Snap-Stabilization in Message-Passing Systems

Sylvie Delaët, Stéphane Devismes, Mikhail Nesterenko, Sébastien Tixeuil

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Snap-Stabilization in Message-Passing Systems

Sylvie Delaët — Stéphane Devismes — Mikhail Nesterenko — Sébastien Tixeuil

N° 9999

February 2008

Thème NUM

ISSN 0249-6399 ISRN INRIA/RR--9999--FR+ENG

Snap-Stabilization in Message-Passing Systems

Sylvie Delaët* , Stéphane Devismes† , Mikhail Nesterenko‡ , Sébastien Tixeuil§

Thème NUM — Systèmes numériques
Projet Grand large

Rapport de recherche n° 9999 — February 2008 — 26 pages

Abstract: In this paper, we tackle the open problem of snap-stabilization in message-passing systems. Snap-stabilization is a nice approach to design protocols that withstand transient faults. Compared to the well-known self-stabilizing approach, snap-stabilization guarantees that the effect of faults is contained immediately after faults cease to occur. Our contribution is twofold: we show that (1) snap-stabilization is impossible for a wide class of problems if we consider networks with finite yet unbounded channel capacity; (2) snap-stabilization becomes possible in the same setting if we assume bounded-capacity channels. We propose three snap-stabilizing protocols working in fully-connected networks. Our work opens exciting new research perspectives, as it enables the snap-stabilizing paradigm to be implemented in actual networks.

Key-words: Distributed systems, Distributed algorithm, Self-stabilization, Snap-Stabilization

* Université Paris-Sud, France
† CNRS, Université Paris-Sud, France
‡ Computer Science Department, Kent State University, USA
§ Université Paris 6, LIP6-CNRS & INRIA, France

Unité de recherche INRIA Futurs
Parc Club Orsay Université, ZAC des Vignes,
4, rue Jacques Monod, 91893 ORSAY Cedex (France)
Téléphone : +33 1 72 92 59 00 — Télécopie : +33 1 60 19 66 08

Stabilisation instantanée dans les systèmes à passage de messages

Résumé : Dans cet article, nous considérons le problème, jusqu'ici ouvert, de la stabilisation instantanée dans les systèmes à passage de messages. La stabilisation instantanée est une approche élégante permettant de réaliser des protocoles qui supportent les fautes transitoires. Par rapport à l'approche auto-stabilisante, la stabilisation instantanément stabilisante assure que l'effet des fautes est contenu immédiatement après que celles-ci cessent. Notre contribution est double: nous prouvons que (1) la stabilisation instantanée est impossible pour de nombreux problèmes si nous supposons des réseaux où la capacité des canaux de communications est finie mais non bornée; (2) la stabilisation instantanée devient possible avec les mêmes paramètres si on suppose que la capacité des canaux est bornée. A titre d'exemple, Nous proposons trois protocoles instantanément stabilisants fonctionnant dans un réseau complet. Ces travaux ouvrent de nouvelles perspectives de recherche car ils démontrent que la stabilisation instantanée peut être implantée dans les réseaux actuels.

Mots-clés : Systèmes distribués, Algorithme distribué, Auto-stabilisation, Stabilisation Instantanée

1 Introduction

Self-stabilization [23] is an elegant approach to forward failure recovery. Regardless of the global state to which the failure drives the system, after the influence of the failure stops, a self-stabilizing system is guaranteed to resume correct operation. This guarantee comes at the expense of temporary safety violation. That is, a self-stabilizing system may behave incorrectly as it recovers. Bui *et al* [11] introduce a related concept of *snap-stabilization*. Given a problem specification, a system is guaranteed to perform according to this specification regardless of the initial state. If the system is sensitive to safety violation snap-stabilization becomes an attractive option. However, the snap-stabilizing protocols presented thus far assume a rather abstract shared memory model. In this model a process reads the states of all of its neighbors and updates its own state in a single atomic step. The protocol design with forward recovery mechanisms such as self- and snap-stabilization under more concrete program model such as asynchronous message-passing is rather challenging. As Gouda and Multari [26] demonstrate, if channels can hold an arbitrary number of messages, a large number of problems could not be solved by self-stabilizing algorithms: a pathological corrupted state with incorrect messages in the channels may prevent the protocol from stabilizing. See also Katz and Perry [29] for additional detail on this topic. The issue is exacerbated for snap-stabilization by the stricter safety requirements. Thus, however attractive the concept, the applicability of snap-stabilization to concrete models, such as message-passing models remained. In this paper we address this problem. We outline the bounds of the achievable and present snap-stabilizing solutions in message-passing systems for several practical problems.

Related literature. Several studies modify the concept of self-stabilization to add safety property during recovery from faults. Dolev and Herman [24] introduce *super-stabilization* where a self-stabilizing protocol can recover from a local fault while satisfying a safety predicate. This theme is further developed as *fault-containment* [25].

A number of snap-stabilizing protocols are presented in the literature. In particular *propagation of information with feedback*(PIF) is a popular problem to address [11, 10, 12, 20, 14, 9, 19]. Several studies present snap-stabilizing token circulation protocols [30, 16, 18]. There also exists snap-stabilizing protocols for neighborhood synchronization [28], binary search tree construction [8] and cut-set detection [17]. Cournier *et al* [15] propose a method to add snap-stabilization to a large class of protocols.

Unlike snap-stabilization, self-stabilizing protocol were designed for message-passing systems of unbounded capacity channels. Afek and Brown [2] use a string of random sequence numbers to counteract the problem of infinite-capacity channels and design a self-stabilizing *alternating-bit* protocol (ABP). Delaët *et al* [22] propose a method to design self-stabilizing protocols for a class of terminating problems in message-passing systems with lossy channels of unbounded capacity. Awerbuch *et al* [6] describe the property of *local correctability* and demonstrate who to design locally-correctable self-stabilizing protocols. Researchers also consider message-passing systems with bounded capacity channels [1, 33, 27, 5, 7].

Our contribution. In this paper, we address the problem of *snap-stabilization* in message-passing systems. We introduce the concept of *safety-distributed problem specification* that encompasses most practical problems and show that it is impossible to satisfy by a snap-stabilizing protocol in message-passing systems with unbounded finite channel capacity. That is if the channel capacity bound is unknown to the processes. As a constructive contribution, we show that snap-stabilization becomes possible if bound for the channel capacity is known. We present the snap-stabilizing protocols that solve the PIF, the ID-learning and the mutual exclusion problems. To the best of our knowledge these are the first snap-stabilizing protocols in such a concrete program model.

Paper outline. The rest of the paper is organized as follows. We define the message-passing program model in Section 2. In the same section, we describe the notion of snap-stabilization and problem specifications. In Section 3, we prove the impossibility of snap-stabilization in message-passing systems with channels of infinite capacity. We present the snap-stabilizing algorithms for the system with bounded capacity channels in Section 4. We conclude the paper in Section 5.

2 The Model

We consider distributed systems having a *finite number of processes* and a *fully-connected topology*: any two distinct processes can communicate together by sending messages through a bidirectionnal link (*i.e.*, two channels in the opposite direction).

A process is a sequential deterministic machine that uses a local memory, a local algorithm, and input/output capabilities. Intuitively, such a process executes a local algorithm. This 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 sake of simplicity, we assume that every process numbers its channels from 1 to $n-1$ (n being the number of processes). In the following, we will indifferently use the notation q to designate the process q or the local channel number of q in the code of some process p. We assume that the channels are FIFO but not necessary *reliable* (messages can be lost). However they all satisfy the following property: if an origin process o sends infinitely many messages to a destination process d, then infinitely many messages are eventually received by d from o. Also, we assume that any message that is never lost is received in a finite (but unbounded) time.

The messages are of the following form: $\langle$*message-type*,*message-value*$\rangle$. The *message-value* field is omitted if the message does not carry any value. The messages can contain more than one *message-value*.

An protocol consists of a collection of actions. An 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 and/or message *sendings*. 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 of an action, if executed, are done in one atomic step. An action is said *enabled* when its guard is true. When several actions are simultaneously enabled at a process p, all these actions are sequentially executed following the order of their appearance in text of the protocol.

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

A distributed system can be described using a *transition system* [32]. A *transition system* is a 3-uple $\mathcal{S} = (\mathcal{C}, \mapsto, \mathcal{I})$ such that: $\mathcal{C}$ is set of configurations, $\mapsto$ is a binary transition relation on $\mathcal{C}$, and $\mathcal{I} \subseteq \mathcal{C}$ is the set of initial configurations. Using the notion of transition system, we can modelize the executions of a distributed system as follows: an *execution* of $\mathcal{S} = (\mathcal{C},\mapsto,\mathcal{I})$ is a *maximal* sequence of configurations $\gamma_0, \ldots, \gamma_{i-1}, \gamma_i, \ldots$ such that: $\gamma_0 \in \mathcal{I}$ and $\forall i > 0$, $\gamma_{i-1} \mapsto \gamma_i$ ($\gamma_{i-1} \mapsto \gamma_i$ is referred to as a *step*). In this paper, we only consider systems $\mathcal{S} = (\mathcal{C},\mapsto,\mathcal{I})$ such that $\mathcal{I} = \mathcal{C}$.

Snap-Stabilization. In the following, a *specification* is a predicate defined on the executions.

Definition 1 (Snap-Stabilization [11]) *Let $\mathcal{SP}_{\mathcal{T}}$ be a specification. An 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 a snap-stabilizing protocol does not guarantee that the system never works in a fuzzy manner. Actually, the main idea behind the snap-stabilization is the following: the protocol is seen as a *function* and the function ensures two properties despites the arbitrary initial configuration of the system: (1) Upon an *external* (*w.r.t.* the protocol) *request* at a process p, the process p (called the *initiator*) starts a *computation* of the function in finite time using special actions called *starting actions*. (2) If the process p starts an *computation*, then the computation performs an *expected task*. With such properties, the protocol always satisfies its specifications. Indeed, when the protocol receives a request, this means that an external application (or a user) requests the computation of a specific task provided by the protocol. In this case, a snap-stabilizing protocol guarantees that the requested task is executed as expected. On the contrary, when there is no request, there is nothing to guarantee[1].

[1]This latter point is the basis of many misunderstandings about snap-stabilization. Indeed, due to the arbitrary initial configuration, some computations may initially run in the system without having been started: of course, snap-stabilization does not provide any guarantee on these non-requested computations. Consider, for instance, the problem of *mutual exclusion*. Starting from any configuration, a snap-stabilizing protocol cannot prevent several (non-requesting) processes to execute the critical section simultaneously. However, it guarantees that every requesting process executes the critical section in an exclusive manner.

Specifications. Due to the *Start* and *Correctness* properties it has to ensure, snap-stabilization requires specifications based on a sequence of actions (request, start, ...) rather than a particular subset of configurations (*e.g.*, the *legitimate configurations*). Hence, for any task $\mathcal{T}$, we consider specifications of the following form:

- When requested, an *initiator* starts a computation of $\mathcal{T}$ in a finite time. **(Start)**
- Any computation of $\mathcal{T}$ that is started is correctly performed. **(Correctness)**

In this paper, the two first protocols we present are of a particular class: the *wave* protocols [32]. The particularity of such protocols is that they compute tasks that are *finite* and each of their computations contains at least one *decision event* that causally depends on an action at each process. Hence, our specifications for wave protocols contain two additionnal requirements:

- Each computation (even non-started) terminates in finite time. **(Termination)**
- When the protocol terminates, if a computation was started, then at least one decision occurred and such a decision causally depends on an action at every process. **(Decision**)

Self- *vs.* Snap-Stabilization. Snap-stabilizing protocols are often compared to the self-stabilizing protocols — such protocols converge in a finite time to a specified behavior starting from any initial configuration ([23]). The main advantage of the snap-stabilizing approach compared to the self-stabilizing one is the following: while a snap-stabilizing protocol ensures that any request is satisfied despite the arbitrary initial configuration, a self-stabilizing protocol often needs to be repeated an unbounded number of times before guarantying the proper processing of any request.

3 Impossibility of Snap-Stabilization in Message-Passing with Unbounded Capacity Channels

In [3], Alpern and Schneider observe that a specification is an intersection of *safety* and *liveness* properties. In [4], the same authors define a *safety* property as a set of "bad things" that must never happen. Hence, it is sufficient to show that a prefix of an execution contains a "bad thing" to prove that the execution (and so the protocol) violates the safety property. We now consider *safety-distributed* specifications, *i.e.*, specifications having some *safety-distributed* properties. Roughly speaking, a *safety-distributed* property is a safety property that does not only depend on the behavior of a single process: some local behaviors at some processes are forbidden to be executed simultaneously while they are possible and do not violate the safety-distributed property if they are executed alone. For example, in the mutual exclusion problem, a requesting process eventually executes the critical section but no two requesting processes must execute the critical section concurrently.

We now introduce the notions of *abstract configuration, state-projection, and sequence-projection.* These three notions are useful to formalize *safety-distributed* specifications.

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

Definition 3 (State-Projection) *Let γ be configuration and p be a process. The* state-projection *of γ on p, noted $\phi_p(\gamma)$, is the local state of p in γ. Similary, the* state-projection *of γ on all processes, $\phi(\gamma)$ 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, noted $\Phi_p(s)$, is the state sequence $\phi_p(\gamma_0),\phi_p(\gamma_1), \ldots$ Similary, the* sequence-projection *of s on all processes, noted $\Phi(s)$, 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* 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 $\Phi(e_1) =$ 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 $\Phi_p(e_p^1) = \Phi_p($BAD$)$.

Almost all classical problems of distributed computing have *safety-distributed* specifications, *e.g.*, mutual exclusion, phase synchronization, ... For example, in mutual exclusion a *bad-factor* is any sequence of abstract configurations where several requesting processes executes the critical section concurrently. 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.

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 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, $e_p^1 = \beta_0,\beta_1,\ldots$, and e_p^2 such that $e_p = e_p^0e_p^1e_p^2$ and $\Phi_p(e_p^1) = \Phi_p($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 each two processes p, q such that $p \neq q$, the link $\{p,q\}$ as 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 Point (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 and β_1. Any message that p receives in $\beta_0 \mapsto \beta_1$ can be received by p in the first step from γ_0: $\gamma_0 \mapsto \gamma_1$. Now, $\phi_p(\gamma_0) = \phi_p(\beta_0)$. So, p can behave in $\gamma_0 \mapsto \gamma_1$ as in $\beta_0 \mapsto \beta_1$. In that case, $\phi_p(\gamma_1) = \phi_p(\beta_1)$. Hence, if every process p behaves in $\gamma_0 \mapsto \gamma_1$ as in the first step of its execution factor e_p^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 noted $PRED$ such that $\Phi(PRED) = \texttt{BAD}$. 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. □

Intuitively, the impossibility result of Theorem 1 is due to the fact that in a system with unbounded capacity channels, any initial configuration can contain an unbounded number of messages. If we consider now systems with bounded and known channel capacity, we can circumvent the impossibility result by designing protocols that require a number of messages that is greater than the bound on the channel capacity to perform their specified task. This is our approach in the next section.

4 Snap-Stabilizing Message-Passing Protocols

We now consider systems with channels having a bounded capacity. In such systems, we assume that if a process sends a message in a channel that is full, then the message is lost. We restrict our study to systems with single-message capacity channels. The extention to an arbitrary but known bounded message capacity is straightforward (see [6, 7]). We propose three snap-stabilizing protocols (Algorithms 1-3) for the *Propagation of Information with Feedback* (PIF), IDs-Learning, and mutual exclusion problem, respectively. The PIF is a basic tool allowing us to solve the two other problems. The IDs-Learning is a simple application of the PIF. Finally, the mutual exclusion protocol uses the two former protocols.

4.1 A PIF Protocol

The concept of *Propagation of Information with Feedback* (PIF), also called *Wave Propagation*, has been introduced by Chang [13] and Segall [31]. PIF has been extensively studied in the distributed literature because many fundamental protocols, *e.g.*, *Reset*, *Snapshot*, *Leader Election*, and *Termination Detection*, can be solved using a PIF-based solution. The

PIF scheme can be informally described as follows: when requested, a process starts the first phase of the PIF-computation by broadcasting a specific message m into the network (this phase is called the *broadcast phase*). Then, every non-initiator acknowledges[2] to the initiator the receipt of m (this phase is called the *feedback phase*). The PIF-computation terminates when the initiator received acknowledgments from every other process and decides taking these acknowledgments into account. In distributed systems, any process may need to initiate a PIF-computation. Thus, any process can be the initiator of a PIF-computation and several PIF-computations may run concurrently. Hence, any PIF protocol has to cope with concurrent PIF-computations.

Specification 1 (PIF-Execution) *An execution e satisfies* PIF-execution*(e)* if and only if *e satisfies the following four properties:*

- **Start.** *When there is a request for a process p to broadcast a message* m*, p starts a* PIF-computation *in finite time.*
- **Correctness.** *During any* PIF-computation *started by p for the message* m*:*
  - *Any process different of p receives* m*.*
  - *p receives acknowledgments for* m *from every other process.*
- **Termination.** *Any PIF-computation (even non-started) terminates in finite time.*
- **Decision.** *When a* PIF-computation *started by p terminates at p, p decides taking all acknowledgments of the last message it broadcasts into account only.*

Approach. In the following, we refer to our snap-stabilizing PIF as Protocol $\mathcal{PIF}$. We describe our approach using a network of two processes: p and q. The generalization to a fully-connected network of more than two processes is straightforward and presented in Algorithm 1.

Consider the following example. Each process maintains in the variable *Old* its own age and p wants to know the age of q. Then, p performs a PIF of the message "How old are you?". To that goal, we need the following input/output variables:

- $\texttt{Request}_p$. This variable is used to manage the PIF'requests for p. $\texttt{Request}_p$ is (externally) set to `Wait` when there is a request for p to perform a PIF. $\texttt{Request}_p$ is switched from `Wait` to `In` at the start of each PIF-computation (*n.b.* p starts a PIF-computation upon a request only). Finally, $\texttt{Request}_p$ is switched from `In` to `Done` at the termination of each PIF-computation (this latter switch also corresponds to the *decision event*). Since a PIF-computation is started by p, we assume that p does not set $\texttt{Request}_p$ to `Wait` until the termination of the current PIF-computation, *i.e.*, until $\texttt{Request}_p = \texttt{Done}$.
- $\texttt{B-Mes}_p$. This variable contains the message to broadcast.
- $\texttt{F-Mes}_q$. When q receives the broadcast message, q assigns the acknowledgment message in $\texttt{F-Mes}_q$.

[2] An acknowledgment is a message sent by the receiving process to inform the sender about data it have correctly received.

Using these variables, we perform a PIF of "How old are you?" as follows: $\mathcal{PIF}$.B-Mes$_p$ and $\mathcal{PIF}$.Request$_p$ are respectively (externally) set to "How old are you?" and Wait meaning that we request that p broadcasts "How old are you?" to q. Consequently to this request, Protocol $\mathcal{PIF}$ starts a PIF-computation by setting $\mathcal{PIF}$.Request$_p$ to In and this computation terminates when $\mathcal{PIF}$.Request$_p$ is set to Done. Between this start and this termination, $\mathcal{PIF}$ generates two events. First, a "**receive-brd**$\langle$*How old are you*?$\rangle$ **from** p" event at q. When this event occurs, q sets $\mathcal{PIF}$.F-Mes$_q$ to Old_q so that $\mathcal{PIF}$ feedbacks the value of Old_q to p. Protocol $\mathcal{PIF}$ then transmits the value of Old_q to p: this generates a "**receive-fck**$\langle x\rangle$ **from** q" event at p where x is the value of Old_q.

A naive attempt to implement Protocol $\mathcal{PIF}$ could be the following:

- When $\mathcal{PIF}$.Request$_p$ = Wait, p sends a broadcast message containing the data message $\mathcal{PIF}$.B-Mes$_p$ to q and sets $\mathcal{PIF}$.Request$_p$ to In (meaning that the PIF-computation is in processing).
- Upon receiving a broadcast message containing the data B, a "**receive-brd**$\langle B\rangle$ **from** p" event is generated at q so that the application (at q) that uses the PIF treats the message B. Upon this event, the application is assumed to set the feedback message into $\mathcal{PIF}$.F-Mes$_q$. Then, q sends a feedback message containing $\mathcal{PIF}$.F-Mes$_q$ to p.
- Upon receiving a feedback message containing the data F, a "**receive-fck**$\langle F\rangle$ **from** q" event is generated at p so that the application (at p) that uses the PIF treats the feedback and then sets $\mathcal{PIF}$.Request$_p$ to Done.

Unfortunately, such a simple approach is not snap-stabilizing in our system:

(1) Due to the unreliability of the channels, the system may suffer of deadlock. If the broadcast message from p or feedback message from q are lost, Protocol $\mathcal{PIF}$ never terminates at p.

(2) Due to the arbitrary initial configuration, the link $\{p,q\}$ may initially already contain an arbitrary message in the channel from p to q and another in the channel from q to p. Hence, after sending the broadcast message to q, p may receive a feedback message that was not sent by q. Also, q may receive a broadcast message that was not sent by p: as a consequence, q generates an undesirable feedback message.

To circumvent these two problems, we use two additionnal variables at each process:

- State$_p$ $\in$ {0,1,2,3,4} (resp. State$_q$) is a flag value that p (resp. q) puts into its messages.
- NeigState$_p$ (resp. NeigState$_q$) is equal to the last State$_q$ (resp. State$_p$) that p (resp. q) receives from q (resp. p).

(Note that we use a single message type, noted PIF, to manage the PIF-computations initiated by both p and q.)

Our protocol works as follows: p starts a *PIF-computation* by setting State$_p$ to 0. Then, until State$_p$ = 4, p repeatedly sends $\langle$PIF,B-Mes$_p$,F-Mes$_p$,State$_p$,NeigState$_p\rangle$ to q. When q receives $\langle B,F,pState,qState\rangle$ (from p), q updates NeigState$_q$ to $pState$ and then

sends a message $\langle$PIF,B-Mes$_q$,F-Mes$_q$,State$_q$,NeigState$_q\rangle$ to p if $pState < 4$ (*i.e.*, if p is still waiting for a message from q). Finally, p increments State$_p$ only when it receives a $\langle$PIF,B,B,$qState$,$pState\rangle$ message from q such that State$_p$ $= pState$ and $pState < 4$. Hence, after p starts, State$_p = 4$ only after p successively receives $\langle$PIF,B,F,$qState$,$pState\rangle$ messages (from q) with $pState$ = 0,1,2,3. Now, considering the arbitrary initial value of NeigState$_q$ and the at most two arbitrary messages initially in the link $\{p,q\}$ (one in the channel from p to q and one in the channel from q to p), we are sure that after p starts, p receives a $\langle$PIF,B,F,$qState$,$pState\rangle$ from q with $pState = State_p = 3$ only if this message was sent by q consequently to the reception by q of a message sent by p.

Figure 1 illustrates the worst case of Protocol $\mathcal{PIF}$ in terms of configurations. In this example, p may increment State$_p$ after receiving the initial message with the flag value $pState = 0$. Then, if q starts a PIF-computation, q sends messages with the flag value $pState = 1$ until receiving (from p) the initial message with the value $pState = 2$. Hence, p can still increment State$_p$ twice due to the values 1 and 2 (*i.e.*, State$_p$ then reaches the value 3). But, after these incrementations, p no more increments State$_p$ until receiving a message with the value $pState = 3$ and q starts sending messages with the value $pState = 3$ only after receiving a message from p with the value $pState = 3$. Finally, note that after receiving a message with the value $pState = 3$, p increments State$_p$ to 4 and stops sending messages until the next request. This ensures that if the requests eventually stop, the system eventually contains no message.

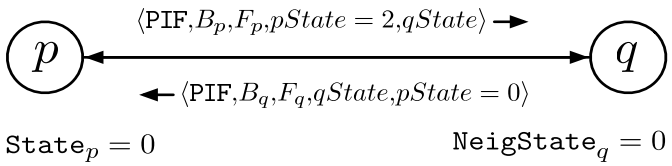

Figure 1: Worst case of Protocol $\mathcal{PIF}$ in terms of configurations.

It remains to see when a process can generate the **receive-brd** and **receive-fck** events:

- q receives at least 4 copies of the broadcast messages. But, q generate a **receive-brd** event only once for each broadcast message: when q switches NeigState$_q$ to 3.
- After it starts, p is sure to receive the "good" feedback only when it receives a message with $pState = $ State$_p = 3$. As previously, to limit the number of events, p generates a **receive-fck** events only when it switches State$_p$ from 3 to 4. The other copies are then ignored. Also, note that after receiving this message, p can only receives duplicates until the next PIF-computation. Hence, when p decides, it decides only taking the "good" feedbacks into account.

We generalize this snap-stabilizing one-to-one broadcast with feedback to a snap-stabilizing all-to-all broadcast with feedback (*i.e.*, a PIF) in Algorithm 1. It is important to note that

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

Constant: n: integer, number of processes

Variables:

$\texttt{Request}_p \in \{\texttt{Wait},\texttt{In},\texttt{Done}\}$	:	input/output variable
$\texttt{B-Mes}_p$	:	data to broadcast, input variable
$\texttt{F-Mes}_p[1 \ldots n-1]$	:	array of messages to feedback, input variable
$\texttt{State}_p[1 \ldots n-1] \in \{0,1,2,3,4\}^{n-1}$	:	internal variable
$\texttt{NeigState}_p[1 \ldots n-1] \in \{0,1,2,3,4\}^{n-1}$	:	internal variable

Actions:

$\texttt{A}_1$ :: $(\texttt{Request}_p = \texttt{Wait}) \rightarrow$

$\texttt{Request}_p \leftarrow \texttt{In}$ /* Start */
for all $q \in [1 \ldots n-1]$ **do**
  $\texttt{State}_p[q] \leftarrow 0$
done

$\texttt{A}_2$ :: $(\texttt{Request}_p = \texttt{In}) \rightarrow$

if $(\forall q \in [1 \ldots n-1], \texttt{State}_p[q] = 4)$ **then**
  $\texttt{Request}_p \leftarrow \texttt{Done}$ /* Termination */
else
  for all $q \in [1 \ldots n-1]$ **do**
    if $(\texttt{State}_p[q] \neq 4)$ **then**
      send$\langle \texttt{PIF},\texttt{B-Mes}_p,\texttt{F-Mes}_p[q],\texttt{State}_p[q],\texttt{NeigState}_p[q]\rangle$**to** q
    end if
  done
end if

$\texttt{A}_3$ :: **receive**$\langle \texttt{PIF},B,F,qState,pState\rangle$**from** $q \rightarrow$

if $(\texttt{NeigState}_p[q] \neq 3) \wedge (qState = 3)$ **then**
  generate a "**receive-brd**$\langle B\rangle$ **from** q" event
end if
$\texttt{NeigState}_p[q] \leftarrow qState$
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 "**receive-fck**$\langle F\rangle$ **from** q" event
  end if
end if
if $(qState < 4)$ **then**
  send$\langle \texttt{PIF},\texttt{B-Mes}_p,\texttt{F-Mes}_p[q],\texttt{State}_p[q],\texttt{NeigState}_p[q]\rangle$**to** q
end if

our protocol does not prevent processes to generate unexpected **receive-brd** or **receive-fck** events. Actually, what our protocol ensures is: when a process p starts to broadcast a message m, then (1) every other process eventually receives m (**receive-brd**), (2) p eventually receives a feedback for m from any other process (**receive-fck**), and (3) p decides ($\mathcal{PIF}.\texttt{Request}_p \leftarrow \texttt{Done}$) by only taking the "good" feedbacks into account. Another interesting property of our protocol is the following: after the first complete computation of $\mathcal{PIF}$ (from the start to the termination), the channels from and to p contain no message from the initial configuration.

Proof of Snap-Stabilization. The proof of snap-stabilization of $\mathcal{PIF}$ just consists in showing that, despite the arbitrary initial configuration, any execution of $\mathcal{PIF}$ always satisfies the four properties of Specification 1. In the following proofs, the *message-values* will be replaced by "−" when they have no impact on the reasonning.

Lemma 1 (Start) *Starting from any configuration, when there is a request for a process p to broadcast a message, p starts a* PIF-computation *in finite time.*

Proof. We assumed that $\texttt{Request}_p$ is externally set to `Wait` when there is a request for the process p to broadcast a message. Moreover, we claim that a process p starts Protocol $\mathcal{PIF}$ by switching $\texttt{Request}_p$ from `Wait` to `In`. Now, when $\texttt{Request}_p = \texttt{Wait}$, Action $\texttt{A}_1$ is continuously enabled at p and by executing $\texttt{A}_1$, p sets $\texttt{Request}_p$ to `In`. Hence, the lemma holds. □

The following Lemmas (Lemmas 2-6) hold assuming that no *PIF-computation* (even non-started) can be interrupted due to another request:

Hypothesis 1 *While* $\texttt{Request}_p \neq \texttt{Done}$, $\texttt{Request}_p$ *is not (externally) set to* `Wait`.

Lemma 2 *Consider two distinct processes p and q. Starting from any configuration, if* $(\texttt{Request}_p = \texttt{In}) \wedge (\texttt{State}_p[q] < 4)$, *then* $\texttt{State}_p[q]$ *is eventually incremented.*

Proof. Assume, for the purpose of contradiction, that $\texttt{Request}_p = \texttt{In}$ and $\texttt{State}_p[q] = i$ with $i < 4$ but $\texttt{State}_p[q]$ is never incremented. Then, from Algorithm 1, $\texttt{Request}_p = \texttt{In}$ and $\texttt{State}_p[q] = i$ hold forever and by 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 $\texttt{A}_3$, $\texttt{NeigState}_q[p] = i$ 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 (see Action $\texttt{A}_3$). 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 (Termination) *Starting from any configuration, any* PIF-computation *(even non-started) terminates in finite time.*

Proof. Assume, for the purpose of contradiction, that a *PIF-computation* never terminates at some process p, *i.e.*, $\texttt{Request}_p \neq \texttt{Done}$ forever. Then, $\texttt{Request}_p = \texttt{In}$ eventually holds forever by Lemma 1. Now, by Lemma 2 and owing the fact that $\forall q \in [1 \dots n-1]$, $\texttt{State}_p[q]$ cannot decrease while the computation is not terminated at p, we can deduce that p eventually satisfies "$\forall q \in [1 \dots n-1], \texttt{State}_p[q] = 4$" forever. In this case, p sets $\texttt{Request}_p$ to `Done` by Action $\texttt{A}_2$ — a contradiction. □

Lemma 4 *Let p and q be two distinct processes. After p starts to broadcast a message from an arbitrary configuration, 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 are of the form $\langle \texttt{PIF},-,-,i,-\rangle$ *with* $i \neq 3$.
(2) $\texttt{NeigState}_q[p] \neq 3$.
(3) Any message in the channel from q to p are of the form $\langle \texttt{PIF},-,-,-,j\rangle$ *with* $j \neq 3$.

Proof. p starts to broadcast a message by executing Action $\texttt{A}_1$ (*n.b.* $\texttt{A}_1$ is the only *starting action* of $\mathcal{PIF}$). When p executes $\texttt{A}_1$, p sets (in particular) $\texttt{State}_p[q]$ 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 incrementations of $\texttt{State}_p[q]$:

- **From 0 to 1.** $\texttt{State}_p[q]$ switches from 0 to 1 only after p receives $\langle \texttt{PIF},-,-,-,0\rangle$ from q (Action $\texttt{A}_3$). 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 $\texttt{NeigState}_q[p] = 1$ held when q sent $\langle \texttt{PIF},-,-,-,1\rangle$ to p. From that point, $\texttt{NeigState}_q[p] = 1$ 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, $\texttt{NeigState}_q[p] \neq 3$ 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, $\texttt{NeigState}_q[p] \neq 3$ (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 (Correctness) *Starting from any configuration, if p starts to broadcast a message* m*, then:*

- *Any process different of p receives* m.
- *p receives acknowledgments for* m *from every other process.*

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

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

When q receives the first $\langle \mathtt{PIF}, m, -, 3, - \rangle$ message from p, q generates a "**receive-brd**$\langle m \rangle$ **from** p" event and then starts to send $\langle \mathtt{PIF}, -, F, -, 3 \rangle$ messages to p[3]. From that point and until p decides, q only receives $\langle \mathtt{PIF}, m, -, 3, - \rangle$ 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 $\langle \mathtt{PIF}, -, F, -, 3 \rangle$ message from q, p generates a "**receive-fck**$\langle F \rangle$ **from** q" event and then sets $\mathtt{State}_p[q]$ to 4.

Hence, $\forall q \in [1 \dots 0]$, the broadcast of m generates a "**receive-brd**$\langle m \rangle$ **from** p" event at process q and then an associated "**receive-fck**$\langle F \rangle$ **from** q" event at p, which proves the lemma. □

Lemma 6 (Decision) *Starting from any configuration, when a* PIF-computation *started by p terminates at p, p decides taking all acknowledgments of the last message it broadcasts into account only.*

Proof. First, p starts to broadcast a message m by executing Action $\mathtt{A}_1$: p switches $\mathtt{Request}_p$ from `Wait` to `In` and sets $\mathtt{State}_p[q]$ to 0, $\forall q \in [1 \dots 0]$. Then, $\mathtt{Request}_p$ remains equal to `In` until p decides by $\mathtt{Request}_p \leftarrow$ `Done`. Now, (1) p decides in finite time by Lemma 3, (2) when p decides, we have $\mathtt{State}_p[q] = 4$, $\forall q \in [1 \dots 0]$ (Action $\mathtt{A}_2$), and (3) after p decides, each time q receives a message from p with the data m, the message is ignored (this is a consequence of Claim (2)). From the code of Algorithm 1, we know that exactly one "**receive-fck**$\langle F \rangle$ **from** q" event per neighbor q occurs at p before p decides: when p switches $\mathtt{State}_p[q]$ from 3 to 4. Now, Lemma 5 and Claim (3) imply that each of these feedbacks corresponds to an acknowledgment for m. Hence, p decides taking all acknowledgments of m into account only and the lemma is proven. □

By Lemmas 1, 3, 5, and 6, starting from any arbitrary initial configuration, any execution of $\mathcal{PIF}$ always satisfies Specification 1. Hence, follows:

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

Below, we give an additionnal property of $\mathcal{PIF}$, this property will be used in the snap-stabilization proof of Protocol $\mathcal{ME}$.

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

Property 1 *If p starts a PIF-computation (using Protocol $\mathcal{PIF}$) in the configuration γ_0 and the computation 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-computation (using Protocol $\mathcal{PIF}$) in the configuration γ_0. Then, as $\mathcal{PIF}$ is snap-stabilizing for 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-computation terminates at p in configuration γ_k □

4.2 A IDs-Learning Protocol

Protocol $\mathcal{IDL}$ (its implementation is presented in Algorithm 2) is a simple application of Protocol $\mathcal{PIF}$. This protocol assumes IDs on processes (ID_p denotes the identity of the process p) and uses three variables at each process p:

- $\texttt{Request}_p \in \{\texttt{Wait},\texttt{In},\texttt{Done}\}$. The goal of this variable is the same as in $\mathcal{PIF}$.
- $\texttt{minID}_p$. After a complete execution of $\mathcal{IDL}$ (*i.e.*, from the start to the termination), $\texttt{minID}_p$ contains the minimal ID of the system.
- $\texttt{ID-Tab}_p[1 \ldots n]$. After a complete execution of $\mathcal{IDL}$, $\texttt{ID-Tab}_p[q]$ contains the ID of the p'neighbor q.

When requested ($\mathcal{IDL}.\texttt{Request}_p = \texttt{Wait}$) at p, Protocol $\mathcal{IDL}$ evaluates the ID of each of its neighbors q and the minimal ID of the system using Protocol $\mathcal{PIF}$. The results of the computation are available for p since p decides (when $\mathcal{IDL}.\texttt{Request}_p \leftarrow \texttt{Done}$). Based on the specification of $\mathcal{PIF}$, it is easy to see that $\mathcal{IDL}$ is snap-stabilizing for the following specification:

Specification 2 (IDs-Learning-Execution) *An execution e satisfies* IDs-Learning-execution*(e)* if and only if *e satisfies the following four properties:*

- **Start.** *When requested, a process p starts a* IDs-Learning-computation *in finite time.*
- **Correctness.** *At the end of any* IDs-Learning-computation *started by p:*
  - $\forall q \in [1 \ldots n-1]$, $\texttt{ID-Tab}_p[q] = ID_q$.
  - $\texttt{minID}_p = \min(\{ID_q,\ q \in [1 \ldots n-1]\} \cup \{ID_p\})$.
- **Termination.** *Any* IDs-Learning-computation *(even non-started) terminates in finite time.*
- **Decision.** *If p is in a* terminal state *and a* IDs-Learning-computation *was started by p, then p decided knowing the minimal ID of the system and the ID of every of its neighbors.*

Theorem 3 *Protocol $\mathcal{IDL}$ is snap-stabilizing for Specification 2.*

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

Constant:

n : integer, number of processes
ID_p : integer, identity of p

Variables:

$\texttt{Request}_p \in \{\texttt{Wait},\texttt{In},\texttt{Done}\}$: input/output variable
$\texttt{minID}_p$: integer, output variable
$\texttt{ID-Tab}_p[1 \ldots n-1] \in \mathbb{N}^{n-1}$: output variable

Actions:

A_1	::	$(\texttt{Request}_p = \texttt{Wait})$	$\rightarrow$	$\texttt{Request}_p \leftarrow \texttt{In}$ /* Start */ $\texttt{minID}_p \leftarrow ID_p$ $\mathcal{PIF}.\texttt{B-Mes}_p \leftarrow \texttt{IDL}$ $\mathcal{PIF}.\texttt{Request}_p \leftarrow \texttt{Wait}$
A_2	::	$(\texttt{Request}_p = \texttt{In}) \wedge (\mathcal{PIF}.\texttt{Request}_p = \texttt{Done})$	$\rightarrow$	$\texttt{Request}_p \leftarrow \texttt{Done}$ /* Termination */
A_3	::	**receive-brd**$\langle \texttt{IDL} \rangle$ **from** q	$\rightarrow$	$\mathcal{PIF}.\texttt{F-Mes}_p[q] \leftarrow ID_p$
A_4	::	**receive-fck**$\langle qID \rangle$ **from** q	$\rightarrow$	$\texttt{ID-Tab}_p[q] \leftarrow qID$ $\texttt{minID}_p \leftarrow \min(\texttt{minID}_p, qID)$

4.3 A Mutual Exclusion Protocol

We now consider the problem of *mutual exclusion*. Mutual exclusion is a well-known mechanism allowing to allocate a common resource. Indeed, a mutual-exclusion mechanism ensures that a special section of code, called *critical section* (noted $\langle \texttt{CS} \rangle$ in the following), can be executed by at most one process at any time. The processes can use their critical section to access to a shared ressource. Generally, this resource corresponds to a set of shared variables in a common store or a shared hardware device (*e.g.*, a printer). The first snap-stabilizing implementation of mutual exclusion is presented in [21] but in the state model (a stronger model than the message-passing model). In [21], authors adopt the following specification[4]:

Specification 3 (ME-Execution) *An execution e satisfies* ME-execution*(e)* if and only if *e satisfies the following two properties:*

- **Start.** *Any process that requests the $\langle \texttt{CS} \rangle$ enters in the $\langle \texttt{CS} \rangle$ in finite time.*
- **Correctness.** *If a requesting process enters in the $\langle \texttt{CS} \rangle$, then it executes the $\langle \texttt{CS} \rangle$ alone.*

Approach. We now propose a snap-stabilizing mutual exclusion protocol called Protocol $\mathcal{ME}$. The implementation of $\mathcal{ME}$ is presented in Algorithm 3. As for the previous solutions, Protocol $\mathcal{ME}$ uses the input/output variable $\texttt{Request}$. A process p (externally) sets $\mathcal{ME}.\texttt{Request}_p$ to $\texttt{Wait}$ when it requests the access to the $\langle \texttt{CS} \rangle$. Process p is then called a *requestor* and assumed to not execute $\mathcal{ME}.\texttt{Request}_p \leftarrow \texttt{Wait}$ until $\mathcal{ME}.\texttt{Request}_p = \texttt{Done}$, *i.e.*, until its current request is done.

[4] This specification was firstly introduced and justified in [15].

The main idea of the protocol is the following: we assume IDs on processes and the process with the smallest ID — called the *leader* — decides using a variable called `Value` which process can executes the ⟨`CS`⟩. When a process learns that it is authorized to access the ⟨`CS`⟩:

(1) It first ensures that no other process can execute the ⟨`CS`⟩.
(2) It then executes the ⟨`CS`⟩ if it wishes.
(3) Finally, it notifies to the leader that it releases the ⟨`CS`⟩ so that the leader (fairly) authorizes another process to access the ⟨`CS`⟩.

To apply this scheme, $\mathcal{ME}$ executes by phases from Phase 0 to 4 in such way that each process goes through Phase 0 infinitely often. For each process p, $\texttt{Phase}_p$ denotes in which phase process p is. After requesting the ⟨`CS`⟩ ($\mathcal{ME}.\texttt{Request}_p \leftarrow \texttt{Wait}$), a process p can access the ⟨`CS`⟩ only after executing Phase 0. Indeed, p can access to the ⟨`CS`⟩ only if $\mathcal{ME}.\texttt{Request}_p = \texttt{In}$ and p switches $\mathcal{ME}.\texttt{Request}_p$ from `Wait` to `In` only when executing Phase 0. Hence, our protocol has just to ensure that after executing its phase 0, a process always executes the ⟨`CS`⟩ alone. Our protocol offers such a guarantee thanks to the five phases described below:

- **Phase 0.** When a process p is in Phase 0, it starts a computation of $\mathcal{IDL}$, sets $\mathcal{ME}.\texttt{Request}_p$ to `In` if $\mathcal{ME}.\texttt{Request}_p = \texttt{Wait}$ (*i.e.*, if p requests the ⟨`CS`⟩, then the protocol takes this request into account), and finally switches to Phase 1.
- **Phase 1.** When a process p is in Phase 1, p waits the termination of $\mathcal{IDL}$ to know (1) the ID of each of its neighbors q ($\texttt{ID-Tab}_p[q]$) and (2) the leader of the system ($\mathcal{IDL}.\texttt{minID}_p$), *i.e.*, the process with the smallest ID. Then, p starts a PIF of the message `ASK` to know which is the process authorized to access the ⟨`CS`⟩ and switches to Phase 2. Upon receiving a message `ASK` from p, any process q answers `YES` if $\texttt{Value}_q$ is equal to the channel number of p at q, `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 the termination of the PIF started in Phase 1. After $\mathcal{PIF}$ terminates, the answers of any neighbors q of p are stored in $\texttt{Privileges}_p[q]$ and, so, p knows if it is authorized to access the ⟨`CS`⟩. Actually, p is authorized to access the ⟨`CS`⟩ (see $Winner(p)$) if: (1) p is the leader and $\texttt{Value}_p = 0$ or (2) the leader answers `YES` to p. If p has the authorization to access the ⟨`CS`⟩, p starts a PIF of the message `EXIT`. The goal of this message is to force all other processes to restart to Phase 0. This ensures no other process executes the ⟨`CS`⟩ until p notifies to the leader that it releases the ⟨`CS`⟩. Indeed, due to the arbitrary initial configuration, some process $q \neq p$ may believe that it is authorized to execute the ⟨`CS`⟩: if q never starts Phase 0. On the contrary, after restarting to 0, q cannot receive any authorization from the leader until p notifies to the leader that it releases the ⟨`CS`⟩. Finally, p terminates Phase 2 by switching to Phase 3.
- **Phase 3.** When a process p is in Phase 3, it waits the termination of the last PIF. After $\mathcal{PIF}$ terminates, if p is authorized to execute the ⟨`CS`⟩, then: p executes the

$\langle$CS$\rangle$ if $\mathcal{ME}$.Request$_p$ = In (*i.e.*, if the system took a request of p into account) and then either (1) p is the leader and switches Value$_p$ from 0 to 1 or (2) p is not the leader and starts a PIF of the message EXITCS to notify to the leader that it releases the $\langle$CS$\rangle$. Upon receiving such a message, the leader increments its variable Value modulus $n+1$ to authorize another process to access the $\langle$CS$\rangle$. Finally, p terminates Phase 3 by switching to Phase 4.

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

Proof of Snap-Stabilization. We begin the proof of snap-stabilization of Protocol $\mathcal{ME}$ by showing that, despite the arbitrary initial configuration, any execution of $\mathcal{ME}$ always satisfies the correctness property of Specification 3.

Assume that a process p requests the $\langle$CS$\rangle$, *i.e.*, $\mathcal{ME}$.Request$_p$ = Wait. Then, p cannot enters in the $\langle$CS$\rangle$ before executing Action A$_0$, indeed:

- p enters in the $\langle$CS$\rangle$ only if $\mathcal{ME}$.Request$_p$ = In, and
- Action A$_0$ is the only action of $\mathcal{ME}$ allowing p to set $\mathcal{ME}$.Request$_p$ to In.

Hence, to show the correctness property of Specification 3 (Corollary 1), we have just to prove that, despite the initial configuration, after p executes Action A$_0$, if p enters in the $\langle$CS$\rangle$, then it executes the $\langle$CS$\rangle$ alone (Lemma 9).

Lemma 7 *Let p be a process. Starting from any configuration, after p executes* A$_0$*, if p enters in the* $\langle$CS$\rangle$*, then every other process has switches to Phase 0 at least once.*

Proof. By checking all the actions of Algorithm 3, we can remark that after p executes A$_0$, p must execute the four actions A$_0$, A$_1$, A$_2$, and A$_3$ successively to enter in the $\langle$CS$\rangle$ (in A$_3$). Also, to execute the $\langle$CS$\rangle$ in Action A$_3$, p must satisfy the predicate $Winner(p)$. The value of the predicate $Winner(p)$ only depends on (1) the $\mathcal{IDL}$ computation started in A$_0$ and (2) the PIF of the message ASK started in A$_1$. Now, this two computations are done when p executes A$_2$. So, the fact that p satisfies $Winner(p)$ when executing A$_3$ implies that p also satisfies $Winner(p)$ when executing A$_2$. As a consequence, p starts a PIF of the message EXIT in A$_2$. Now, p executes A$_3$ only after this PIF terminates. Hence, p executes A$_3$ only after every other process executes 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 of the system with the smallest ID. 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* $(p = q \wedge \text{Value}_p = 0) \vee (p \neq q \wedge \text{Value}_p = q)$.

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

Constant:

n	:	integer, number of processes
ID_p	:	integer, identity of p

Variables:

$\texttt{Request}_p \in \{\texttt{Wait},\texttt{In},\texttt{Done}\}$	:	input/output variable
$\texttt{Phase}_p \in \{0,1,2,3,4\}$	:	internal variable
$\texttt{Value}_p \in \{0 \dots n-1\}$	:	internal variable
$\texttt{Privileges}_p[1 \dots n-1] \in \{true, false\}^{n-1}$	:	internal variable

Predicate:

$Winner(p) \equiv (\mathcal{IDL}.\texttt{minID}_p = ID_p \wedge \texttt{Value}_p = 0) \vee (\exists q \in [1 \dots n-1],\ \texttt{Privileges}_p[q] \wedge \mathcal{IDL}.\texttt{ID-Tab}_p[q] = \mathcal{IDL}.\texttt{minID}_p)$

Actions:

$\texttt{A}_0$ :: $(\texttt{Phase}_p = 0)$ →
$\mathcal{IDL}.\texttt{Request}_p \leftarrow \texttt{Wait}$
if $\texttt{Request}_p = \texttt{Wait}$ **then**
  $\texttt{Request}_p \leftarrow \texttt{In}$ /* Start */
end if
$\texttt{Phase}_p \leftarrow \texttt{Phase}_p + 1$

$\texttt{A}_1$ :: $(\texttt{Phase}_p = 1) \wedge (\mathcal{IDL}.\texttt{Request}_p = \texttt{Done})$ →
$\mathcal{PIF}.\texttt{B-Mes}_p \leftarrow \texttt{ASK}$
$\mathcal{PIF}.\texttt{Request}_p \leftarrow \texttt{Wait}$
$\texttt{Phase}_p \leftarrow \texttt{Phase}_p + 1$

$\texttt{A}_2$ :: $(\texttt{Phase}_p = 2) \wedge (\mathcal{PIF}.\texttt{Request}_p = \texttt{Done})$ →
if $Winner(p)$ **then**
  $\mathcal{PIF}.\texttt{B-Mes}_p \leftarrow \texttt{EXIT}$
  $\mathcal{PIF}.\texttt{Request}_p \leftarrow \texttt{Wait}$
end if
$\texttt{Phase}_p \leftarrow \texttt{Phase}_p + 1$

$\texttt{A}_3$ :: $(\texttt{Phase}_p = 3) \wedge (\mathcal{PIF}.\texttt{Request}_p = \texttt{Done})$ →
if $Winner(p)$ **then**
  if $(\texttt{Request}_p = \texttt{In})$ **then**
    ⟨CS⟩
    $\texttt{Request}_p \leftarrow \texttt{Done}$ /* Termination */
  end if
  if $(\mathcal{IDL}.\texttt{minID}_p = ID_p)$ **then**
    $\texttt{Value}_p \leftarrow 1$
  else
    $\mathcal{PIF}.\texttt{B-Mes}_p \leftarrow \texttt{EXITCS}$
    $\mathcal{PIF}.\texttt{Request}_p \leftarrow \texttt{Wait}$
  end if
end if
$\texttt{Phase}_p \leftarrow \texttt{Phase}_p + 1$

$\texttt{A}_4$ :: $(\texttt{Phase}_p = 4) \wedge (\mathcal{PIF}.\texttt{Request}_p = \texttt{Done})$ →
$\texttt{Phase}_p \leftarrow 0$

$\texttt{A}_5$ :: **receive-brd**⟨ASK⟩ **from** q →
if $\texttt{Value}_p = q$ **then**
  $\mathcal{PIF}.\texttt{F-Mes}_p[q] \leftarrow \texttt{YES}$
else
  $\mathcal{PIF}.\texttt{F-Mes}_p[q] \leftarrow \texttt{NO}$
end if

$\texttt{A}_6$ :: **receive-brd**⟨EXIT⟩ **from** q →
$\texttt{Phase}_p \leftarrow 0$
$\mathcal{PIF}.\texttt{F-Mes}_p[q] \leftarrow \texttt{OK}$

$\texttt{A}_7$ :: **receive-brd**⟨EXITCS⟩ **from** q →
if $(\texttt{Value}_p = q)$ **then**
  $\texttt{Value}_p \leftarrow (\texttt{Value}_p + 1) \bmod (n+1)$
end if
$\mathcal{PIF}.\texttt{F-Mes}_p[q] \leftarrow \texttt{OK}$

$\texttt{A}_8$ :: **receive-fck**⟨YES⟩ **from** q →
$\texttt{Privileges}_p[q] \leftarrow true$

$\texttt{A}_9$ :: **receive-fck**⟨NO⟩ **from** q →
$\texttt{Privileges}_p[q] \leftarrow false$

$\texttt{A}_{10}$ :: **receive-fck**⟨OK⟩ **from** q →
/* do nothing */

Lemma 8 *Let p be a process. Starting from any configuration, after p executes $\mathtt{A}_0$, p enters in the $\langle\mathtt{CS}\rangle$ only if the* leader *favours p until p releases the $\langle\mathtt{CS}\rangle$.*

Proof. By checking all the actions of Algorithm 3, we can remark that after p executes $\mathtt{A}_0$, p must execute the four actions $\mathtt{A}_0$, $\mathtt{A}_1$, $\mathtt{A}_2$, and $\mathtt{A}_3$ successively to enter in the $\langle\mathtt{CS}\rangle$ (in $\mathtt{A}_3$). Moreover, p executes a complete $\mathcal{IDL}$-computation between $\mathtt{A}_0$ and $\mathtt{A}_1$. So:

(1) $\mathcal{IDL}.\mathtt{minID}_p = ID_{\mathcal{L}}$ when p executes $\mathtt{A}_3$ (by Theorem 3, $\mathcal{IDL}$ is snap-stabilizing for Specification 2).

(2) Also, from the configuration where p executes $\mathtt{A}_1$, all messages in the channels from and to p have been sent after $\mathcal{IDL}$ starts at p in Action $\mathtt{A}_0$ (Property 1, page 16).

Let us now study the two following cases:

- $p = \mathcal{L}$. In this case, when p executes $\mathtt{A}_3$, p must satisfy $\mathtt{Value}_p = \mathtt{Value}_{\mathcal{L}} = 0$ to enter in the $\langle\mathtt{CS}\rangle$ by (1). This means that $\mathcal{L}$ favours p (actually itself) when p enters in the $\langle\mathtt{CS}\rangle$. Morever, as the execution of $\mathtt{A}_3$ is atomic, $\mathcal{L}$ favours p until p releases the $\langle\mathtt{CS}\rangle$ and the lemma holds in this case.
- $p \neq \mathcal{L}$. In this case, when p executes $\mathtt{A}_3$, p satisfies $\mathcal{IDL}.\mathtt{minID}_p = ID_{\mathcal{L}}$ by (1). So, p executes the $\langle\mathtt{CS}\rangle$ only if $\exists q \in [1 \ldots n-1]$ such that $\mathcal{IDL}.\mathtt{ID\text{-}Tab}_p[q] = ID_{\mathcal{L}} \wedge \mathtt{Privileges}_p[q] = \mathit{true}$ (see Predicate $Winner(p)$). To that goal, p must receive a feedback message YES from $\mathcal{L}$ during the PIF of the message ASK started in Action $\mathtt{A}_1$. Now, $\mathcal{L}$ sends such a feedback to p only if $\mathtt{Value}_{\mathcal{L}} = p$ when the "**receive-brd**$\langle\mathtt{ASK}\rangle$ from p" event occurs at $\mathcal{L}$ (see Action $\mathtt{A}_5$). Also, since $\mathcal{L}$ satisfies $\mathtt{Value}_{\mathcal{L}} = p$, $\mathcal{L}$ updates $\mathtt{Value}_p$ only after receiving an EXITCS message from p (see Action $\mathtt{A}_7$). 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 $\langle\mathtt{CS}\rangle$ (see Action $\mathtt{A}_3$). Hence, $\mathcal{L}$ favours p until p releases the $\langle\mathtt{CS}\rangle$ and the lemma holds in this case.

□

Lemma 9 *Let p be a process. Starting from any configuration, if p enters in the $\langle\mathtt{CS}\rangle$ after executing $\mathtt{A}_0$, then it executes the $\langle\mathtt{CS}\rangle$ alone.*

Proof. Assume, for the purpose of contradiction, that p enters in the $\langle\mathtt{CS}\rangle$ after executing $\mathtt{A}_0$ but executes the $\langle\mathtt{CS}\rangle$ concurrently with another process q. Then, q also executes Action $\mathtt{A}_0$ before executing the $\langle\mathtt{CS}\rangle$ by Lemma 7. By Lemma 8, we have the two following property:

- $\mathcal{L}$ favours p during the whole period where p executes the $\langle\mathtt{CS}\rangle$.
- $\mathcal{L}$ favours q during the whole period where q executes the $\langle\mathtt{CS}\rangle$.

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

Corollary 1 (Correctness) *Starting from any configuration, if a requesting process enters in the $\langle\mathtt{CS}\rangle$, then it executes the $\langle\mathtt{CS}\rangle$ alone.*

We now show that, despite the arbitrary initial configuration, any execution of $\mathcal{ME}$ always satisfies the start property of Specification 3.

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

Proof. Consider the two following cases:

- *"***receive-brd**$\langle$`EXIT`$\rangle$*" events occur at p infinitely often.* Then, each time such an event occurs at p, p switches to Phase 0 (see $\mathtt{A}_6$). So, the lemma holds in this case.
- *Only a finite number of "***receive-brd**$\langle$`EXIT`$\rangle$*" events occurs at p.* In this case, p eventually reaches a configuration from which it no more executes Action $\mathtt{A}_6$. From this configuration, $\mathtt{Phase}_p$ can only be incremented modulus 5 and depending of the value of $\mathtt{Phase}_p$, we have the following possibilities:
  - $\mathtt{Phase}_p = 0$. In this case, $\mathtt{A}_0$ is continuously enabled at p. Hence, p eventually sets $\mathtt{Phase}_p$ to 1 (see Action $\mathtt{A}_0$).
  - $\mathtt{Phase}_p = i$ *with* $i > 0$. 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 $\mathtt{Phase}_p$ modulus 5.

  Hence, if only a finite number of "**receive-brd**$\langle$`EXIT`$\rangle$" events occurs at p, then $\mathtt{Phase}_p$ is eventually incremented modulus 5 infinitely often, which proves the lemma in this case.

□

Lemma 11 *Starting from any configuration,* $\mathtt{Value}_{\mathcal{L}}$ *is incremented modulus* $n+1$ *infinitely often.*

Proof. Assume, for the purpose of contradiction, that $\mathtt{Value}_{\mathcal{L}}$ is eventually no more incremented 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 "***receive-brd**$\langle$`EXIT`$\rangle$*" events occurs at p.* To that goal, assume, for the purpose of contradiction, that an infinite number of "**receive-brd**$\langle$`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-computation terminates in finite time (termination property of Specification 1, page 9). 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 $q \neq p$, $\mathtt{Value}_{\mathcal{L}} \neq 0$, or (2) $\mathcal{L}$ has feedback `NO` to the PIF of the message `ASK` started by q because $\mathtt{Value}_{\mathcal{L}} = p \neq q$. 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 the message `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 two following cases:

- $p = \mathcal{L}$. Then, $\mathtt{Value}_p = 0$ 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 for Specification 2 (page 16). Hence, p updates $\mathtt{Value}_p$ to 1 when executing $\mathtt{A}_3$ — a contradiction.
- $p \neq \mathcal{L}$. Then, $\mathcal{IDL}.\mathtt{minID}_p = ID_p$ 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 for Specification 2 (page 16). Also, p receives $\mathtt{YES}$ from $\mathcal{L}$ because p executes a complete PIF of the message $\mathtt{ASK}$ between $\mathtt{A}_1$ and $\mathtt{A}_2$ and $\mathcal{PIF}$ is snap-stabilizing for Specification 1 (page 9). Hence, p satisfies the predicate $Winner(p)$ when executing $\mathtt{A}_3$ and, as a consequence, starts a PIF of the message $\mathtt{EXITCS}$ 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 $\mathtt{Value}_{\mathcal{L}}$ — a contradition.

□

Lemma 12 (Start) *Starting from any configuration, any process that requests the* $\langle\mathtt{CS}\rangle$*, enters in the* $\langle\mathtt{CS}\rangle$ *in finite time.*

Proof. Assume, for the purpose of contradiction, that from a configuration γ, a process p requests but never enters in the $\langle\mathtt{CS}\rangle$. Then, Lemma 10 implies that p eventually executes $\mathtt{A}_0$ and after executing $\mathtt{A}_0$, $\mathtt{Request}_p = \mathtt{In}$ holds forever ($\mathtt{Request}_p$ is switched to $\mathtt{Done}$ only after p releases the $\langle\mathtt{CS}\rangle$). From the code of Algorithm 3, we can then deduce that there is two possibilities after p executes $\mathtt{A}_0$:

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

Consider then the two following cases:

- $p = \mathcal{L}$. Then, $\mathtt{Value}_p = 0$ eventually holds forever — a contradiction to Lemma 11.
- $p \neq \mathcal{L}$. In this case, p no more starts any PIF of the message $\mathtt{EXITCS}$. Now, every PIF-computation terminates in finite time (termination property of Specification 1, page 9). Hence, the "**receive-brd**$\langle\mathtt{EXITCS}\rangle$ **from** p" event eventually no more occurs at $\mathcal{L}$. As a consequence, $\mathtt{Value}_{\mathcal{L}}$ eventually no more switches from value p to $(p+1)$ mod $(n+1)$ — a contradiction to Lemma 11.

□

By Corollary 1 and Lemma 12, starting from any configuration, any execution of $\mathcal{ME}$ always satisfies Specification 3. Hence, follows:

Theorem 4 *Protocol* $\mathcal{ME}$ *is snap-stabilizing from Specification 3.*

5 Conclusion

We addressed the problem of *snap-stabilization* in message-passing systems and presented matching negative and positive results. On the negative side, we show 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 show 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, IDs-Learning, and mutual exclusion problem in a fully-connected network.

On the theoretical side, it is worth investigating if the results presented in this paper could be extended to more general networks, *e.g.* with general topologies, and/or where nodes are subject to permanent *aka* crash failures. On the practical side, our result implies the possibility of implementing snap-stabilizing protocols on real networks, and actually implementing them is a future challenge.

References

[1] Y Afek and A Bremler. Self-stabilizing unidirectional network algorithms by power supply. *Chicago Journal of Theoretical Computer Science*, 1998:Article 3, 1998.

[2] Y Afek and GM Brown. Self-stabilization over unreliable communication media. *Distributed Computing*, 7):27–34, 1993.

[3] Bowen Alpern and Fred B. Schneider. Defining liveness. *Inf. Process. Lett.*, 21(4):181–185, 1985.

[4] Bowen Alpern and Fred B. Schneider. Recognizing safety and liveness. *Distributed Computing*, 2(3):117–126, 1987.

[5] Anish Arora and Mikhail Nesterenko. Unifying stabilization and termination in message-passing systems. *Distributed Computing*, 17(3):279–290, 2005.

[6] B Awerbuch, B Patt-Shamir, and G Varghese. Self-stabilization by local checking and correction. In *FOCS91 Proceedings of the 31st Annual IEEE Symposium on Foundations of Computer Science*, pages 268–277, 1991.

[7] Baruch Awerbuch, Shay Kutten, Yishay Mansour, Boaz Patt-Shamir, and George Varghese. A time-optimal self-stabilizing synchronizer using a phase clock. *IEEE Trans. Dependable Sec. Comput.*, 4(3):180–190, 2007.

[8] Doina Bein, Ajoy Kumar Datta, and Vincent Villain. Snap-stabilizing optimal binary search tree. In Ted Herman and Sébastien Tixeuil, editors, *Self-Stabilizing Systems*, volume 3764 of *Lecture Notes in Computer Science*, pages 1–17. Springer, 2005.

[9] L Blin, A Cournier, and V Villain. An improved snap-stabilizing PIF algorithm. In *DSN SSS'03 Workshop: Sixth Symposium on Self-Stabilizing Systems (SSS'03)*, pages 199–214. LNCS 2704, 2003.

[10] A Bui, AK Datta, F Petit, and V Villain. Snap-stabilizing PIF algorithm in tree networks without sense of direction. In *SIROCCO'99, The 6th International Colloquium On Structural Information and Communication Complexity Proceedings*, pages 32–46. Carleton University Press, 1999.

[11] A Bui, AK Datta, F Petit, and V Villain. State-optimal snap-stabilizing PIF in tree networks. In *Proceedings of the Fourth Workshop on Self-Stabilizing Systems*, pages 78–85, Austin, Texas, USA, June 1999. IEEE Computer Society Press.

[12] Alain Bui, Ajoy Kumar Datta, Franck Petit, and Vincent Villain. Snap-stabilization and pif in tree networks. *Distributed Computing*, 20(1):3–19, 2007.

[13] EJH Chang. Echo algorithms: depth parallel operations on general graphs. *IEEE Transactions on Software Engineering*, SE-8:391–401, 1982.

[14] A Cournier, AK Datta, F Petit, and V Villain. Snap-stabilizing PIF algorithm in arbitrary rooted networks. In *22st International Conference on Distributed Computing Systems (ICDCS-22)*, pages 199–206. IEEE Computer Society Press, 2002.

[15] A Cournier, AK Datta, F Petit, and V Villain. Enabling snap-stabilization. In *23th International Conference on Distributed Computing Systems (ICDCS 2003)*, pages 12–19, Providence, Rhode Island USA, May 19-22 2003. IEEE Computer Society Press.

[16] A Cournier, S Devismes, F Petit, and V Villain. Snap-Stabilizing Depth-First Search on Arbitrary Networks. *The Computer Journal*, 49(3):268–280, 2006.

[17] A Cournier, S Devismes, and V Villain. Snap-stabilizing detection of cutsets. In *HIPC 2005, 12th Annual IEEE Conference on High Performance Computing*, pages 488–497. LNCS 3769, 2005.

[18] A Cournier, S Devismes, and V Villain. A snap-stabilizing DFS with a lower space requirement. In *Seventh International Symposium on Self-Stabilizing Systems (SSS'05)*, pages 33–47, Barcelona, Spain, 2005. LNCS 3764.

[19] A Cournier, S Devismes, and V Villain. Snap-stabilizing PIF and useless computations. In *The Twelfth International Conference on Parallel and Distributed Systems (ICPADS'06)*, volume 1, pages 39–46, Minneapolis, USA, 2006. IEEE Computer Society Press P2612.

[20] Alain Cournier, Ajoy Kumar Datta, Franck Petit, and Vincent Villain. Optimal snap-stabilizing pif algorithms in un-oriented trees. *J. High Speed Networks*, 14(2):185–200, 2005.

[21] Alain Cournier, Stéphane Devismes, and Vincent Villain. Light enabling snap-stabilization. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 2007. Under soumission.

[22] Sylvie Delaët, Bertrand Ducourthial, and Sébastien Tixeuil. Self-stabilization with r-operators revisited. *Journal of Aerospace Computing, Information, and Communication*, 2006.

[23] EW Dijkstra. Self stabilizing systems in spite of distributed control. *Communications of the Association of the Computing Machinery*, 17:643–644, 1974.

[24] Shlomi Dolev and Ted Herman. Superstabilizing protocols for dynamic distributed systems. *Chicago Journal of Theoretical Computer Science*, 1997.

[25] S Ghosh, A Gupta, T Herman, and SV Pemmaraju. Fault-containing self-stabilizing distributed protocols. Technical Report 00-01, Department of Computer Science, University of Iowa, 2000.

[26] Mohamed G. Gouda and Nicholas J. Multari. Stabilizing communication protocols. *IEEE Trans. Computers*, 40(4):448–458, 1991.

[27] Rodney R. Howell, Mikhail Nesterenko, and Masaaki Mizuno. Finite-state self-stabilizing protocols in message-passing systems. *J. Parallel Distrib. Comput.*, 62(5):792–817, 2002.

[28] Colette Johnen, Luc Alima, Ajoy K. Datta, and Sébastien Tixeuil. Optimal snap-stabilizing neighborhood synchronizer in tree networks. *Parallel Processing Letters*, 12(3-4):327–340, 2002.

[29] S Katz and KJ Perry. Self-stabilizing extensions for message-passing systems. *Distributed Computing*, 7:17–26, 1993.

[30] Franck Petit and Vincent Villain. Optimal snap-stabilizing depth-first token circulation in tree networks. *J. Parallel Distrib. Comput.*, 67(1):1–12, 2007.

[31] A Segall. Distributed network protocols. *IEEE Transactions on Information Theory*, IT-29:23–35, 1983.

[32] G Tel. *Introduction to distributed algorithms.* Cambridge University Press, Cambridge, UK, Second edition 2001.

[33] George Varghese. Self-stabilization by counter flushing. *SIAM J. Comput.*, 30(2):486–510, 2000.

Unité de recherche INRIA Futurs
Parc Club Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 ORSAY Cedex (France)

Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)
Unité de recherche INRIA Sophia Antipolis : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399