our problem.

For example, the well-known Election problem is specified by S_{LE} such that $\mathbf{G}S_{LE}\mathbf{G}'$ if $\mathbf{G}' = (\mathbf{G}, \lambda')$ has only one node labelled by the special label ELECTED. The Size problem where the algorithm has to compute the number of nodes of the network is specified by S_{size} such that $\mathbf{G}S_{size}(\mathbf{G}, |V(\mathbf{G})|)$.

Definition 2 Given a terminating task (S, \mathcal{F}) , an algorithm ALGO solves S on $\mathbf{G} \in \mathcal{F}$ if for any execution $(\mathbf{G}_0, \mathbf{G}_1, \mathbf{G}_2, \dots)$ with $\mathbf{G}_0 = \mathbf{G}$:

decision $\forall v \in V(\mathbf{G}), \text{OUT}(v)$ is written exactly once by v ;
 stabilization the execution stabilizes and the terminal state is denoted \mathbf{G}^f ;
 correction $\mathbf{G}S\mathbf{G}^f$.

Definition 3 The terminating task (S, \mathcal{F}) is solvable if there exists an algorithm ALGO such that ALGO solves S for all $\mathbf{G} \in \mathcal{F}$.

When the stabilization is obtained with only finite executions, we say the algorithm is silent. When, besides **correction**, the **stabilization** property is the only property, we talk about *implicit termination* (or message termination [Tel00]). When we have both **stabilization** and **decision**, we talk about *explicit termination* (or process termination [Tel00]). In the context of this paper solvability is meant in the explicit termination setting. Implicit termination is weaker than explicit termination, and for obvious reason, it is the termination for self-stabilizing algorithms. Note that, in a distant area of Distributed Computing, this is also the termination type of *failure detectors* [CT96]. Those are the two main termination mode that are classically considered in distributed algorithms. See also [CGM08, GMT10] for other types of termination.

2.2 Self- and Snap-Stabilization

Informally, a distributed algorithm is said to be self-stabilizing if an execution starting from any arbitrary global state has a suffix belonging to the set of legitimate states. Note that when we consider the terminating task (S, \mathcal{F}) , the set of legitimate states corresponds simply to the set of S -admissible outputs for the given input graph, that is the set $\{(\mathbf{G}, \text{MEM}, \text{OUT}) \in \mathcal{G} \mid \mathbf{G} \in \mathcal{F}, \mathbf{G}S(\mathbf{G}, \text{OUT})\}$. So, in the context of terminating tasks, this corresponds to the definition of solvability with implicit termination if we require the domain \mathcal{F} to be closed by arbitrary corruption of the initial memory.

More formally, it is possible to define self-stabilization in the framework of the previous section. Given a family \mathcal{F} , we define $\overline{\mathcal{F}} = \{(\mathbf{G}, \text{mem}) \mid \mathbf{G} \in \mathcal{F}, \text{mem} : V(\mathbf{G}) \rightarrow \Lambda\}$. The terminating task (S, \mathcal{F}) is solvable with self-stabilization if $(S, \overline{\mathcal{F}})$ is solvable with implicit termination.

Here we focus on snap-stabilization and give only a formal definition for snap-stabilization. Snap-stabilizing algorithms were introduced in [BDPV99]. A more recent exposition can be found in [CDD⁺16]. A snap-stabilizing algorithm computes tasks that are initiated by “requests” at some nodes of the network. A request is a special event. This event is an event exterior to the

algorithm and occurs *after* the end of the faults that led to arbitrary incorrect values. Given that the initial memory can be arbitrarily corrupted, the safety requirement of the problem specification has to have a special form that takes into account the fact that starting nodes have seen a request, see [CDD⁺16]. In order to have a unified framework, we chose in our equivalent presentation, to accept any specification S but to “implement” the special form in the definition, independently of the specific specification.

So since the initial memory can be arbitrarily corrupted, the correction of the OUT register is only required to be satisfied by nodes that have been causally influenced by the initial requests, i.e. nodes for which there exists a sequence of atomic moves that follow a path originating from a node where a request has been made. In other words, a distributed algorithm is snap-stabilizing if an execution starting from any arbitrary global state has *all its causal suffixes* belonging to the set of legitimate states.

Given a specific daemon and an algorithm, the system evolves according to the daemon and the algorithm: at one step, some nodes are activable and activated (their actions are processed). Given an execution ρ on \mathbf{G} , that is a sequence $\mathbf{G}_0, \mathbf{G}_1, \mathbf{G}_2 \dots$ of relabelling of \mathbf{G} where $\mathbf{G}_0 = \mathbf{G}$, we denote A_1, A_2, \dots the sequence of activated nodes. We have that \mathbf{G}_i is obtained by applying to \mathbf{G}_{i-1} the actions for the nodes of A_i .

We proceed to the formal definition. One or more external actions, *the requests*, are applied at some nodes $U \subset V(\mathbf{G})$. At time t , a node v is *causally influenced* by U if there exists a path u_0, u_1, \dots, u_k such that $u_0 \in U$, $u_k = v$, and there exists a strictly increasing function $\sigma : \mathbb{N} \rightarrow \mathbb{N} \forall i \geq 1, u_i \in A_{\sigma(i)}$, and $\sigma(k) \leq t$.

Definition 4 Given a terminating task (S, \mathcal{F}) , an algorithm ALGO is snap-stabilizing to S on $\mathbf{G} \in \mathcal{F}$ if for any request applied to $U \subset V(\mathbf{G})$,

causal decision $\forall v \in V(\mathbf{G})$, $\text{OUT}(v)$ is written exactly once after v has been causally influenced by U ;
 stabilization the execution stabilizes and the terminal state is denoted \mathbf{G}^f ;
 correction $\mathbf{G}S\mathbf{G}^f$.

Definition 5 The terminating task (S, \mathcal{F}) is solvable by snap-stabilization if there exists an algorithm ALGO such that ALGO is snap-stabilizing to S for all $\mathbf{G} \in \overline{\mathcal{F}}$.

For the sake of simplicity, in the following we always assume that $\overline{\mathcal{F}} = \mathcal{F}$.

2.3 Examples

To illustrate the various definitions we present in Fig. 1 an Election algorithm inspired by the well-known Le Lann Chang-Roberts algorithm [LeL77, CR79]. We will show that it is (non-silently) self-stabilizing to the Election task on unidirectional rings, but that it is not snap-stabilizing.

LCR1 : InitiateGuard :

- $\min(v_0) < \min(v)$,
- $\min(v_0) < id(v_0)$,

Action :

- $\min(v_0) := id(v_0)$,
- $tll(v_0) := N$

LCR2 : CirculateGuard :

- $\min(v_0) > \min(v)$,

Action :

- $\min(v_0) := \min(v)$
- $tll(v_0) := tll(v) - 1$

LCR3 : CleaningGuard :

- $\min(v_0) \neq \min(v)$ or $tll(v_0) \neq tll(v) - 1$
- $tll(v_0) \neq N$

Action :

- $\min(v_0) := id(v_0)$,
- $tll(v_0) := N$

LCR4 : ElectionGuard :

- $id(v_0) = \min(v)$,
- $tll(v) = 1$,

Action :

- $OUT(v_0) = \text{ELECTED}$
- $tll(v_0) = 0$,

Fig. 1 A LCR Election algorithm. The center of the cell is denoted v_0 , v is $pred(v_0)$.

We consider a unidirectional ring of known size N . The predecessor of a node v is denoted $pred(v)$. Each node v is equipped with a unique identity denoted $id(v)$. The algorithm maintains two variables \min and tll .

By considering the sequences of consecutive nodes, it is immediate to see that the labels are stable if and only if the sequence starts from a local minimum and the variables follow the semantic of the propagation of this local minimum according to the original LCR algorithm. This algorithm is therefore self-stabilizing but not snap-stabilizing even when adding a special Initiate rule to deal with the requests as below.

snapLCR1 : InitiateGuard :

- $Request(v_0)$

Action :

- $\min(v_0) = id(v_0)$,
- $tll(v_0) = N$

Indeed any node corrupted in such a way that the Election rule is immediately applicable will incorrectly set its output value to ELECTED if its predecessor is requested.

3 Computability of Terminating Tasks

We start by considering snap-stabilizing terminating tasks. We show how the general techniques from explicitly terminating non-stabilizing tasks can be extended to the snap-stabilizing case as well.

3.1 Digraphs and Fibrations

3.1.1 Definitions

In the following, we give the definitions for the tools introduced by Boldi and Vigna, and extensively studied in [BV02a], to characterize self-stabilizing tasks in [BV02b]. To introduce the main tool, that is *fibrations*, we need to consider directed graphs (or digraphs) with multiple arcs and self-loops. A *digraph* $D = (V(D), A(D))$ is defined by a set $V(D)$ of vertices and a set $A(D) \subset V(D) \times V(D)$ of arcs. Given an arc a , we denote $s(a)$ and $t(a)$, the source and target of the arc. An undirected graph G corresponds to the digraph $Dir(G)$ obtained by replacing all edges of G by the two corresponding arcs. In the following, we will not distinguish G and $Dir(G)$ when the context permits. The family of all digraphs with multiple arcs and self-loops is denoted \mathcal{D} . Note that the simple symmetric graphs of \mathcal{G} have direct counterparts in \mathcal{D} via Dir .

A dipath π of length p from u to v in D is a sequence of arcs a_1, a_2, \dots, a_p such that $s(a_1) = u, t(a_p) = v$ and for all i , $s(a_{i+1}) = t(a_i)$. A digraph is strongly connected if there is a path between all pairs of vertices. We assume all digraphs to be strongly connected.

Labelled digraphs will be designated by bold letters like **D**, **G**, **H** ...

A *homomorphism* γ between the digraphs D and D' is a mapping $\gamma : V(D) \cup A(D) \rightarrow V(D') \cup A(D')$ such that the image of a vertex is a vertex, the image of an arc is an arc and for each arc $a \in A(D)$, $\gamma(s(a)) = s(\gamma(a))$ and $\gamma(t(a)) = t(\gamma(a))$. A homomorphism $\gamma : V(D) \cup A(D) \rightarrow V(D') \cup A(D')$ is an *isomorphism* if γ is bijective.

As previously we consider labelled graphs and digraphs. We extend the definition of homomorphisms to labelled digraphs by adding the condition they also preserve the labelling ($\lambda(v) = \lambda(\gamma(v))$ for any vertex v).

In a digraph **G**, given $v_0 \in V(\mathbf{G})$ and $r \in \mathbb{N}$, we denote by $B_{\mathbf{G}}^-(v_0, r)$ the in-ball of center v_0 and radius r , that is the set of vertices v and arcs a such that there is a dipath of length at most r from v to v_0 .

3.1.2 Fibrations and Quasi-Fibrations

The notions of fibrations and quasi-fibrations enable to describe exactly the “similarity” between two anonymous networks that yields “similar” execution for any algorithm in the model of this paper. For the model of Angluin (used by Mazurkiewicz), the notions of coverings and quasi-coverings are the graph morphisms to be used, see eg. [GM02b].

A digraph \mathbf{D}' is a fibration of a digraph \mathbf{D} via ϕ if ϕ is a homomorphism from \mathbf{D}' to \mathbf{D} such that for each arc $a \in A(\mathbf{D})$ and for each vertex $v \in \phi^{-1}(t(a))$ (resp. $v \in \phi^{-1}(s(a))$), there exists a unique arc $a' \in \phi^{-1}(a)$ such that $t(a') = v$ (resp. $s(a') = v$).

The following lemma shows the importance of fibrations when we deal with anonymous networks. This is the counterpart of the lifting lemma that Angluin gives for coverings of simple graphs [Ang80] and the proof can be found in [BCG⁺96, BV02b, CM07].

Lemma 1 (Lifting Lemma [BCG⁺96]) *If \mathbf{D}' is a fibration of \mathbf{D} via ϕ , then for any daemon, any execution ρ of an algorithm ALGO on \mathbf{D} can be lifted up to an execution ρ' of ALGO on \mathbf{D}' , such that at any step, for all $v \in V(\mathbf{D}')$, $(\text{MEM}(v), \text{OUT}(v)) = (\text{MEM}(\phi(v)), \text{OUT}(\phi(v)))$.*

In particular, when the execution ρ has stabilized, the execution ρ' has also stabilized and the computed values are the same for v and $\phi(v)$.

In the following, one also needs to express similarity between two digraphs up to a certain distance. The notion of quasi-coverings was introduced as a formal tool in [MMW97, GM02a] for this purpose in the Mazurkiewicz model. The next definition is an adaptation of this tool to fibrations.

Definition 6 Given digraphs \mathbf{K} and \mathbf{H} , and integer r and $v \in V(\mathbf{K})$ and an homomorphism γ from $B_{\mathbf{K}}^-(v, r)$ to \mathbf{H} , \mathbf{K} is a quasi-fibration of \mathbf{H} of center v and radius r via γ if there exists a finite or infinite digraph \mathbf{G} such that \mathbf{G} is a fibration of \mathbf{H} via a homomorphism ϕ and there exists $w \in V(\mathbf{G})$ and an isomorphism δ from $B_{\mathbf{K}}^-(v, r)$ to $B_{\mathbf{G}}^-(w, r)$ such that for any $x \in V(B_{\mathbf{K}}^-(v, r)) \cup A(B_{\mathbf{K}}^-(v, r))$, $\gamma(x) = \phi(\delta(x))$

If a digraph \mathbf{G} is a fibration of \mathbf{H} , then for any $v \in V(\mathbf{G})$ and for any $r \in \mathbb{N}$, \mathbf{G} is a quasi-fibration of \mathbf{H} , of center v and of radius r . Conversely, if \mathbf{K} is a quasi-fibration of \mathbf{H} of radius r strictly greater than the diameter of \mathbf{K} , then \mathbf{K} is a fibration of \mathbf{H} . The following lemma is the counterpart of the lifting lemma for quasi-fibrations.

Lemma 2 (Quasi-Lifting Lemma, [CGM08, CGM12]) *Consider a digraph \mathbf{K} that is a quasi-fibration of \mathbf{H} of center v and of radius r via γ . For any algorithm ALGO, any execution ρ of ALGO on \mathbf{H} can be lifted up to an execution ρ' of ALGO on \mathbf{K} , such that at any step $t \leq r$, for all $v \in V(\mathbf{K})$, $(\text{MEM}(v), \text{OUT}(v)) = (\text{MEM}(\phi(v)), \text{OUT}(\phi(v)))$.*

In particular, when the execution ρ has stabilized in less than r steps, the execution ρ' has also stabilized and the computed values are the same for v and $\phi(v)$.

3.2 Main Result

In this section we state our main result in Theorem 3. By comparing its statement to that of [CGM08] we obtain Theorem 2. It is obvious that the impossibility result of [CGM08] applies here, as well as its proof. We present the impossibility result integrally here to make the paper self-contained.

We recall some technical notations and definitions from [CGM08]. We denote \mathcal{D}_\bullet the set $\{(\mathbf{G}, v) \mid \mathbf{G} \in \mathcal{D}, v \in V(\mathbf{G})\}$. Given a family $\mathcal{F} \subset \mathcal{G}$, we denote by \mathcal{F}_\bullet the set $\{(\mathbf{G}, v) \mid \mathbf{G} \in \mathcal{F}, v \in V(\mathbf{G})\}$. A function $f : \mathcal{D} \rightarrow \Lambda \cup \{\perp\}$ is an *output function* for a task (S, \mathcal{F}) if for each network $\mathbf{G} \in \mathcal{F}$ the labelling obtained by applying f on each node $v \in V(\mathbf{G})$ satisfies the specification S . That is $\mathbf{G}S(\mathbf{G}, \lambda)$ where $\forall v \in V(\mathbf{G}), \lambda(v) = f(\mathbf{G}, v)$.

In order to give our characterization, we need to formalize the following idea. When the in-ball at distance k of two processes v_1, v_2 in two digraphs $\mathbf{D}_1, \mathbf{D}_2$ cannot be distinguished (this is captured by the notion of quasi-fibrations and Lemma 2), and v_1 computes its final value in k rounds, then v_2 computes the same final value.

Definition 7 Given a function $r : \mathcal{D}_\bullet \rightarrow \mathbb{N} \cup \{\infty\}$ and a function $f : \mathcal{D}_\bullet \rightarrow \Lambda \cup \{\perp\}$, the function f is r -lifting closed if for all $\mathbf{K}, \mathbf{H} \in \mathcal{D}$ such that \mathbf{K} is a quasi-fibration of \mathbf{H} , of center $v \in V(\mathbf{K})$ and of radius $k \in \mathbb{N}$ via the homomorphism γ , if $k \geq \min\{r(\mathbf{K}, v), r(\mathbf{H}, \gamma(v))\}$, then $f(\mathbf{K}, v) = f(\mathbf{H}, \gamma(v))$.

Intuitively, a function f is r -lifting closed if $f(\mathbf{G}, v)$ depends only of $B_{\mathbf{G}}^-(v, r(\mathbf{G}, v))$, and it is undefined if $r(\mathbf{G}, v) = \infty$.

We give now the characterization of terminating snap-stabilizing tasks. We give the proof of the necessary condition. The converse will be proved in the following section, by describing a snap-stabilizing version of Mazurkiewicz' algorithm.

Theorem 3 A terminating task (S, \mathcal{F}) is solvable by snap-stabilization if and only if there exists a function $r : \mathcal{D}_\bullet \rightarrow \mathbb{N} \cup \{\infty\}$ and an output function $f : \mathcal{D}_\bullet \rightarrow \Lambda \cup \{\perp\}$ for (S, \mathcal{F}) such that,

- 3.i for all $(\mathbf{G}, v) \in \mathcal{D}_\bullet$, $r(\mathbf{G}, v) \neq \infty$ if and only if $f(\mathbf{G}, v) \neq \perp$;
- 3.ii f and r are r -lifting-closed;

Proof (of the necessary condition) Consider ALGO a distributed algorithm that snap-stabilizes to S on \mathcal{F} in t rounds.

We construct r and f by considering a subset of the possible executions of ALGO. We consider the synchronous execution of ALGO on any digraph $\mathbf{G} \in \mathcal{D}$. For any $v \in V(\mathbf{G})$, if $\text{OUT}(v) = \perp$ during the whole execution, then we set $f(\mathbf{G}, v) = \perp$ and $r(\mathbf{G}, v) = \infty$. This is possible since it could be that $\mathcal{F} \subsetneq \mathcal{D}$ and ALGO might be not terminating on graphs not in \mathcal{F} . Let r_v be the first causal step after which $\text{OUT}(v) \neq \perp$; in this case, if $r_v \leq t$, we set $f(\mathbf{G}, v) = \text{OUT}(v)$ and $r(\mathbf{G}, v) = r_v$. If $t < r_v$, then we set $f(\mathbf{G}, v) = \perp$ and $r(\mathbf{G}, v) = \infty$. By construction, 3.i is satisfied.

We also show that f is an output function and that f and r satisfy 3.ii. Consider two digraphs \mathbf{K} and \mathbf{H} such that \mathbf{K} is a quasi-fibration of \mathbf{H} , of center $v_0 \in V(\mathbf{K})$ and of radius k via γ with $k \geq r_0 = \min\{r(\mathbf{K}, v_0), r(\mathbf{H}, \gamma(v_0))\}$. If $r_0 = \infty$, then $r(\mathbf{K}, v_0) = r(\mathbf{H}, \gamma(v_0)) = \infty$ and $f(\mathbf{K}, v_0) = f(\mathbf{H}, \gamma(v_0)) = \perp$.

Otherwise, from Lemma 2, we know that after r_0 rounds, $\text{OUT}(v_0) = \text{OUT}(\gamma(v_0))$. Thus $r_0 = r(\mathbf{K}, v_0) = r(\mathbf{H}, \gamma(v_0))$ and $f(\mathbf{K}, v_0) = f(\mathbf{H}, \gamma(v_0))$. Consequently, f and r are r -lifting closed.

□

The previous proof shows that the output function f can be seen as corresponding to the final values obtained from the deterministic execution of an algorithm solving (S, \mathcal{F}) under the synchronous daemon. The value of $r(\mathbf{G}, v)$ can be understood as the number of steps needed by v to compute its final value in \mathbf{G} .

4 Main Algorithm

In this section, in order to obtain our sufficient condition, we present a general algorithm $\mathcal{M}_{f,r}$ in Figure 2 for which we use parameters that depend on functions f and r corresponding, via Theorem 3, to the terminating task (S, \mathcal{F}) we are interested in solving. This algorithm is a combination of a snap-stabilizing enumeration algorithm, adapted from [God02] and a generalization of an algorithm of Szymanski, Shy and Prywes (the SSP algorithm for short) [SSP85].

The algorithm in [God02] is described in a different model, where each computation step involves some strong synchronization between adjacent processes. It is a self-stabilizing adaptation of an enumeration algorithm presented by Mazurkiewicz in [Maz88]. The SSP algorithm enables to detect the global termination of an algorithm provided the processes know a bound on the diameter of the graph. The Mazurkiewicz-like algorithm always stabilizes on any network \mathbf{G} and during its execution, each process v can compute an integer $n(v)$ and reconstruct at some computation step i a digraph $\mathbf{G}_i(v)$ such that \mathbf{G} is a quasi-fibration of $\mathbf{G}_i(v)$ and the image of v is $n(v)$.

By applying the output function f on $\mathbf{G}_i(v)$ for $n(v)$, v can compute its OUT value. However, the enumeration algorithm does not enable v to compute effectively the radius of this quasi-fibration. We use a generalization of the SSP algorithm to compute a counter that is a lower bound on this radius, as it has already been done in Mazurkiewicz' model [GMT10] and in the message passing model [CGM08]. When the SSP counter is greater than $r(\mathbf{G}_i(v), n(v))$, the condition on f and r from Theorem 3 implies that the OUT value at v is correctly computed for S .

4.1 Modifying Mazurkiewicz' Enumeration Algorithm

An enumeration algorithm on a network \mathbf{G} is a distributed algorithm such that the OUT value are integers and the result of any computation is a labelling of the vertices that is a bijection from $V(\mathbf{G})$ to $\{1, 2, \dots, |V(\mathbf{G})|\}$. In particular, an enumeration of the vertices where vertices know whether the algorithm has terminated solves the Election Problem. Since Election is not solvable in all networks, it is not possible to solve the Enumeration problem on all networks. However, even if not solving Enumeration, in any network \mathbf{G} , the Enumeration algorithm of Mazurkiewicz always stabilizes and yields a digraph $\mathbf{G}_i(v)$ such that \mathbf{G} is a quasi-fibration of $\mathbf{G}_i(v)$.

We give first a general description of the Mazurkiewicz algorithm. Every vertex attempts to get its own name in \mathbb{N}^1 . A vertex chooses a name and broadcasts it together with the name of its adjacent vertices all over the network. If a vertex u discovers the existence of another vertex v with the same name, then it compares its *local view*, i.e., the labelled in-ball of center u and radius 1, with the *local view* of its rival v . If the local view of v is "stronger", then u chooses another name. Node u also chooses another name if it appears twice in the view of some other vertex as a result of a corrupted initial state. Each new name is broadcast 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 fibration minimal. However, all nodes with the same name will have the same local view, i.e., isomorphic labelled neighborhoods.

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 vertex cannot decrease during the computation. To describe the local view we use the following notation: if v has degree d and its in-neighbors have names n_1, \dots, n_d , with $n_1 > \dots > n_d$, then $\overline{N}(v)$, the local view, is the d -tuple (n_1, \dots, n_d) . Let T be the set of such ordered tuples. The lexicographic order defines a total order, $<$, on T . Vertices v are labelled by triples of the form (n, \overline{N}, M) representing during the computation:

- $n(v) \in \mathbb{N}$ is the name of the vertex v ,
- $\overline{N}(v) \in T$ is the latest view of v ,
- $M(v) \subset \mathbb{N} \times T$ is the mailbox of v and contains all information received at this step of the computation.

We introduce other notations. We want to count the number of times a given name appear in a local view. For a local view \overline{N} , and $n \in \mathbb{N}$, we define $\delta_{\overline{N}}(n)$ to be the cardinality of n in the tuple \overline{N} . For a given view \overline{N} , we denote by $sub(\overline{N}, n, n')$ the copy of \overline{N} where any occurrence of n is replaced by n' .

The complete algorithm is given in Fig. 2. The rules are given in the *priority order* and v_0 denotes the center of the cell (ie the in-ball of radius 1).

The labeling function obtained at the end of a run ρ of Mazurkiewicz' algorithm is noted π_ρ . If v is a vertex of \mathbf{G} , the couple $\pi_\rho(v)$ associated with v

¹ this name shall be an integer between 1 and $|V(\mathbf{G})|$ to have an actual Enumeration algorithm. Here we would need more work to enforce this, however since this is not needed for our purpose, these technicalities will be skipped. See [God02] for a way to get a real Enumeration.

Enum1 : Initialization

Guard :

- $\text{Request}(v_0)$

Action :

- $n(v_0) := 0,$
- $\overline{N}(v_0) := N(v_0),$
- $M(v_0) := \emptyset,$
- $a(v_0) := -1.$

Enum2 : Diffusion rule

Guard :

- There exists $v \in B(v_0)$ such that
 $M(v) \neq M(v_0).$
or $(n(v_0), N(v_0)) \notin M(v_0),$
or $\overline{N}(v_0) \neq N(v_0).$

Action :

- $M(v_0) := \bigcup_{w \in B(v_0)} M(w) \cup \{(n(v_0), N(v_0))\}.$
- $\overline{N}(v_0) := N(v_0).$
- $a(v_0) := -1.$

Enum3 : Renaming rule

Guard :

- For all $v \in B(v_0), M(v) = M(v_0).$
- $(n(v_0) = 0)$ or $(n(v_0) > 0$ and there exists $(n(v_0), N) \in M(v_0)$ such that $((N(v_0) \prec N))$).
- $n(v_0) > 0$ and $\exists (n_1, N_1) \in M(v_0)$ such that $\delta_{N_1}(n(v_0)) \geq 2.$

Action :

- $n(v_0) = 1 + \max\{n \in \mathbb{N} \mid (l, n, N) \in M(v_0) \text{ for some } l, N\}.$
- $M(v_0) = M(v_0) \cup \{(n(w), N(w)) \mid w \in B(v_0)\},$
- $a(v_0) = -1.$

gSSPfix : Fix gSSP counter

Guard :

- If there exists $v \in B(v_0), |a(v) - a(v_0)| \geq 2$ or $(M(v) \neq M(v_0) \text{ and } a(v_0) \neq -1)$

Action :

- $a(v_0) := -1.$

gSSP : gSSP rule

Guard :

- $\forall v \in B(v_0), M(v) = M(v_0),$
 $|a(v) - a(v_0)| \leq 1$ and $\neg \mathbb{P}(v_0)$

Action :

- $a(v_0) := 1 + \min\{a(v) \mid v \in B(v_0)\}.$

Decision : Output rule

Guard :

- For all $v \in B(v_0), M(v) = M(v_0)$ and $\mathbb{P}(v_0)$

Action :

- $\text{OUT}(v_0) = f(\mathbf{K}(v_0), w(v_0))$

Fig. 2 Snap-stabilizing algorithm $\mathcal{M}_{f,r}$. The parameters are the functions f and r from Theorem 3. \mathbf{K} is defined by a local procedure and the predicate \mathbb{P} depends on r .

is denoted $(n_\rho(v), M_\rho(v))$. We also note the final local view of v by $N_\rho(v)$. For a given mailbox M and a given $n \in \mathbb{N}$, we note $\text{STRONG}_M(n)$ the local view that dominates all $\overline{N}, (n, \overline{N}) \in M$ (i.e. $\overline{N} \prec \text{STRONG}_M(n)$). Except for the first corrupted stages, $\text{STRONG}_{M(v)}(n)$ is actually the “strongest local view” of n .

Theorem 4 *A run ρ of Mazurkiewicz’ Enumeration Algorithm on \mathbf{G} with any initial values finishes and computes a final labeling π_ρ verifying the following conditions for all vertices v, v' of $V(\mathbf{G})$:*

- 4.i $M_\rho(v) = M_\rho(v').$
- 4.ii $\text{STRONG}_{M_\rho(v')}(n_\rho(v)) = \overline{N}(v) = N_\rho(v).$
- 4.iii $n_\rho(v) = n_\rho(v')$ if and only if $N_\rho(v) = N_\rho(v').$

Proof Even if the model is different, beside technicalities, this can be proved similarly to the proof of [God02].

□

Now we explain how it is possible to extract the map of a minimal base. This is usually done by considering the graphs induced by the numbers and

associated local views that have maximal views. However, here, due to the arbitrary initial failures, the mailbox should be cleaned up before use. It is possible to have some maximal (n, \overline{N}) but n does not actually exist on any v .

Finally, each vertex shall compute locally the set of actual final names from the final mailbox M_ρ . We note \mathbf{G}_ρ the graph defined by

$$\begin{aligned} V_\rho &= \{n_\rho(v) | v \in V(\mathbf{G})\}, \\ A_\rho &= \{(n_\rho(v_1), n_\rho(v_2)) | (v_1, v_2) \in A(G)\}. \end{aligned}$$

For a mailbox M and an integer n , we define the set $V^M(n)$ by induction.

$$\begin{aligned} V_0^M &= \{n\}, \\ V_{i+1}^M &= V_i^M \cup \{t | \exists s V_i^M, \delta_{\text{STRONG}_M(s)}(t) = 1\}. \end{aligned}$$

If i_0 is such that $V_{i_0}^M = V_{i_0+1}^M$ then we define $V^M(n) = V_{i_0}^M$. Finally, we have,

Lemma 3 ([God02]) For all $v \in V(\mathbf{G})$, $V^{M_\rho}(n_\rho(v)) = V_\rho$.

By defining A^M by $\{(n_1, n_2) | n_1, n_2 \in V^M(n) \text{ and } \delta_{\text{STRONG}_M(n_1)}(n_2) = 1\}$, we obtain a graph $\mathbf{G}_{M(v)} = (V^{M(v)}, A^{M(v)})$. We can not readily use $\mathbf{G}_{M(v)}$ since it could be that it is not in \mathcal{F} . We denote by $\mathbf{K}(v)$ a digraph that is in \mathcal{F} and that is a quasi-fibration of $\mathbf{G}_{M(v)}$ of radius $a(v)$ and of center $w(v)$. Such a digraph can be found by a local procedure enumerating all graphs and vertices of \mathcal{F}_\bullet until one is found. This semi-algorithm will always terminate because of the following property.

Proposition 1 Let P be the set of requesting processes. Let v that has been causally influenced by P , and such that $a(v) \geq 0$. The graph \mathbf{G} is a quasi-fibration of $\mathbf{G}_{M(v)}$ of center v and radius $a(v)$.

Proof We add that every $w \in B(v, a(v))$ has been influenced to the statement and prove this new statement by induction on i , the number of steps since P has received the requests.

Initially, at step 1, the requests are being processed by Enum1, i.e. the set of influenced nodes is P and the property holds trivially.

Assume the property holds at step i and consider v_0 a vertex that is activated at round $i + 1$. We have to consider two cases, either v_0 was already influenced at round i or it is a newly influenced node.

If v_0 is a newly influenced node. The only rule of interest is gSSP because other rules are setting $a(v_0)$ to -1 . But we show that v_0 cannot apply this rule. Indeed, assume $M(v_0) \neq \emptyset$, then, the causality path to v_0 starts in a root whose variables have been reset, and from which the causality chain of applications will propagate its new name. So $M(v_0)$ has to be updated to, at least, this name before being able to apply gSSP.

If v_0 has already been influenced then the induction statement applies at the previous round. Denote $a(v_0)$ the value of the counter at the end of round

i and assume that for all $v \in N(v_0)$, $a(v) = a(v_0)$. We prove that the statement holds for $a(v_0) + 1$ at round $i + 1$.

If $a(v_0) = 0$ then, by the same argument as in the previous case, the neighbours of v_0 have all been influenced and the statement holds with a radius 1.

If $a(v_0) > 0$ then the neighbours have been influenced by induction assumption. Moreover, every $v \in N(v_0)$ is the center of a quasi-fibration of radius $a(v_0)$. Therefore, v_0 is the center of a quasi-fibration of radius $a(v_0) + 1$. Similarly, every $w \in B(v, a(v_0))$ has been influenced and the ball $B(v_0, a(v_0) + 1)$ is totally influenced. The statement holds at round $i + 1$. \square

The algorithm from Fig. 2 uses the functions f and r given in the necessary condition of Theorem 3. The two functions are used to define a digraph \mathbf{K} (defined above) and a predicate \mathbb{P} defined below. The predicate needs to make the counter a to increase when what can be extracted from the mailboxes (that is the minimum base of \mathbf{G}) is the same locally. But it must also make the algorithm stop when there is enough information to conclude. This information is enough when the value r for the reconstructed base matches the counter of stability a .

Theorem 5 *With $\mathbb{P}(v) := (a(v) < r(\mathbf{K}(v), n(v)))$, the algorithm $\mathcal{M}_{f,r}$ snap-stabilizes to S for any set P of requested nodes.*

Proof Consider a node v just after it has applied rule Decision, we have $\text{STRONG}(M(v))$ that is constant in the neighbourhood, $r(\mathbf{K}(v), n(v)) \leq a(v)$ and $\text{out}(v) = f(\mathbf{K}(v), w(v))$. Since, by construction, $\mathbf{K}(v)$ is a quasi-fibration of $\mathbf{G}_{M(v)}$ of radius $a(v) \geq r(\mathbf{K}(v), n(v))$ and of center $n(v)$, and since f and r are r -lifting closed, $\text{OUT}(v) = f(\mathbf{K}(v), w(v)) = f(\mathbf{G}_{M(v)}, n(v))$, and $r(\mathbf{K}(v), w(v)) = r(\mathbf{G}_{M(v)}, n(v))$. From Prop. 1, since $a(v) \geq r(\mathbf{G}_{M(v)}, n(v))$ and since f is r -lifting closed, $\text{OUT}(v) = f(\mathbf{G}_{M(v)}, n(v)) = f(\mathbf{G}, v)$.

Since f is an output function for (S, \mathcal{F}) , the OUT labels are correct for S in \mathbf{G} . \square

4.2 Complexity

The algorithm $\mathcal{M}_{f,r}$ is a universal algorithm and therefore for given s, \mathcal{F}) it can have a bigger complexity than a tailored algorithm. However it should be noted that the complexity of $\mathcal{M}_{f,r}$ is divided in two components, the stabilization of the Enumeration part and the increase of the SSP counter until it is greater than r . Note that the former depends on the graph \mathbf{G} only and that the latter depends on the family \mathcal{F} . The complexity from the Enumeration has been shown in [God02] to be, in the Angluin model, at most $t|V(\mathbf{G})|^2$ where t is the sum of the number of vertices and of the highest name n initially known. The proof can be extended to the model of this paper.

5 Conclusion

We have shown that for anonymous networks, the terminating tasks that can be solved by a snap-stabilizing algorithms are exactly the ones that can be solved by a distributed algorithm with explicit termination. This complements the already known task-equivalence between self-stabilizing terminating tasks and distributed tasks computed with implicit termination. The important consequence is that the partial knowledge (like bound on the size, diameter etc ...) that could be used to get explicit termination in the non-stabilizing case are also the ones that can be used to have snap-stabilizing solutions.

A limit of this result is that it does not give the intrinsic complexity of a problem and it could be that solving a problem by snap-stabilization is harder than solving it with explicit termination. The computability is equivalent however whether the complexity is also equivalent is an open problem.

For lack of space, we do not discuss probabilistic snap-stabilization [AD14]. It is not difficult to see that the techniques presented here enable to prove that a task has a probabilistic snap-stabilizing solution if and only if it has a (non-stabilizing) Las Vegas solution.

An interesting open question, as in the self-stabilizing case, would be to find a *direct* way to transform any given anonymous algorithm into a snap-stabilizing one. Such transformation might have benefits regarding the complexity.

The author wishes to thank Jérémie Chalopin for sharing ideas and fruitful discussions about distributed computability in various settings, including some closely related to this paper.

References

- AD14. Karine Altisen and Stéphane Devismes. On probabilistic snap-stabilization. In *Distributed Computing and Networking*, Lecture Notes in Computer Science, page 272–286. Springer Berlin Heidelberg, Jan 2014.
- Ang80. D. Angluin. Local and global properties in networks of processors. In *Proceedings of the 12th Symposium on Theory of Computing*, pages 82–93, 1980.
- BCG⁺96. Paolo Boldi, Bruno Codenotti, Peter Gemmell, Shella Shammah, Janos Simon, and Sebastiano Vigna. Symmetry breaking in anonymous networks: Characterizations. In *Proc. 4th Israeli Symposium on Theory of Computing and Systems*, pages 16–26. IEEE Press, 1996.
- BDPV99. Alain Bui, Ajoy Kumar Datta, Franck Petit, and Vincent Villain. State-optimal snap-stabilizing pif in tree networks. In *Workshop on Self-stabilizing Systems*, ICDCS '99, pages 78–85. IEEE Computer Society, 1999.
- BV01. Paolo Boldi and Sebastiano Vigna. An effective characterization of computability in anonymous networks. In Jennifer L. Welch, editor, *Distributed Computing. 15th International Conference, DISC 2001*, volume 2180 of *Lecture Notes in Computer Science*, pages 33–47. Springer-Verlag, 2001.
- BV02a. Paolo Boldi and Sebastiano Vigna. Fibrations of graphs. *Discrete Math.*, 243(243):21–66, 2002.
- BV02b. Paolo Boldi and Sebastiano Vigna. Universal dynamic synchronous self-stabilization. *Distr. Computing*, (15), 2002.

- CDD⁺16. Alain Cournier, Ajoy Kumar Datta, Stéphane Devismes, Franck Petit, and Vincent Villain. The expressive power of snap-stabilization. *Theor. Comput. Sci.*, 626:40–66, 2016.
- CDV09. Alain Cournier, Stéphane Devismes, and Vincent Villain. Light enabling snap-stabilization of fundamental protocols. *ACM Trans. Auton. Adapt. Syst.*, 4(1):6:1–6:27, 2009.
- CGM08. Jérémie Chalopin, Emmanuel Godard, and Yves Métivier. *Local Terminations and Distributed Computability in Anonymous Networks*, volume 5218 of *Lecture Notes in Computer Science*, page 47–62. Springer Berlin / Heidelberg, 2008. 10.1007/978-3-540-87779-0 4.
- CGM12. Jérémie Chalopin, Emmanuel Godard, and Yves Métivier. Election in partially anonymous networks with arbitrary knowledge in message passing systems. *Distributed Computing*, 25(4):297–311, August 2012.
- Cha06. Jérémie Chalopin. *Algorithmique distribuée, calculs locaux et homomorphismes de graphes*. PhD thesis, Université de Bordeaux I, 2006.
- CM07. Jérémie Chalopin and Yves Métivier. An efficient message passing election algorithm based on mazurkiewicz’s algorithm. *Fundam. Inform.*, 80(1-3):221–246, 2007.
- CR79. Ernest Chang and Rosemary Roberts. An improved algorithm for decentralized extrema-finding in circular configurations of processes. *Commun. ACM*, 22(5):281–283, May 1979.
- CT96. T. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *journal of the ACM*, 43(2):225–267, Mar 1996.
- Dol00. Schlomi Dolev. *Self-Stabilization*. MIT Press, 2000.
- GM02a. E. Godard and Y. Métivier. A characterization of families of graphs in which election is possible (ext. abstract). In M. Nielsen and U. Engberg, editors, *Proc. of Foundations of Software Science and Computation Structures, FOSSACS’02*, number 2303 in LNCS, pages 159–171. Springer-Verlag, 2002.
- GM02b. Emmanuel Godard and Yves Métivier. A characterization of families of graphs in which election is possible. In Mogens Nielsen and Uffe Engberg, editors, *Foundations of Software Science and Computation Structures*, volume 2303 of *Lecture Notes in Computer Science*, pages 159–171. Springer Berlin / Heidelberg, 2002. 10.1007/3-540-45931-6_12.
- GMT10. Emmanuel Godard, Yves Métivier, and Gerard Tel. Termination detection of local computations. Technical Report arXiv:1001.2785v2, January 2010.
- God02. E. Godard. A self-stabilizing enumeration algorithm. *Information Processing Letters*, 82(6):299–305, 2002.
- LeL77. G. LeLann. Distributed systems: Towards a formal approach. In B. Gilchrist, editor, *Information processing’77*, pages 155–160. North-Holland, 1977.
- Maz88. A. Mazurkiewicz. Solvability of the asynchronous ranking problem. *Inf. Processing Letters*, 28:221–224, 1988.
- Maz97. A. Mazurkiewicz. Distributed enumeration. *Inf. Processing Letters*, 61:233–239, 1997.
- MMW97. Yves Métivier, Anca Muscholl, and Pierre-André Wacrenier. About the local detection of termination of local computations in graphs. In D. Krizanc and P. Widmayer, editors, *SIROCCO 97 - 4th International Colloquium on Structural Information & Communication Complexity*, Proceedings in Informatics, pages 188–200. Carleton Scientific, 1997.
- SSP85. B. Szymanski, Y. Shy, and N. Prywes. Terminating iterative solutions of simultaneous equations in distributed message passing systems. In *Proc. of the 4th Symposium of Distributed Computing*, pages 287–292, 1985.
- Tel00. G. Tel. *Introduction to distributed algorithms*. Cambridge University Press, 2000.
- YK96. M. Yamashita and T. Kameda. Computing on anonymous networks: Part i - characterizing the solvable cases. *IEEE Transactions on parallel and distributed systems*, 7(1):69–89, 1996.