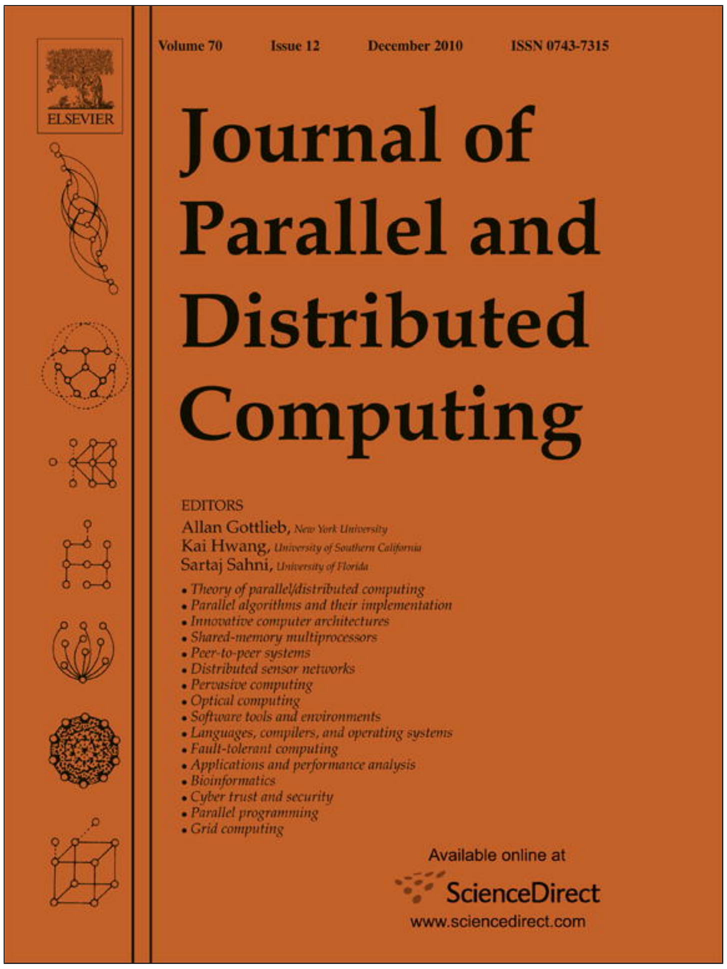
Volume 70 Issue 12 December 2010 ISSN 0743-7315
Journal of Parallel and Distributed Computing
ELSEVIER
EDITORS
Allan Gottlieb, New York University
Kai Hwang, University of Southern California
Sartaj Sahni, University of Florida
Available online at
ScienceDirect
www.sciencedirect.com

J. Parallel Distrib. Comput. 70 (2010) 1220–1230

Contents lists available at ScienceDirect

J. Parallel Distrib. Comput.

journal homepage: www.elsevier.com/locate/jpdc

Snap-stabilization in message-passing systems[☆]

Sylvie Delaët[a], Stéphane Devismes[b], Mikhail Nesterenko[c], Sébastien Tixeuil[d,*]

[a] LRI UMR 8623, Université de Paris-Sud, France
[b] VERIMAG UMR 5104, Université Joseph Fourier, France
[c] Computer Science Department, Kent State University, United States
[d] LIP6 UMR 7606, Université Pierre et Marie Curie, France

A R T I C L E   I N F O

Article history:
Received 17 August 2009
Received in revised form
14 April 2010
Accepted 24 April 2010
Available online 25 May 2010

Keywords:
Distributed systems
Self-stabilization
Snap-stabilization

A B S T R A C T

In this paper, we tackle the problem of *snap-stabilization* in message-passing systems. Snap-stabilization allows designing protocols that withstand transient faults: indeed, any computation that is started after faults cease *immediately* satisfies the expected specification.

Our contribution is twofold: we demonstrate that in message-passing systems (i) snap-stabilization is impossible for nontrivial problems if channels are of finite yet unbounded capacity, and (ii) snap-stabilization becomes possible in the same setting with bounded capacity channels. The latter contribution is constructive, as we propose two snap-stabilizing protocols for propagation of information with feedback and mutual exclusion.

Our work opens exciting new research perspectives, as it enables the snap-stabilizing paradigm to be implemented in actual networks.

1. Introduction

Self-stabilization [18,19,28] is an elegant approach to forward failure recovery: regardless of the global state to which the failure drives the system, after the failures stop, a self-stabilizing system is guaranteed to resume correct operation within *finite time*. This guarantee comes at the expense of temporary safety violation in the sense that a self-stabilizing system may behave incorrectly as it recovers, without a user of the system being notified of this misbehavior.

Bui et al. [9] introduce the related concept of *snap-stabilization* which guarantees that a protocol *immediately* operates correctly, regardless of the arbitrary initial state of the system. Snap-stabilization offers stronger fault-tolerance properties than self-stabilization: regardless of the global state to which the failure drives the system, after the failures stop, a snap-stabilizing system immediately resumes correct behavior.

The notion of safety that is guaranteed by snap-stabilization is orthogonal to the notion of safety that is guaranteed by super-stabilization [20] or safe stabilization [22]. In [20,22], some safety predicates on configurations and executions are preserved at all times while the system is running. Of course, not all safety predicates can be guaranteed when the system is started from an arbitrary global state. In contrast, snap-stabilization's notion of safety is *user-centric*: when the user initiates a request, then the received response is correct. However, between the request and the response, the system can behave arbitrarily (except from giving an erroneous response to the user). In the snap-stabilizing model, all user-safety predicates can be guaranteed while recovering from arbitrary states. Then, if the system user is sensitive to safety violation, snap-stabilization becomes an attractive option.

However, nearly every snap-stabilizing protocol presented so far assumes a high level communication model in which any process is able to read the states of every communication neighbor and update its own state in a single atomic step (this model is often referred to as the *shared memory model with composite atomicity* in the literature). Designing protocols with forward recovery properties (such as self-stabilizing and snap-stabilizing ones) using lower level communication models such as asynchronous message-passing is rather challenging. In such models, a process may either send a message to a single neighbor or receive a message from a single neighbor (but not both) together with some local computations; also messages in transit could be lost or duplicated. It is especially important to consider these low level models since Varghese and Jayaram [30] proved that simple process crashes and restarts and unreliable communication channels can drive protocols to arbitrary states.

☆ A preliminary version of this paper will be presented in Delaët et al. (2009) [16].
∗ Corresponding author.
E-mail addresses: sylvie.delaet@lri.fr (S. Delaët), stephane.devismes@imag.fr (S. Devismes), mikhail@cs.kent.edu (M. Nesterenko), sebastien.tixeuil@lip6.fr (S. Tixeuil).

0743-7315/$ – see front matter © 2010 Published by Elsevier Inc.
doi:10.1016/j.jpdc.2010.04.002

1.1. Related works

Several papers investigate the possibility of self-stabilization in message-passing systems [23,26,2,1,25,29,5,6,17]. The crucial assumption for communication channels is their *boundedness*. That is, whether or not processes are aware of the maximum number of messages that can be in transit in a particular channel. Gouda and Multari [23] show that for a wide class of problems such as the alternating bit protocol (ABP), deterministic self-stabilization is impossible using bounded memory per process when channel capacities are unbounded. They also present a self-stabilizing version of the ABP with unbounded channels that uses unbounded memory per process. Afek and Brown [2] present a self-stabilizing ABP replacing unbounded process memory by an infinite sequence of random numbers. Katz and Perry [26] derive a self-stabilizing ABP to construct a self-stabilizing snapshot protocol. In turn, the snapshot protocol allows to transform almost all non-stabilizing protocols into self-stabilizing ones. Delaët et al. [17] propose a method to design self-stabilizing protocols with bounded memory per process in message-passing systems with unreliable channels with unbounded capacity for a class of fix-point problems. Awerbuch et al. [7] introduce the property of *local correctability* and demonstrate that protocols that are locally correctable can be self-stabilized using bounded memory per process in spite of unbounded capacity channels. Guaranteeing self-stabilization with bounded memory per process for general (*i.e.* ABP-like) specifications requires considering bounded capacity channels [1,25,29,5,6]. In particular, Varghese [29] presented such self-stabilizing solutions for a wide class of problems, *e.g.* token circulation and propagation of information with feedback (PIF).

A number of snap-stabilizing protocols are presented in the literature. In particular, PIF is the "benchmark" application for snap-stabilization [10,13]. Moreover [14,12] present token circulation protocols. Snap-stabilization has also been investigated for fix-point tasks such as binary search tree construction [8] and cut-set detection [15]. Following the scheme of [26], Cournier et al. [11] propose a method to add snap-stabilization to a large class of protocols. To our knowledge, the only paper that deals with snap-stabilization in message-passing networks is [21]. However the snap-stabilizing snapshot protocol that is presented in [21] for multi-hops networks relies on the assumption that there exists an underlying snap-stabilizing protocol for *one-hop* message transmission, we do not make such an assumption here. To date, the question whether this assumption can be implemented remains open.

1.2. Our contribution

In this paper, we address the problem of *snap-stabilization* in one-hop message-passing systems. Our contribution is twofold:

(1) We show that contrary to the high level shared memory model, snap-stabilization is strictly more difficult to guarantee than self-stabilization in the low level message-passing model. In more detail, for nontrivial distributed problem specifications, there exists no snap-stabilizing (even with unbounded memory per process) solution in message-passing systems with unbounded yet finite capacity channels. This is in contrast to the self-stabilizing setting, where solutions with unbounded memory per process [23], unbounded sequences of random numbers [2], or operating on a restricted set of specifications [17,7] do exist.

(2) We prove that snap-stabilization in the low level message-passing model is feasible when channels have bounded capacity. Our proof is constructive both for dynamic and fix-point distributed specifications, as we present snap-stabilizing protocols for PIF and mutual exclusion.

1.3. Outline

The rest of the paper is organized as follows. We define the message-passing model in Section 2. In the same section, we describe the notion of snap-stabilization and problem specifications. In Section 3, we exhibit a wide class of problems that have no snap-stabilizing solution in message-passing systems with unbounded capacity channels. We present snap-stabilizing algorithms for the message-passing systems with bounded capacity channels in Section 4. We conclude the paper in Section 5.

2. Preliminaries

2.1. Computational model

We consider distributed systems having *n processes* and a *fully-connected topology*: any two distinct processes can communicate by sending messages through a bidirectional link, i.e., two channels in the opposite direction.

A process is an asynchronous sequential deterministic machine that uses a local memory, a local algorithm, and input/output capabilities. The local algorithm modifies the state of the process memory, and sends/receives messages through channels.

We assume that the channels incident to a process are locally distinguished by a *channel number*. For the sake of simplicity, every process numbers its channels from 1 to $n-1$, and in the code of any process we simply denote by the *label* q the number of the channel incoming from the process q.[1] We assume that the channels are FIFO (meaning that messages are received in the order they are sent) but not necessarily *reliable* (messages can be lost). However they all satisfy the following fairness property: if a sender process s sends infinitely many messages to a receiver process r, then infinitely many messages are eventually received by r from s. Any message that is never lost is received in finite but unbounded time.

The messages are of the following form: ⟨*message-type*, *message-value*⟩. The *message-value* field is omitted if the message does not carry any value. A message may also contain more than one *message-value*.

A *protocol* is a collection of n local algorithms, one held by each process. A local algorithm consists of a collection of actions. Any action is of the following form:

$\langle label \rangle$ :: $\langle guard \rangle \rightarrow \langle statement \rangle$.

A *guard* is a boolean expression over the variables of a process and/or an *input* message. A *statement* is a sequence of assignments of the process variables and/or message *sends*. An action can be executed only if its guard is true. We assume that the actions are *atomically* executed, meaning that the evaluation of the guard and the execution of the corresponding statement, if executed, are done in one atomic step. An action is *enabled* when its guard is true. Any *continuously* enabled action is executed within finite yet unbounded time.

We reduce the *state* of each process to the state of its local memory, and the state of each link to its content. The global state of the system, referred to as *configuration*, is defined as the product of the states of the memories of processes and the contents of the links.

We describe a distributed system as a *transition system* [27] $\mathcal{S} = (\mathcal{C}, \mapsto, \mathcal{I})$ such that: $\mathcal{C}$ is a set of configurations, $\mapsto$ is a binary transition relation on $\mathcal{C}$, and $\mathcal{I} \subseteq \mathcal{C}$ is the set of initial configurations. Here, we only consider systems $\mathcal{S} = (\mathcal{C}, \mapsto, \mathcal{I})$ such that $\mathcal{I} = \mathcal{C}$, meaning that any possible configuration can be initial.

[1] The label q does not denote the identifier of q. When necessary, we will use the notation ID_q to denote the identifier of q.

An *execution* of $\mathcal{S}$ is a *maximal* sequence of configurations $\gamma_0, \ldots, \gamma_{i-1}, \gamma_i, \ldots$ such that: $\gamma_0 \in \mathcal{I}$ and $\forall i > 0$, $\gamma_{i-1} \mapsto \gamma_i \wedge \gamma_{i-1} \neq \gamma_i$. Any transition $\gamma_{i-1} \mapsto \gamma_i$ is called a *step* and materializes the fact that some processes and/or links change their states. A process changes its state by executing an enabled action of its local algorithm. The state of a link is modified each time a message is sent, received, or lost.

2.2. Snap-stabilization

In the following, we call *specification* a predicate defined over sequences of configurations.

Definition 1 (*Snap-Stabilization [9]*)**.** Let $\mathcal{SP}_{\mathcal{T}}$ be a specification. A protocol $\mathcal{P}$ is *snap-stabilizing* for $\mathcal{SP}_{\mathcal{T}}$ *if and only if* starting from any configuration, any execution of $\mathcal{P}$ satisfies $\mathcal{SP}_{\mathcal{T}}$.

It is important to note that snap-stabilization is suited for *user-centric* specifications. Such specifications are based on a sequence of actions (request, start, etc.) rather than a particular subset of configurations (e.g., the *legitimate configurations*) and are composed of two main properties:

(1) **Start.** Upon an *external* (*w.r.t.* the protocol) *request*, a process (called the *initiator*) starts within finite time a distributed *computation* of specific task $\mathcal{T}$[2] by executing a special type of action called *starting action*.
(2) **User-Safety.** Any computation of $\mathcal{T}$ that has been started is correctly performed.

In snap-stabilization, we consider the system after the last fault occurs: hence we study the behavior of the system from an arbitrary configuration, yet considered as the initial one, and we assume that no more faults occur. Starting from such an arbitrary configuration (*i.e.*, after the end of faults), any snap-stabilizing protocol always guarantee the **start** and the **user-safety** properties. Hence, snap-stabilization is attractive for system users: when a user make a request, it has the guarantee that the requested task will be correctly performed regardless of the initial configuration of the system under the assumption that no further error occurs. We do not have such a guarantee with self-stabilizing protocols. Indeed, while a snap-stabilizing protocol ensures that any request is satisfied despite the arbitrary initial configuration, a self-stabilizing protocol often needs to repeat its computations an unbounded number of times before guaranteeing the proper processing of any request.

3. Message-passing systems with unbounded capacity channels

Alpern and Schneider [3] observe that a specification is an intersection of *safety* and *liveness* properties. They define a *safety* property as a set of "bad things" that must never happen. We now introduce the notion of *safety-distributed* specifications and show that no problem having such a specification admits a snap-stabilizing solution in message-passing systems with finite yet unbounded capacity channels. Intuitively, safety-distributed specification has a safety property that depends on the behavior of more than one process. That is, certain process behaviors may satisfy safety if done sequentially, while violate it if done concurrently. For example, in mutual exclusion, any requesting process must eventually execute the critical section but if several requesting processes execute the critical section concurrently, the safety is violated. Roughly speaking, the class of *safety-distributed* specifications includes all distributed problems where processes need to exchange messages in order to preclude any safety violation.

[2] E.g., a broadcast, a circulation of a single token, ...

The following three definitions are used to formalize the *safety-distributed* specifications.

Definition 2 (*Abstract Configuration*)**.** We call an *abstract configuration* any configuration restricted to the state of the processes (i.e., the state of each link has been removed).

Definition 3 (*State-Projection*)**.** Let γ be a configuration and p be a process. The *state-projection* of γ on p, denoted $\phi_p(\gamma)$, is the local state of p in γ. Similarly, the *state-projection* of γ on all processes, denoted $\phi(\gamma)$, is the product of the local states of all processes in γ (*n.b.* $\phi(\gamma)$ is an abstract configuration).

Definition 4 (*Sequence-Projection*)**.** Let $s = \gamma_0, \gamma_1, \ldots$ be a configuration sequence and p be a process. The *sequence-projection* of s on p, denoted $\Phi_p(s)$, is the state sequence $\phi_p(\gamma_0), \phi_p(\gamma_1), \ldots$. Similarly, the *sequence-projection* of s on all processes, denoted $\Phi(s)$, is the abstract configuration sequence $\phi(\gamma_0), \phi(\gamma_1), \ldots$.

Definition 5 (*Safety-Distributed*)**.** A specification $\mathcal{SP}$ is *safety-distributed* if there exists a sequence of abstract configurations $\mathtt{BAD}$, called *bad-factor*, such that:

(1) For each execution e, if there exist three configuration sequences e_0, e_1, and e_2 such that $e = e_0e_1e_2$ and $\Phi(e_1) = \mathtt{BAD}$, then e does not satisfy $\mathcal{SP}$.
(2) For each process p, there exists at least one execution e_p satisfying $\mathcal{SP}$ where there exist three configuration sequences e_p^0, e_p^1, and e_p^2 such that $e_p = e_p^0e_p^1e_p^2$ and $\Phi_p(e_p^1) = \Phi_p(\mathtt{BAD})$.

Almost all classical problems of distributed computing have *safety-distributed* specifications including all synchronization and resource allocation problems. For example, in mutual exclusion a *bad-factor* is any sequence of abstract configurations where several requesting processes execute the critical section concurrently. For the PIF, the *bad-factor* consists in the sequence of abstract configurations where the initiator decides of the termination of a PIF it started while some other processes are still broadcasting the message.

We now consider a message-passing system with unbounded capacity channels and show the impossibility of *snap-stabilization* for *safety-distributed* specifications in that case. Since most of classical synchronization and resource allocation problems are safety-distributed, this result prohibits the existence of snap-stabilizing protocols in message-passing systems if no further assumption is made.

We prove the theorem by showing that a "bad" execution can be obtained by filling the communication channels with messages entailing an unsafe state, and no process can detect that those messages occur due to errors because of the safety-distributed nature of the specification.

Theorem 1. *There exists no safety-distributed specification that admits a snap-stabilizing solution in message-passing systems with unbounded capacity channels.*

Proof. Let $\mathcal{SP}$ be a *safety-distributed* specification and $\mathtt{BAD} = \alpha_0$, $\alpha_1, \ldots$ be a *bad-factor* of $\mathcal{SP}$.

Assume, for the purpose of contradiction, that there exists a protocol $\mathcal{P}$ that is snap-stabilizing for $\mathcal{SP}$. By Definition 5, for each process p, there exists an execution e_p of $\mathcal{P}$ that can be split into three execution factors e_p^0, $e_p^1 = \beta_0^p, \beta_1^p, \ldots$, and e_p^2 such that $e_p = e_p^0e_p^1e_p^2$ and $\Phi_p(e_p^1) = \Phi_p(\mathtt{BAD})$. Let us denote by $MesSeq_p^q$ the ordered sequence of messages that p receives from any process q in e_p^1. Consider now the configuration γ_0 such that:

(1) $\phi(\gamma_0) = \alpha_0$.

(2) For all pairs of distinct processes p and q, the link $\{p, q\}$ has the following state in γ_0:
   (a) The messages in the channel from q to p are exactly the sequence $MesSeq_p^q$ (keeping the same order).
   (b) The messages in the channel from p to q are exactly the sequence $MesSeq_q^p$ (keeping the same order).

(It is important to note that we have the guarantee that γ_0 exists because we assume unbounded capacity channels. Assuming channels with a bounded capacity c, no configuration satisfies (2) if there are at least two distinct processes p and q such that $|MesSeq_p^q| > c$.)

As $\mathcal{P}$ is snap-stabilizing, γ_0 is a possible initial configuration of $\mathcal{P}$. To obtain the contradiction, we now show that there is an execution starting from γ_0 that does not satisfy $\mathcal{SP}$. By definition, $\phi(\gamma_0) = \alpha_0$. Consider a process p and the two first configurations of e_p^1: β_0^p and β_1^p. Any message that p receives in $\beta_0^p \mapsto \beta_1^p$ can be received by p in the first step from γ_0: $\gamma_0 \mapsto \gamma_1$. Now, $\phi_p(\gamma_0) = \phi_p(\beta_0^p)$. So, p can behave in $\gamma_0 \mapsto \gamma_1$ as in $\beta_0^p \mapsto \beta_1^p$. In this case, $\phi_p(\gamma_1) = \phi_p(\beta_1^p)$. Hence, if each process q behaves in $\gamma_0 \mapsto \gamma_1$ as in the first step of its execution factor e_q^1, we obtain a configuration γ_1 such that $\phi(\gamma_1) = \alpha_1$. By induction principle, there exists an execution prefix starting from γ_0 denoted *PRED* such that $\Phi(PRED) = \mathtt{BAD}$. As $\mathcal{P}$ is snap-stabilizing, there exists an execution *SUFF* that starts from the last configuration of *PRED*. Now, merging *PRED* and *SUFF* we obtain an execution of $\mathcal{P}$ that does not satisfy $\mathcal{SP}$ — this contradicts the fact that $\mathcal{P}$ is snap-stabilizing. $\square$

The proof of Theorem 1 hinges on the fact that after some transient faults the configuration may contain an unbounded number of arbitrary messages. Note that a safety-distributed specification involves more than one process and thus requires the processes to communicate to ensure that safety is not violated. However, with unbounded channels, each process cannot determine if the incoming message is indeed sent by its neighbor or is the result of faults. Thus, the communication is thwarted and the processes cannot differentiate safe and unsafe behavior.

4. Message-passing systems with bounded capacity channels

We now consider systems with bounded capacity channels. In such systems, we assume that if a process sends a message in a channel that is full, then the message is lost. For the sake of simplicity, we restrict our study to systems with single-message capacity channels. The extension to an arbitrary but known bounded message capacity is straightforward (by applying the principles described in [29,6,7]).

Below, we propose two snap-stabilizing protocols for fully-connected networks (Algorithms 1 and 3) respectively for the PIF (*Propagation of Information with Feedback*) and mutual exclusion problems. The mutual exclusion algorithm is obtained using several PIFs.

4.1. PIF for fully-connected networks

4.1.1. Principle

Informally, the PIF algorithm (also called *echo* algorithm) can be described as follows: upon a request, a process – called *initiator* – starts the first phase of the PIF by broadcasting a data message m into the network (the *broadcast phase*). Then, each non-initiator *acknowledges*[3] to the initiator the receipt of m (the *feedback phase*). The PIF terminates by a *decision event* at the initiator.[4] This decision is taken following the acknowledgments for m, meaning that when the decision event for the message m occurs, the last acknowledgments the initiator delivers from all other processes are acknowledgments for m.

Note that any process may need to initiate a PIF. Thus, any process can be the initiator of a PIF and several PIFs may run concurrently. Hence, any PIF protocol has to cope with concurrent PIFs.

More formally, the PIF problem can be specified as follows:

Specification 1 (*PIF-Exec*). *An execution e satisfies Predicate* PIF-Exec if and only if *e satisfies the following two properties:*

(1) **Start.** *Upon a request to broadcast a message m, a process p starts a PIF of m.*
(2) **User-Safety.** *During any PIF of some message m started by p:*
   - *Every process other than p receives m.*
   - *p receives acknowledgments for m from all other processes.*
   - *p executes the decision event in finite time, this decision is taken following the acknowledgments for m.*

We now outline a snap-stabilizing implementation of the PIF called $\mathcal{PIF}$ (the code is provided in Algorithm 1). In the following (and in the rest of the paper), the *message-values* will be replaced by "—" when they have no impact on the reasoning.

A basic PIF implementation requires the following input/output variables:

- The variable $\mathtt{Req}_p$ is used to manage the requests at the process p. The value of $\mathtt{Req}_p$ belongs to $\{\mathtt{Wait}, \mathtt{In}, \mathtt{Done}\}$. $\mathtt{Req}_p = \mathtt{Wait}$ means that a PIF is requested. $\mathtt{Req}_p = \mathtt{In}$ means that the protocol is currently executing a PIF. $\mathtt{Req}_p = \mathtt{Done}$ means no PIF is under execution, i.e., the protocol is waiting for the next request.
- The buffer variable $\mathtt{BMes}_p$ is used to store the message to broadcast.
- The array $\mathtt{FMes}_p[1 \ldots n-1]$ is used to store acknowledgments, i.e., $\mathtt{FMes}_p[q]$ contains the acknowledgment for the broadcast message coming from q.

Using these variables, the protocol proceeds as follows: assume that a user wants to broadcast a message m from process p. It waits until the current PIF terminates (i.e., until $\mathtt{Req}_p = \mathtt{Done}$) even if the current PIF is due to a fault, and then notifies the request to p by setting $\mathtt{BMes}_p$ to m and $\mathtt{Req}_p$ to $\mathtt{Wait}$. Consequently to this request, a PIF is started (in particular, $\mathtt{Req}_p$ is set to $\mathtt{In}$). The current PIF terminates when $\mathtt{Req}_p$ is set to $\mathtt{Done}$ (this latter assignment corresponds to the decision event). Between the start and the termination, the protocol has to generate two types of events at the application level. First, a "**B-receive** $\langle m \rangle$ **from** p" event at each other process q. When this event occurs, the application at q is assumed to process the broadcast message m and put an acknowledgment Ack_m into $\mathtt{FMes}_q[p]$. The protocol then transmits $\mathtt{FMes}_q[p]$ to p: this generates a "**F-receive** $\langle Ack_m \rangle$ **from** q" event at p so that the application at p can access the acknowledgment.

Note that the protocol has to operate correctly despite arbitrary messages in the channels left after the faults. Note also that messages may be lost. To counter the message loss the protocol repeatedly sends duplicate messages. To deal with the arbitrary initial messages and the duplicates, we mark each message with a flag that ranges from 0 to 4. Two arrays are used to manage the flag values:

- In $\mathtt{State}_p[q]$, process p stores a flag value that it attaches to the messages it sends to its qth neighbor.

3 An acknowledgment is a message sent by the receiving process to inform the sender about data it has correctly received (*cf.* [27]).

4 That is, an event that causally depends on an action at each process (this definition comes from [27]).

- In $\texttt{NState}_p[q]$, p stores the flag of the last message it has accepted from its qth neighbor.

Using these two arrays, our protocol proceeds as follows: when p starts a PIF, it initializes $\texttt{State}_p[q]$ to 0, for every index q. The PIF terminates when $\texttt{State}_p[q] \geq 4$ for every index q.

During the PIF, p repeatedly sends $\langle \texttt{PIF}, \texttt{BMes}_p, -, \texttt{State}_p[q], - \rangle$ to every process q such that $\texttt{State}_p[q] < 4$. When a process q receives $\langle \texttt{PIF}, B, -, pState, - \rangle$ from p, q updates $\texttt{NState}_q[p]$ to $pState$. Then, if $pState < 4$, q sends $\langle \texttt{PIF}, -, \texttt{FMes}_q[p], -, \texttt{NState}_q[p] \rangle$ to p. Finally, p increments $\texttt{State}_p[q]$ only when it receives a $\langle \texttt{PIF}, -, F, -, qNState \rangle$ message from q such that $qNState = \texttt{State}_p[q]$ and $qNState < 4$.

The main idea behind the algorithm is as follows: assume that p starts to broadcast the message m. While $\texttt{State}_p[q] < 4$, $\texttt{State}_p[q]$ is incremented only when p received a message $\langle \texttt{PIF}, -, F, -, qNState \rangle$ from q such that $qNState = \texttt{State}_p[q]$. So, $\texttt{State}_p[q]$ will be equal to 4 only after p successively receives $\langle \texttt{PIF}, -, F, -, qNState \rangle$ messages from q with the flag values 0, 1, 2, and 3. Now, initially there is at most one message in the channel from p to q and at most another one in the channel from q to p. So these messages can only cause at most two increments of $\texttt{State}_p[q]$. Finally, the arbitrary initial value of $\texttt{NState}_q[p]$ can cause at most one increment of $\texttt{State}_p[q]$. Hence, since $\texttt{State}_p[q] = 3$, we have the guarantee that p will increment $State_p[q]$ to 4 only after it receives a message sent by q after q receives a message sent by p. That is, this message is a correct acknowledgment of m by q.

It remains to describe the generation of the **B-receive** and **F-receive** events:

- Any process q receives at least four copies of the broadcast message from p. But, q generates a **B-receive** event only once for each broadcast message from p: when q switches $\texttt{NState}_q[p]$ to 3.
- After it starts, p is sure to receive the correct feedback from q since it receives from q a $\langle \texttt{PIF}, -, F, -, qNState \rangle$ message such that $qNState = \texttt{State}_p[q] = 3$. As previously, to limit the number of events, p generates a **F-receive** event only when it switches $\texttt{State}_p[q]$ from 3 to 4. The next copies are ignored.

4.1.2. Correctness

Below, we prove that Algorithm $\mathcal{PIF}$ is a snap-stabilizing PIF algorithm for fully-connected networks. Note that the principle of proof is similar to [21]. However, the proof details are quite different, mainly due to the fact that our communication model is weaker than the one used in [21]. Remember that in [21], authors abstract the activity of communication links by assuming an underline snap-stabilizing ARQ data link algorithm. Here, we just assume that links are fair-lossy and FIFO.

The proof of snap-stabilization of $\mathcal{PIF}$ consists in showing that, despite the arbitrary initial configuration, any execution of $\mathcal{PIF}$ always satisfies the **start** and the **user-safety** properties of Specification 1.

Considering an arbitrary initial configuration, we state the **start** property (Corollary 1) in two steps:

(S1) We first prove that each time a user wants to broadcast a message from some process p, then it can eventually submit its request to the process (i.e. it is eventually enabled to execute $\texttt{Req}_p \leftarrow \texttt{Wait}$).
(S2) We then prove that once a request has been submitted to some process p, the process starts (i.e., executes action $\texttt{A}_1$) the corresponding PIF within finite time.

To prevent the aborting of a previous PIF, a user can initiate a request at some process p only if $\texttt{Req}_p = \texttt{Done}$. Hence, to show (S1), we show that from any configuration where $\texttt{Req}_p \in \{\texttt{Wait}, \texttt{In}\}$, the system eventually reaches a configuration where $\texttt{Req}_p = \texttt{Done}$. This latter claim is proven in two stages:

- We first show in Lemma 1 that from any configuration where $\texttt{Req}_p = \texttt{Wait}$, in finite time the system reaches a configuration where $\texttt{Req}_p = \texttt{In}$.
- We then show in Lemma 3 that from any configuration where $\texttt{Req}_p = \texttt{In}$, in finite time the system reaches a configuration where $\texttt{Req}_p = \texttt{Done}$.

Lemma 1. *Let p be any process. From any configuration where $\texttt{Req}_p = \texttt{Wait}$, in finite time the system reaches a configuration where $\texttt{Req}_p = \texttt{In}$.*

Proof. When $\texttt{Req}_p = \texttt{Wait}$, action $\texttt{A}_1$ is continuously enabled at p and by executing $\texttt{A}_1$, p sets $\texttt{Req}_p$ to $\texttt{In}$. □

The next technical lemma is used in the proof of Lemma 3.

Lemma 2. *Let p and q be two distinct processes. From any configuration where $(\texttt{Req}_p = \texttt{In}) \wedge (\texttt{State}_p[q] < 4)$, $\texttt{State}_p[q]$ is incremented in finite time.*

Proof. Assume, for the sake of contradiction, that $\texttt{Req}_p = \texttt{In}$ and $\texttt{State}_p[q] = i$ with $i < 4$ but $\texttt{State}_p[q]$ is never incremented. Then, $\texttt{Req}_p = \texttt{In}$ and $\texttt{State}_p[q] = i$ hold forever and by checking actions $\texttt{A}_2$ and $\texttt{A}_3$, we know that:

- p only sends to q messages of the form $\langle \texttt{PIF}, -, -, i, - \rangle$.
- p sends such messages infinitely many times.

As a consequence, q eventually only receives from p messages of the form $\langle \texttt{PIF}, -, -, i, - \rangle$ and q receives such messages infinitely often. By action $\texttt{A}_3$, $\texttt{NState}_q[p] = i$ eventually holds forever. From that point, any message that q sends to p is of the form $\langle \texttt{PIF}, -, -, -, i \rangle$. Also, as $i < 4$ and q receives infinitely many messages from p, q sends infinitely many messages of the form $\langle \texttt{PIF}, -, -, -, i \rangle$ to p. Hence, p eventually receives $\langle \texttt{PIF}, -, -, -, i \rangle$ from q and, as a consequence, increments $\texttt{State}_p[q]$ (see action $\texttt{A}_3$) — a contradiction. □

Lemma 3. *Let p be any process. From any configuration where $\texttt{Req}_p = \texttt{In}$, in finite time the system reaches a configuration where $\texttt{Req}_p = \texttt{Done}$.*

Proof. Assume, for the sake of contradiction, that from some configuration $\texttt{Req}_p \neq \texttt{Done}$ forever. Then, $\texttt{Req}_p = \texttt{In}$ eventually holds forever by Lemma 1. Now, by Lemma 2 and owing the fact that for every index q, $\texttt{State}_p[q]$ cannot decrease while $\texttt{Req}_p = \texttt{In}$, we can deduce that p eventually satisfies "$\forall q \in [1 \dots n-1]$, $\texttt{State}_p[q] = 4$" forever. In this case, p eventually sets $\texttt{Req}_p$ to $\texttt{Done}$ by action $\texttt{A}_2$ — a contradiction. □

As explained before, Lemmas 1 and 3 proves (S1). Lemma 1 also implies (S2) because $\texttt{A}_1$ (the starting action) is the only action where $\texttt{Req}_p$ is set to $\texttt{In}$. Hence, we have the following corollary:

Corollary 1 (*Start*). *Starting from any configuration, $\mathcal{PIF}$ always satisfies the **start** property of Specification 1.*

Still considering an arbitrary initial configuration, we now state the **user-safety** property (Corollary 2), that is, during any PIF of some message m started by p:

(U1) Every process other than p receives m.
(U2) p receives acknowledgments for m from all other processes.
(U3) p executes the decision event[5] in finite time, this decision is taken following the acknowledgments for m.

We first show (U1) and (U2) in Lemma 5.
The next technical lemma is used in the proof of Lemma 5.

[5] Remember that the decision event corresponds to the statement $\texttt{Req}_p \leftarrow \texttt{Done}$.

Algorithm 1 Protocol $\mathcal{PIF}$ for any process p

```
Constant: $n$: integer

Variables:
  $\texttt{Req}_p \in \{\texttt{Wait}, \texttt{In}, \texttt{Done}\}$                                     : input/output
  $\texttt{BMes}_p, \texttt{FMes}_p[1 \dots n-1]$                                      : inputs
  $\texttt{State}_p[1 \dots n-1] \in \{0,1,2,3,4\}^{n-1}, \texttt{NState}_p[1 \dots n-1] \in \{0,1,2,3,4\}^{n-1}$ : internals

Actions:
  $A_1$ :: $(\texttt{Req}_p = \texttt{Wait})$        → $\texttt{Req}_p \leftarrow \texttt{In}$ /* Start */
                                       for all $q \in [1 \dots n-1]$ do $\texttt{State}_p[q] \leftarrow 0$

  $A_2$ :: $(\texttt{Req}_p = \texttt{In})$          → if $(\forall q \in [1 \dots n-1], \texttt{State}_p[q] = 4)$ then
                                           $\texttt{Req}_p \leftarrow \texttt{Done}$ /* Termination */
                                       else
                                           for all $q \in [1 \dots n-1]$ do
                                               if $(\texttt{State}_p[q] < 4)$ then
                                                   send$\langle \texttt{PIF}, \texttt{BMes}_p, \texttt{FMes}_p[q], \texttt{State}_p[q], \texttt{NState}_p[q] \rangle$ to $q$

  $A_3$ :: receive$\langle \texttt{PIF}, B, F, qState, pState \rangle$ → if $(\texttt{NState}_p[q] \neq 3) \wedge (qState = 3)$ then
          from channel $q$                     generate a "B-receive$\langle B \rangle$ from channel $q$" event
                                       $\texttt{NState}_p[q] \leftarrow qState$
                                       if $(\texttt{State}_p[q] = pState) \wedge (\texttt{State}_p[q] < 4)$ then
                                           $\texttt{State}_p[q] \leftarrow \texttt{State}_p[q] + 1$
                                           if $(\texttt{State}_p[q] = 4)$ then
                                               generate a "F-receive$\langle F \rangle$ from channel $q$" event
                                       if $(qState < 4)$ then
                                           send$\langle \texttt{PIF}, \texttt{BMes}_p, \texttt{FMes}_p[q], \texttt{State}_p[q], \texttt{NState}_p[q] \rangle$ to $q$
```

Lemma 4. *Let p and q be two distinct processes. After p starts a PIF (action $\texttt{A}_1$), p switches $\texttt{State}_p[q]$ from 2 to 3 only if the three following conditions hold:*

1. *Any message in the channel from p to q is of the form $\langle \texttt{PIF}, -, -, i, - \rangle$ with $i \neq 3$.*
2. *$\texttt{NState}_q[p] \neq 3$.*
3. *Any message in the channel from q to p is of the form $\langle \texttt{PIF}, -, -, -, j \rangle$ with $j \neq 3$.*

Proof. p starts a PIF with action $\texttt{A}_1$. By executing $\texttt{A}_1$, p sets $\texttt{State}_p[q]$ to 0. From that point, $\texttt{State}_p[q]$ can only be incremented one by one until reaching value 4. Let us study the three first increments of $\texttt{State}_p[q]$:

- **From 0 to 1.** $\texttt{State}_p[q]$ switches from 0 to 1 only after p receives a message $\langle \texttt{PIF}, -, -, -, 0 \rangle$ from q. As the link $\{p, q\}$ always contains at most one message in the channel from q to p, the next message that p will receive from q will be a message sent by q.
- **From 1 to 2.** From the previous case, we know that $\texttt{State}_p[q]$ switches from 1 to 2 only when p receives $\langle \texttt{PIF}, -, -, -, 1 \rangle$ from q and this message was sent by q. From actions $\texttt{A}_2$ and $\texttt{A}_3$, we can then deduce that $\texttt{NState}_q[p] = 1$ held when q sent $\langle \texttt{PIF}, -, -, -, 1 \rangle$ to p. From that point, $\texttt{NState}_q[p] = 1$ holds until q receives from p a message of the form $\langle \texttt{PIF}, -, -, i, - \rangle$ with $i \neq 1$.
- **From 2 to 3.** The switching of $\texttt{State}_p[q]$ from 2 to 3 can occurs only after p receives a message $mes_1 = \langle \texttt{PIF}, -, -, -, 2 \rangle$ from q. Now, from the previous case, we can deduce that p receives mes_1 consequently to the reception by q of a message $mes_0 = \langle \texttt{PIF}, -, -, 2, - \rangle$ from p. Now:
  (a) As the link $\{p, q\}$ always contains at most one message in the channel from p to q, after receiving mes_0 and until $\texttt{State}_p[q]$ switches from 2 to 3, every message in transit from p to q is of the form $\langle \texttt{PIF}, -, -, i, - \rangle$ with $i \neq 3$ (Condition 1 of the lemma) because after p starts to broadcast a message, p sends messages of the form $\langle \texttt{PIF}, -, -, 3, - \rangle$ to q only when $\texttt{State}_p[q] = 3$.
  (b) After receiving mes_0, $\texttt{NState}_q[p] \neq 3$ until q receives $\langle \texttt{PIF}, -, -, 3, - \rangle$. Hence, by (a), after receiving mes_0 and until (at least) $\texttt{State}_p[q]$ switches from 2 to 3, $\texttt{NState}_q[p] \neq 3$ (Condition 2 of the lemma).
  (c) After receiving mes_1, $\texttt{State}_p[q] \neq 3$ until p receives $\langle \texttt{PIF}, -, -, -, 3 \rangle$ from q. As p receives mes_1 after q receives mes_0, by (b) we can deduce that after receiving mes_1 and until (at least) $\texttt{State}_p[q]$ switches from 2 to 3, every message in transit from q to p is of the form $\langle \texttt{PIF}, -, -, -, j \rangle$ with $j \neq 3$ (Condition 3 of the lemma).

Hence, when p switches $\texttt{State}_p[q]$ from 2 to 3, the three conditions (1), (2) and (3) are satisfied, which proves the lemma. □

Lemma 5. *Starting from any configuration, if p starts a PIF of some message m (action $\texttt{A}_1$), then:*

- *All other process eventually receive m.*
- *p eventually receives acknowledgments for m from all other processes.*

Proof. p starts a PIF of m by executing action $\texttt{A}_1$: p switches $\texttt{Req}_p$ from $\texttt{Wait}$ to $\texttt{In}$ and sets $\texttt{State}_p[q]$ to 0, for every index q. Then, $\texttt{Req}_p$ remains equal to $\texttt{In}$ until p decides by setting $\texttt{Req}_p$ to $\texttt{Done}$. Now, p decides in finite time by Lemma 3 and when p decides, we have $\texttt{State}_p[q] = 4, \forall q \in [1 \dots 0]$ (action $\texttt{A}_2$). From the code of Algorithm 1, this means that for every index q, $\texttt{State}_p[q]$ is incremented one by one from 0 to 4. By Lemma 4, for every index q, $\texttt{State}_p[q]$ is incremented from 3 to 4 only after:

- q receives a message sent by p of the form $\langle \texttt{PIF}, m, -, 3, - \rangle$, and then
- p receives a message sent by q of the form $\langle \texttt{PIF}, -, -, 3, - \rangle$.

When q receives $\langle \texttt{PIF}, m, -, 3, - \rangle$ from p for the first time, it generates the event "**B-receive** $\langle m \rangle$ **from** channel p" and then starts to send $\langle \texttt{PIF}, -, F, -, 3 \rangle$ messages to p.[6] From that point and until p decides, q only receives $\langle \texttt{PIF}, m, -, 3, - \rangle$ message from p. So, from that point and until p decides, any message that q sends to p acknowledges the reception of m. Since, p receives the first $\langle \texttt{PIF}, -, F, -, 3 \rangle$ message from q, p generates a "**F-receive** $\langle F \rangle$ **from** channel q" event and then sets $\texttt{State}_p[q]$ to 4.

[6] q sends a $\langle \texttt{PIF}, -, F, -, 3 \rangle$ message to p (at least) each time it receives a $\langle \texttt{PIF}, m, -, 3, - \rangle$ message from p.

Hence, for every process q, the broadcast of m generates a "**B-receive** $\langle m \rangle$ **from** channel p" event at q and the associated "**F-receive** $\langle F \rangle$ **from** channel q" event at p, which proves the lemma. □

The next lemma proves (U3).

Lemma 6. *Starting from any configuration, during any PIF of some message m started by p, (1) p executes a decision event (i.e.,* $\texttt{Req}_p \leftarrow \texttt{Done}$*) in finite time and (2) this decision is taken following the acknowledgments for m.*

Proof. First, during any PIF of some message m started by p, p decides in finite time by Lemma 3.

Let q be a process such that $q \neq p$. We now show the second part of the lemma by proving that *between the start of the PIF of m and the corresponding decision, p generates exactly one* "**F-receive** $\langle F \rangle$ **from** channel q" *event where F is an acknowledgment sent by q for m.*

First p starts a PIF of m by executing action $\texttt{A}_1$: p switches $\texttt{Req}_p$ from $\texttt{Wait}$ to $\texttt{In}$ and sets $\texttt{State}_p[q]$ to 0. Then, $\texttt{Req}_p$ remains equal to $\texttt{In}$ until p decides by setting $\texttt{Req}_p$ to $\texttt{Done}$. When p decides, we have $\texttt{State}_p[q] = 4$, for every index q. From the code of Algorithm 1, we know that exactly one "**F-receive** $\langle F \rangle$ **from** channel q" event occurs at p before p decides: when p switches $\texttt{State}_p[q]$ from 3 to 4. Lemma 4 implies that F is an acknowledgment for m sent by q and the lemma is proven. □

By Lemmas 5 and 6, follows:

Corollary 2 (*User-Safety*)**.** *Starting from any configuration, $\mathcal{PIF}$ always satisfies the **User-Safety** property of Specification 1.*

By Corollaries 1 and 2, follows:

Theorem 2. *$\mathcal{PIF}$ is snap-stabilizing to Specification 1.*

Below, we give an additional property of $\mathcal{PIF}$, this property will be used in the snap-stabilization proof of $\mathcal{ME}$ (Section 4.2).

Property 1. *If p starts a PIF in the configuration γ_0 and the PIF terminates at p in the configuration γ_k, then any message that was in a channel from and to p in γ_0 is no longer in the channel in γ_k.*

Proof. Assume that a process p starts a PIF in the configuration γ_0. Then, as $\mathcal{PIF}$ is snap-stabilizing to Specification 1, we have the guarantee that for every p'neighbor q, at least one broadcast message crosses the channel from p to q and at least one acknowledgment message crosses the channel from q to p during the PIF-computation. Now, we assumed that each channel has a single-message capacity. Hence, every message that was in a channel from and to p in the configuration γ_0 has been received or lost when the PIF terminates at p in configuration γ_k. □

4.2. Mutual exclusion for fully-connected networks

4.2.1. Specification

We now consider the problem of *mutual exclusion*. The mutual exclusion specification requires that a special section of code, called the *critical section*, is executed by at most one process at any time. A snap-stabilizing mutual exclusion protocol (only) guarantees its safety property when the process requests the critical section after the faults stop [11]. The safety property is not otherwise guaranteed. Hence, we specify the mutual exclusion protocol as follows:

Specification 2 (*ME-Exec*)**.** *An execution e satisfies Predicate ME-Exec if and only if e satisfies the following two properties:*

1. **Start.** *Upon a request, a process enters the critical section in finite time.*
2. **User-Safety.** *If a requested process enters the critical section, then it executes the critical section alone.*

In order to simplify the design of our mutual exclusion algorithm, we propose below an identifier-discovery algorithm, $\mathcal{IDL}$, that is a straightforward extension of $\mathcal{PIF}$.

4.2.2. Identifier-discovery

$\mathcal{IDL}$ assumes that each process has a unique identifier (ID_p denotes the identifier of the process p) and uses three variables at each process p:

- $\texttt{Req}_p \in \{\texttt{Wait}, \texttt{In}, \texttt{Done}\}$. The purpose of this variable is the same as in $\mathcal{PIF}$.
- $\texttt{minID}_p$. After a complete computation of $\mathcal{IDL}$, $\texttt{minID}_p$ contains the minimal identifier of the system.
- $\texttt{IDTab}_p[1 \ldots n]$. After a complete computation of $\mathcal{IDL}$, $\texttt{IDTab}_p[q] = ID_q$.

When requested at p, $\mathcal{IDL}$ evaluates the identifiers of all other processes and the minimal identifier of the system using $\mathcal{PIF}$. The results of the computation are available for p since p decides. Based on the specification of $\mathcal{PIF}$, it is easy to see that $\mathcal{IDL}$ is snap-stabilizing to the following specification:

Specification 3 (*ID-Discovery-Exec*)**.** *An execution e satisfies Predicate ID-Discovery-Exec if and only if e satisfies the following two properties:*

1. **Start.** *When requested, a process p starts in finite time to discover the identifiers of the processes.*
2. **User-Safety.** *Any identifier-discovery started by p terminates in finite time by a decision event at p and when the decision occurs, we have:*
   - $\forall q \in [1 \ldots n-1]$, $\texttt{IDTab}_p[q] = ID_q$.
   - $\texttt{minID}_p = \min(\{ID_q, q \in [1 \ldots n-1]\} \cup \{ID_p\})$.

Theorem 3. *$\mathcal{IDL}$ is snap-stabilizing for Specification 3.*

4.2.3. Principle

We now describe a snap-stabilizing mutual exclusion protocol called $\mathcal{ME}$ (Algorithm 3). $\mathcal{ME}$ also uses the variable $\texttt{Req}$ with the same meaning as previously: $\mathcal{ME}.\texttt{Req}_p$ is to $\texttt{Wait}$ when the process p is requested to execute the critical section. Process p is then called a *requestor* and we assume that $\mathcal{ME}.\texttt{Req}_p$ cannot be set to $\texttt{Wait}$ again until $\mathcal{ME}.\texttt{Req}_p = \texttt{Done}$, i.e., until its current request is done.

The main idea behind the protocol is the following: we assume identifiers on processes and the process with the smallest identifier – called the *leader* – bounces a single token to every process using a variable called $\texttt{Val}$, this variable ranges over $\{0 \ldots n-1\}$. The $\texttt{Val}$ variable of the leader $\mathcal{L}$ designates which process holds the token: process p holds the token if and only if either $p = \mathcal{L}$ and $\texttt{Val}_{\mathcal{L}} = 0$ or $p \neq \mathcal{L}$ and $\texttt{Val}_{\mathcal{L}}$ is equal to the number of its channel incoming from p. A process can access the critical section only if it holds the token. Thus, the processes continuously ask the leader to know if they hold the token.

When a process learns that it holds the token:

(1) It first ensures that no other process can execute the critical section (due to the arbitrary initial configuration, some other processes may wrongly believe that they also hold the token).
(2) It then executes the critical section if it wishes to (it may refuse if it is not a requestor).
(3) Finally, it notifies to the leader that it has terminated Step (2) so that the leader passes the token to another process.

To apply this scheme, $\mathcal{ME}$ is executed in phases from Phase 0 to 4 in such way that each process goes through Phase 0 infinitely often. After a request, a process p can access the critical section only after executing Phase 0: indeed p can access the critical section only if $\mathcal{ME}.\texttt{Req}_p = \texttt{In}$ and p switches $\mathcal{ME}.\texttt{Req}_p$ from $\texttt{Wait}$ to $\texttt{In}$ only in Phase 0. Hence, our protocol just ensures that after executing Phase

Algorithm 2 Protocol $\mathcal{IDL}$ for any process p

```
Constants: n, ID_p: integers

Variables:
  Req_p ∈ {Wait, In, Done}                 :  input/output
  minID_p ∈ ℕ, IDTab_p[1...n − 1] ∈ ℕ^{n−1} :  outputs

Actions:
  A_1 :: (Req_p = Wait)                        →  Req_p ← In /* Start */
                                                  minID_p ← ID_p; PIF.BMes_p ← IDL
                                                  PIF.Req_p ← Wait

  A_2 :: (Req_p = In) ∧ (PIF.Req_p = Done)     →  Req_p ← Done /* Termination */

  A_3 :: B-receive⟨IDL⟩ from channel q         →  PIF.FMes_p[q] ← ID_p

  A_4 :: F-receive⟨ID⟩ from channel q          →  IDTab_p[q] ← ID; minID_p ← min(minID_p, ID)
```

0, a process always executes the critical section alone. Below, we describe the five phases of our protocol:

Phase 0. When a process p is in Phase 0, it requests a computation of $\mathcal{IDL}$ to collect the identifiers of all processes and to evaluate which one is the leader. It also sets $\mathcal{ME}.\mathtt{Req}_p$ to In if $\mathcal{ME}.\mathtt{Req}_p = \mathtt{Wait}$. It then switches to Phase 1.

Phase 1. When a process p is in Phase 1, p waits for the termination of $\mathcal{IDL}$. Then, p requests a PIF of the message ASK to know if it is the token holder and switches to Phase 2. Upon receiving a message ASK from the channel p, any process q answers YES if $\mathtt{Val}_q = p$, NO otherwise. Of course, p will only take the answer of the leader into account.

Phase 2. When a process p is in Phase 2, it waits for the termination of the PIF requested in Phase 1. After the PIF terminates, p knows if it is the token holder. If p holds the token, it requests a PIF of the message EXIT and switches to Phase 3. The goal of this message is to force all other processes to restart to Phase 0. This ensures that no other process believes to be the token holder when p accesses the critical section. Indeed, due to the arbitrary initial configuration, some process $q \neq p$ may believe to be the token holder, if q never starts Phase 0. On the contrary, after restarting to 0, q cannot receive positive answer from the leader until p notifies to the leader that it releases the critical section.

Phase 3. When a process p is in Phase 3, it waits for the termination of the current PIF. After the PIF terminates, if p is the token holder, then:

1) p executes the critical section and switches $\mathcal{ME}.\mathtt{Req}_p$ from In to Done if $\mathcal{ME}.\mathtt{Req}_p = \mathtt{In}$, and then
2) (a) Either, p is the leader and switches $\mathtt{Val}_p$ from 0 to 1.
   (b) Or, p is not the leader and requests a PIF of the message EXITCS to notify to the leader that it releases the critical section. Upon receiving such a message, the leader increments its variable Val modulus $n+1$ to pass the token to another process.

In any case, p terminates Phase 3 by switching to Phase 4.

Phase 4. When a process p is in Phase 4, it waits for the termination of the last PIF and then switches to Phase 0.

4.2.4. Correctness

We begin the proof of snap-stabilization of $\mathcal{ME}$ by showing that, despite the arbitrary initial configuration, any execution of $\mathcal{ME}$ always satisfies the **user-safety** property of Specification 2.

Assume that a process p is a requestor, i.e., $\mathcal{ME}.\mathtt{Req}_p = \mathtt{Wait}$. Then, p cannot enters the critical section before executing action $\mathtt{A}_0$. Indeed:

- p enters the critical section only if $\mathcal{ME}.\mathtt{Req}_p = \mathtt{In}$, and
- action $\mathtt{A}_0$ is the only action of $\mathcal{ME}$ allowing p to set $\mathcal{ME}.\mathtt{Req}_p$ to In.

Hence, to show the **user-safety** property of Specification 2 (Corollary 3), we have to prove that, despite the initial configuration, after p executes action $\mathtt{A}_0$, if p enters the critical section, then it executes the critical section alone (Lemma 9).

Lemma 7. *Let p be a process. Starting from any configuration, after p executes $\mathtt{A}_0$, if p enters the critical section, then all other processes have switched to Phase 0 at least once.*

Proof. After p executes $\mathtt{A}_0$, to enter the critical section (in $\mathtt{A}_3$) p must execute the three actions $\mathtt{A}_1$, $\mathtt{A}_2$, and $\mathtt{A}_3$ successively. Also, to execute the critical section in action $\mathtt{A}_3$, p must satisfy the predicate $Winner(p)$. The value of the predicate $Winner(p)$ depends on (1) the $\mathcal{IDL}$ computation requested in $\mathtt{A}_0$ and (2) the PIF of the message ASK requested in $\mathtt{A}_1$. Now, these two computations are done when p executes $\mathtt{A}_2$. So, the fact that p satisfies $Winner(p)$ when executing $\mathtt{A}_3$ implies that p also satisfies $Winner(p)$ when executing $\mathtt{A}_2$. As a consequence, p requests a PIF of the message EXIT in $\mathtt{A}_2$. Now, p executes $\mathtt{A}_3$ only after this PIF terminates. Hence, p executes $\mathtt{A}_3$ only after every other process executes $\mathtt{A}_6$ (i.e., the feedback of the message EXIT): by this action, every other process switches to Phase 0. □

Definition 6 (*Leader*)**.** We call *Leader* the process with the smallest identifier. In the following, this process will be denoted by $\mathcal{L}$.

Definition 7 (*Favour*)**.** We say that the process p *favours* the process q if and only if $(p = q \wedge \mathtt{Val}_p = 0) \vee (p \neq q \wedge \mathtt{Val}_p = q)$.

Lemma 8. *Let p be a process. Starting from any configuration, after p executes $\mathtt{A}_0$, p enters the critical section only if $\mathcal{L}$ favours p until p releases the critical section.*

Proof. By checking all the actions of Algorithm 3, we can observe that after p executes $\mathtt{A}_0$, to enter the critical section p must execute the four actions $\mathtt{A}_0$, $\mathtt{A}_1$, $\mathtt{A}_2$, and $\mathtt{A}_3$ successively. Moreover, p executes a complete $\mathcal{IDL}$-computation between $\mathtt{A}_0$ and $\mathtt{A}_1$. Thus:

(1) $\mathcal{IDL}.\mathtt{minID}_p = ID_{\mathcal{L}}$ when p executes $\mathtt{A}_3$.
(2) Also, from the configuration where p executes $\mathtt{A}_1$, all messages in the channels from and to p have been sent after p requests $\mathcal{IDL}$ in action $\mathtt{A}_0$ (Property 1, page 17).

Let us now study the following two cases:

- $p = \mathcal{L}$. In this case, when p executes $\mathtt{A}_3$, to enter the critical section p must satisfy $\mathtt{Val}_p = \mathtt{Val}_{\mathcal{L}} = 0$ by (1). This means that $\mathcal{L}$ favours p (actually itself) when p enters the critical section. Moreover, as the execution of $\mathtt{A}_3$ is atomic, $\mathcal{L}$ favours p until p releases the critical section and this closes the case.

Algorithm 3 Protocol $\mathcal{ME}$ for any process p

```
Constants: $n$, $ID_p$: integers

Variables:
  $\texttt{Req}_p \in \{\texttt{Wait}, \texttt{In}, \texttt{Done}\}$   :   input/output
  $\texttt{Ph}_p \in \{0,1,2,3,4\}$, $\texttt{Val}_p \in \{0 \ldots n-1\}$, $\texttt{Answers}_p[1 \ldots n-1] \in \{true, false\}^{n-1}$   :   internals

Predicate:
  $Winner(p) \equiv (\mathcal{IDL}.\texttt{minID}_p = ID_p \wedge \texttt{Val}_p = 0) \vee (\exists q \in [1 \ldots n-1], \texttt{Answers}_p[q] \wedge \mathcal{IDL}.\texttt{IDTab}_p[q] = \mathcal{IDL}.\texttt{minID}_p)$

Actions:
  $A_0$ :: $(\texttt{Ph}_p = 0)$ → 
      $\mathcal{IDL}.\texttt{Req}_p \leftarrow \texttt{Wait}$
      if $\texttt{Req}_p = \texttt{Wait}$ then $\texttt{Req}_p \leftarrow \texttt{In}$ /* Start */
      $\texttt{Ph}_p \leftarrow \texttt{Ph}_p + 1$

  $A_1$ :: $(\texttt{Ph}_p = 1) \wedge (\mathcal{IDL}.\texttt{Req}_p = \texttt{Done})$ →
      $\mathcal{PIF}.\texttt{BMes}_p \leftarrow \texttt{ASK}$; $\mathcal{PIF}.\texttt{Req}_p \leftarrow \texttt{Wait}$
      $\texttt{Ph}_p \leftarrow \texttt{Ph}_p + 1$

  $A_2$ :: $(\texttt{Ph}_p = 2) \wedge (\mathcal{PIF}.\texttt{Req}_p = \texttt{Done})$ →
      if $Winner(p)$ then
          $\mathcal{PIF}.\texttt{BMes}_p \leftarrow \texttt{EXIT}$; $\mathcal{PIF}.\texttt{Req}_p \leftarrow \texttt{Wait}$
      $\texttt{Ph}_p \leftarrow \texttt{Ph}_p + 1$

  $A_3$ :: $(\texttt{Ph}_p = 3) \wedge (\mathcal{PIF}.\texttt{Req}_p = \texttt{Done})$ →
      if $Winner(p)$ then
          if $\texttt{Req}_p = \texttt{In}$ then
              critical section; $\texttt{Req}_p \leftarrow \texttt{Done}$ /* Termination */
          if $\mathcal{IDL}.\texttt{minID}_p = ID_p$ then
              $\texttt{Val}_p \leftarrow 1$
          else
              $\mathcal{PIF}.\texttt{BMes}_p \leftarrow \texttt{EXITCS}$; $\mathcal{PIF}.\texttt{Req}_p \leftarrow \texttt{Wait}$
      $\texttt{Ph}_p \leftarrow \texttt{Ph}_p + 1$

  $A_4$ :: $(\texttt{Ph}_p = 4) \wedge (\mathcal{PIF}.\texttt{Req}_p = \texttt{Done})$ →
      $\texttt{Ph}_p \leftarrow 0$

  $A_5$ :: B-receive⟨ASK⟩ from channel $q$ →
      if $\texttt{Val}_p = q$ then
          $\mathcal{PIF}.\texttt{FMes}_p[q] \leftarrow \texttt{YES}$
      else
          $\mathcal{PIF}.\texttt{FMes}_p[q] \leftarrow \texttt{NO}$

  $A_6$ :: B-receive⟨EXIT⟩ from channel $q$ →
      $\texttt{Ph}_p \leftarrow 0$; $\mathcal{PIF}.\texttt{FMes}_p[q] \leftarrow \texttt{OK}$

  $A_7$ :: B-receive⟨EXITCS⟩ from channel $q$ →
      if $\texttt{Val}_p = q$ then $\texttt{Val}_p \leftarrow (\texttt{Val}_p + 1) \bmod (n+1)$
      $\mathcal{PIF}.\texttt{FMes}_p[q] \leftarrow \texttt{OK}$

  $A_8$ :: F-receive⟨YES⟩ from channel $q$ →
      $\texttt{Answers}_p[q] \leftarrow true$

  $A_9$ :: F-receive⟨NO⟩ from channel $q$ →
      $\texttt{Answers}_p[q] \leftarrow false$

  $A_{10}$ :: F-receive⟨OK⟩ from channel $q$ →
      /* do nothing */
```

- $p \neq \mathcal{L}$. In this case, when p executes A_3, p satisfies $\mathcal{IDL}.\texttt{minID}_p = ID_{\mathcal{L}}$ by (1). So, p executes the critical section only if $\exists q \in [1 \ldots n-1]$ such that $\mathcal{IDL}.\texttt{IDTab}_p[q] = ID_{\mathcal{L}} \wedge \texttt{Answers}_p[q] = true$ (see Predicate $Winner(p)$). For that, p must receive a feedback message YES from $\mathcal{L}$ during the PIF of the message ASK requested in action A_1. Now, $\mathcal{L}$ sends such a feedback to p only if $\texttt{Val}_{\mathcal{L}} = p$ when the "**B-receive**⟨ASK⟩ **from** p" event occurs at $\mathcal{L}$ (see action A_5). Also, since $\mathcal{L}$ satisfies $\texttt{Val}_{\mathcal{L}} = p$, $\mathcal{L}$ updates $\texttt{Val}_p$ only after receiving an EXITCS message from p (see action A_7). Now, by (2), after $\mathcal{L}$ feedbacks YES to p, $\mathcal{L}$ receives an EXITCS message from p only if p broadcasts EXITCS to $\mathcal{L}$ after releasing the critical section (see action A_3). Hence, $\mathcal{L}$ favours p until p releases the critical section and this closes the case. □

Lemma 9. *Let p be a process. Starting from any configuration, if p enters the critical section after executing A_0, then it executes the critical section alone.*

Proof. Assume, for the sake of contradiction, that p enters the critical section after executing A_0 but executes the critical section concurrently with another process q. Then, q also executes action A_0 before executing the critical section by Lemma 7. By Lemma 8, we have the following two property:

- $\mathcal{L}$ favours p during the whole period where p executes the critical section.
- $\mathcal{L}$ favours q during the whole period where q executes the critical section.

This contradicts the fact that p and q executes the critical section concurrently because $\mathcal{L}$ always favours exactly one process at a time. □

Corollary 3 (*User-Safety*)**.** *Starting from any configuration, $\mathcal{ME}$ always satisfies the* **user-safety** *property of Specification 2.*

We now show that, despite the arbitrary initial configuration, any execution of $\mathcal{ME}$ always satisfies the **start** property of Specification 2 (Lemma 4). As previously, this proof is made in two steps:

(S1) We first prove that each time a user want to execute the critical section at some process p, then it is eventually able to submit its request to the process (i.e. it is eventually enabled to execute $\mathcal{ME}.\texttt{Req}_p \leftarrow \texttt{Wait}$).

(S2) We then prove that once a request has been submitted to some process p, the process enters the critical section in finite time.

To prevent the aborting of the previous request, a user can submit a request at some process p only if $\mathcal{ME}.\texttt{Req}_p = \texttt{Done}$. Hence, to show (S1), we show that from any configuration where $\mathcal{ME}.\texttt{Req}_p \in \{\texttt{Wait}, \texttt{In}\}$, the system eventually reaches a configuration where $\mathcal{ME}.\mathtt{Req}_p = \mathtt{Done}$. This latter claim is proven in two stages:

- We first show in Lemma 11 that from any configuration where $\mathcal{ME}.\mathtt{Req}_p = \mathtt{Wait}$, in finite time the system reaches a configuration where $\mathcal{ME}.\mathtt{Req}_p = \mathtt{In}$.
- We then show in Lemma 13 that from any configuration where $\mathcal{ME}.\mathtt{Req}_p = \mathtt{In}$, in finite time the system reaches a configuration where $\mathtt{Req}_p = \mathtt{Done}$.

The next technical lemma is used in the proof of Lemma 11.

Lemma 10. *Starting from any configuration, every process p switches to Phase* 0 *infinitely often.*

Proof. Consider the following two cases:

- "**B-receive**$\langle\mathtt{EXIT}\rangle$" events occur at p infinitely often. Then, each time such an event occurs at p, p switches to Phase 0 (see $\mathtt{A}_6$) and this closes the case.
- *Only a finite number of* "**B-receive**$\langle\mathtt{EXIT}\rangle$" *events occurs at p*. In this case, p eventually reaches a configuration from which it no longer executes action $\mathtt{A}_6$. From this configuration, $\mathtt{Ph}_p$ can only be incremented modulus 5 and depending of the value of $\mathtt{Ph}_p$, we have the following possibilities:
  - $\mathtt{Ph}_p = 0$. In this case, $\mathtt{A}_0$ is continuously enabled at p. Hence, p eventually sets $\mathtt{Ph}_p$ to 1 (see action $\mathtt{A}_0$).
  - $\mathtt{Ph}_p = i$ with $0 < i \leq 4$. In this case, action $\mathtt{A}_i$ is eventually continuously enabled due to the termination property of $\mathcal{IDL}$ and $\mathcal{PIF}$. By executing $\mathtt{A}_i$, p increments $\mathtt{Ph}_p$ modulus 5.

  Hence, if only a finite number of "**B-receive**$\langle\mathtt{EXIT}\rangle$" events occurs at p, then $\mathtt{Ph}_p$ is incremented modulus 5 infinitely often and this closes the case. □

Lemma 11. *Let p be any process. From any configuration where* $\mathcal{ME}.\mathtt{Req}_p = \mathtt{Wait}$, *in finite time the system reaches a configuration where* $\mathcal{ME}.\mathtt{Req}_p = \mathtt{In}$.

Proof. Assume that $\mathcal{ME}.\mathtt{Req}_p = \mathtt{Wait}$. Lemma 10 implies that p eventually executes action $\mathtt{A}_0$. By action $\mathtt{A}_0$, $\mathcal{ME}.\mathtt{Req}_p$ is set to In. □

The next technical lemma is used in the proof of Lemma 13.

Lemma 12. *Starting from any configuration,* $\mathtt{Val}_{\mathcal{L}}$ *is incremented modulus* $n+1$ *infinitely often.*

Proof. Assume, for the sake of contradiction, that there are a finite number of increments of $\mathtt{Val}_{\mathcal{L}}$ (modulus $n+1$). We can then deduce that $\mathcal{L}$ eventually favours some process p forever.

In order to prove the contradiction, we first show that (*) *assuming that* $\mathcal{L}$ *favours p forever, only a finite number of* "**B-receive**$\langle\mathtt{EXIT}\rangle$" *events occurs at p*. Towards this end, assume, for the sake of contradiction, that an infinite number of "**B-receive**$\langle\mathtt{EXIT}\rangle$" events occurs at p. Then, as the number of processes is finite, there is a process $q \neq p$ that broadcasts EXIT messages infinitely often. Now, every PIF terminates in finite time. So, q performs infinitely many PIF of the message EXIT. In order to start another PIF of the message EXIT, q must then successively execute actions $\mathtt{A}_0, \mathtt{A}_1, \mathtt{A}_2$. Now, when q executes $\mathtt{A}_2$ after $\mathtt{A}_0$ and $\mathtt{A}_1$, $\mathcal{IDL}.\mathtt{minID}_q = ID_{\mathcal{L}}$ and either (1) $q = \mathcal{L}$ and, as $q \neq p$, $\mathtt{Val}_{\mathcal{L}} \neq 0$, or (2) $\mathcal{L}$ has feedback NO to the PIF of the message ASK started by q because $\mathtt{Val}_{\mathcal{L}} = p \neq q$. In both cases, q satisfies $\neg Winner(q)$ and, as a consequence, does not broadcast EXIT (see action $\mathtt{A}_3$). Hence, q eventually stops to broadcast EXIT — a contradiction.

Using Property (∗), we now show the contradiction. By Lemma 10, p switches to Phase 0 infinitely often. By (∗), we know that p eventually stops executing action $\mathtt{A}_6$. So, from the code of Algorithm 3, we can deduce that p eventually successively executes actions $\mathtt{A}_0$, $\mathtt{A}_1$, $\mathtt{A}_2$, $\mathtt{A}_3$, and $\mathtt{A}_4$ infinitely often. Consider the first time p successively executes $\mathtt{A}_0$, $\mathtt{A}_1$, $\mathtt{A}_2$, $\mathtt{A}_3$, and $\mathtt{A}_4$ and study the following two cases:

- $p = \mathcal{L}$. Then, $\mathtt{Val}_p = 0$ and $\mathcal{IDL}.\mathtt{minID}_p = ID_p$ when p executes $\mathtt{A}_3$ because p executes a complete $\mathcal{IDL}$-computation between $\mathtt{A}_0$ and $\mathtt{A}_1$ and $\mathcal{IDL}$ is snap-stabilizing to Specification 3. Hence, p updates $\mathtt{Val}_p$ to 1 when executing $\mathtt{A}_3$ — a contradiction.
- $p \neq \mathcal{L}$. Then, $\mathcal{IDL}.\mathtt{minID}_p = ID_{\mathcal{L}}$ when p executes $\mathtt{A}_3$ because p executes a complete $\mathcal{IDL}$-computation between $\mathtt{A}_0$ and $\mathtt{A}_1$ and $\mathcal{IDL}$ is snap-stabilizing to Specification 3. Also, p receives YES from $\mathcal{L}$ because p executes a complete PIF of the message ASK between $\mathtt{A}_1$ and $\mathtt{A}_2$ and $\mathcal{PIF}$ is snap-stabilizing to Specification 1. Hence, p satisfies the predicate $Winner(p)$ when executing $\mathtt{A}_3$ and, as a consequence, requests a PIF of the message EXITCS in action $\mathtt{A}_3$. This PIF terminates when p executes $\mathtt{A}_4$: from this point on, we have the guarantee that $\mathcal{L}$ has executed action $\mathtt{A}_7$. Now, by $\mathtt{A}_7$, $\mathcal{L}$ increments $\mathtt{Val}_{\mathcal{L}}$ — a contradiction. □

Lemma 13. *Let p be any process. From any configuration where* $\mathcal{ME}.\mathtt{Req}_p = \mathtt{In}$, *in finite time the system reaches a configuration where* $\mathcal{ME}.\mathtt{Req}_p = \mathtt{Done}$.

Proof. Assume, for the sake of contradiction, that from a configuration where $\mathtt{Req}_p = \mathtt{In}$ the system never reaches a configuration where $\mathtt{Req}_p = \mathtt{Done}$. From the code of Algorithm 3, we can then deduce that $\mathtt{Req}_p = \mathtt{In}$ holds forever. In this case there are two possibilities:

- p no longer executes $\mathtt{A}_3$, or
- p satisfies $\neg Winner(p)$ each time it executes $\mathtt{A}_3$.

Consider then the following two cases:

- $p = \mathcal{L}$. Then, $\mathtt{Val}_p \neq 0$ eventually holds forever — a contradiction to Lemma 12.
- $p \neq \mathcal{L}$. In this case, p no longer starts any PIF of the message EXITCS. Now, every PIF terminates in finite time. Hence, eventually there is no more "**B-receive**$\langle\mathtt{EXITCS}\rangle$ **from** p" event at $\mathcal{L}$. As a consequence, $\mathtt{Val}_{\mathcal{L}}$ eventually no longer switches from value p to $(p+1) \mod (n+1)$ — which contradicts Lemma 12. □

As explained before, Lemmas 11 and 13 proves (S1). Lemma 13 also implies (S2) because a process switches $\mathcal{ME}.\mathtt{Req}_p$ from In to Done only after executing the critical section. Hence, we have the following corollary:

Corollary 4 (*Start*)**.** *Starting from any configuration,* $\mathcal{ME}$ *always satisfies the* **start** *property of Specification* 2.

By Corollaries 3 and 4, follows:

Theorem 4. $\mathcal{ME}$ *is snap-stabilizing to Specification* 2.

5. Conclusion

We addressed the problem of *snap-stabilization* in one-hop message-passing systems and presented matching negative and positive results. On the negative side, we showed that *snap-stabilization* is impossible for a wide class of specifications – namely, the *safety-distributed* specifications – in message-passing systems where the channel capacity is finite yet unbounded. On the positive side, we showed that *snap-stabilization* is possible (even for *safety-distributed* specifications) in message-passing systems if we assume a bound on the channel capacity. The proof is constructive, as we presented the first three snap-stabilizing protocols for message-passing systems with a bounded channel capacity. These protocols respectively solve the PIF and mutual exclusion problem in a fully-connected network.

On the theoretical side, it is worth observing that the results presented in this paper can be extended to general topologies using the approach presented in [21], and then to general specifications that admit a Katz and Perry transformer [26]. Yet, the possible extension to networks where nodes are subject to permanent, i.e., crash faults, remains open. On the practical side, our results imply the possibility of implementing snap-stabilizing protocols on real networks. Actually implementing them is a future challenge.

References

[1] Y. Afek, A. Bremler-Barr, Self-stabilizing unidirectional network algorithms by power supply, Chicago J. Theor. Comput. Sci. (1998).

[2] Y. Afek, G.M. Brown, Self-stabilization over unreliable communication media, Distributed Computing 7 (1) (1993) 27–34.

[3] B. Alpern, F.B. Schneider, Recognizing safety and liveness, Distributed Computing 2 (3) (1987) 117–126.

[4] A. Arora (Ed.), 1999 ICDCS Workshop on Self-stabilizing Systems, Austin, Texas, June 5, 1999, Proceedings, IEEE Computer Society, 1999.

[5] A. Arora, M. Nesterenko, Unifying stabilization and termination in message-passing systems, Distributed Computing 17 (3) (2005) 279–290. http://dx.doi.org/10.1007/s00446-004-0111-6.

[6] B. Awerbuch, S. Kutten, Y. Mansour, B. Patt-Shamir, G. Varghese, A time-optimal self-stabilizing synchronizer using a phase clock, IEEE Trans. Dependable Sec. Comput. 4 (3) (2007) 180–190.

[7] B. Awerbuch, B. Patt-Shamir, G. Varghese, Self-stabilization by local checking and correction (extended abstract), in: FOCS, IEEE, 1991, pp. 268–277.

[8] D. Bein, A.K. Datta, V. Villain, Snap-stabilizing optimal binary search tree, in: Herman and Tixeuil [24], pp. 1–17.

[9] A. Bui, A.K. Datta, F. Petit, V. Villain, State-optimal snap-stabilizing pif in tree networks, in: Arora [4], pp. 78–85.

[10] A. Bui, A.K. Datta, F. Petit, V. Villain, Snap-stabilization and pif in tree networks, Distributed Computing 20 (1) (2007) 3–19. http://dx.doi.org/10.1007/s00446-007-0030-4.

[11] A. Cournier, A.K. Datta, F. Petit, V. Villain, Enabling snap-stabilization, in: ICDCS, IEEE Computer Society, 2003, pp. 12–19.

[12] A. Cournier, S. Devismes, F. Petit, V. Villain, Snap-stabilizing depth-first search on arbitrary networks, Comput. J. 49 (3) (2006) 268–280.

[13] A. Cournier, S. Devismes, V. Villain, Snap-stabilizing pif and useless computations, in: ICPADS (1), IEEE Computer Society, 2006, pp. 39–48.

[14] A. Cournier, S. Devismes, V. Villain, A snap-stabilizing dfs with a lower space requirement, in: Herman and Tixeuil [24], pp. 33–47.

[15] A. Cournier, S. Devismes, V. Villain, Snap-stabilizing detection of cutsets, in: D.A. Bader, M. Parashar, S. Varadarajan, V.K. Prasanna (Eds.), HiPC, in: Lecture Notes in Computer Science, vol. 3769, Springer, 2005, pp. 488–497.

[16] S. Delaët, S. Devismes, M. Nesterenko, S. Tixeuil, Snap-stabilization in message-passing systems, in: International Conference on Distributed Systems and Networks, ICDCN 2009, in: LNCS, No. 5404, 2009, pp. 281–286. https://hal.inria.fr/inria-00248465.

[17] S. Delaët, B. Ducourthial, S. Tixeuil, Self-stabilization with r-operators revisited, Journal of Aerospace Computing, Information, and Communication (2006).

[18] E.W. Dijkstra, Self-stabilizing systems in spite of distributed control, Commun. ACM 17 (11) (1974) 643–644.

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

[20] S. Dolev, T. Herman, Superstabilizing protocols for dynamic distributed systems, Chicago J. Theor. Comput. Sci. (1997).

[21] S. Dolev, N. Tzachar, Empire of colonies: self-stabilizing and self-organizing distributed algorithms, in: A.A. Shvartsman (Ed.), OPODIS, in: Lecture Notes in Computer Science, vol. 4305, Springer, 2006, pp. 230–243.

[22] S. Ghosh, A. Bejan, A framework of safe stabilization, in: S.-T. Huang, T. Herman (Eds.), Self-Stabilizing Systems, in: Lecture Notes in Computer Science, vol. 2704, Springer, 2003, pp. 129–140.

[23] M.G. Gouda, N.J. Multari, Stabilizing communication protocols, IEEE Trans. Computers 40 (4) (1991) 448–458.

[24] T. Herman, S. Tixeuil (Eds.), Self-stabilizing systems, in: 7th International Symposium, SSS 2005, Barcelona, Spain, October 26–27, 2005, Proceedings, in: Lecture Notes in Computer Science, vol. 3764, Springer, 2005.

[25] R.R. Howell, M. Nesterenko, M. Mizuno, Finite-state self-stabilizing protocols in message-passing systems, in: Arora [4], pp. 62–69.

[26] S. Katz, K.J. Perry, Self-stabilizing extensions for message-passing systems., Distributed Computing 7 (1) (1993) 17–26.

[27] G. Tel, Introduction to Distributed Algorithms, second ed., Cambridge University Press, Cambridge, UK, 2001.

[28] S. Tixeuil, Self-stabilizing algorithms, in: Algorithms and Theory of Computation Handbook, second ed., in: Chapman & Hall/CRC Applied Algorithms and Data Structures, CRC Press, 2009, pp. 26.1–26.45. Taylor & Francis Group. http://www.crcpress.com/product/isbn/9781584888185.

[29] G. Varghese, Self-stabilization by counter flushing, SIAM J. Comput. 30 (2) (2000) 486–510.

[30] G. Varghese, M. Jayaram, The fault span of crash failures, J. ACM 47 (2) (2000) 244–293.

Sylvie Delaët is an associate professor at the University of Paris Sud-XI (France). She received her Ph.D. in 1995 from the University of Paris Sud-XI for her pioneering work on Self-stabilization. Her research interests include dynamic networks and systems with an emphasis on formal treatment.

Stéphane Devismes is an associate professor at the University Joseph Fourier of Grenoble (France). He is a member of the Synchronous Team of the VERIMAG Laboratory. He received his Ph.D. in 2006 from the University of Picardie Jules Verne (Amiens, France). He has carried out broad research in theoretical issues of distributed fault-tolerant computing, especially relating to self-stabilization.

Mikhail Nesterenko got his Ph.D. in 1998 from Kansas State University. Presently he is an associate professor at Kent State University. He is interested in wireless networking, distributed algorithms and fault-tolerance.

Sébastien Tixeuil is a full professor at the University Pierre & Marie Curie - Paris 6 (France), where he leads the NPA research group. He received his Ph.D. from University of Paris Sud-XI in 2000. His research interests include fault and attack tolerance in dynamic networks and systems.