Theoretical Computer Science 412 (2011) 4285–4296

Contents lists available at ScienceDirect

Theoretical Computer Science

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

How to improve snap-stabilizing point-to-point communication space complexity?☆

Alain Cournier [a], Swan Dubois [b,*], Vincent Villain [a]

[a] MIS Laboratory, Université de Picardie Jules Verne, France

[b] LIP6 - UMR 7606 Université Pierre et Marie Curie - Paris 6/INRIA Rocquencourt, France

ARTICLE INFO

Keywords:
Self-stabilization
Snap-stabilization
Message forwarding

ABSTRACT

A *snap-stabilizing* protocol, starting from any configuration, always behaves according to its specification. In this paper, we are interested in the message forwarding problem in a message-switched network in which the system resources must be managed in order to deliver messages to any processor of the network. To this end, we use the information provided by a routing algorithm. In the context of an arbitrary initialization (due to stabilization), this information may be corrupted. In Cournier et al. (2009) [1], we show that there exist snap-stabilizing algorithms for this problem (in the *state model*). This implies that we can request the system to begin forwarding messages without losses even if routing information is initially corrupted.

In this paper, we propose another snap-stabilizing algorithm for this problem which improves the space complexity of the one in Cournier et al. (2009) [1].

1. Introduction

The quality of a distributed system depends on its *fault-tolerance* properties. Many fault-tolerant schemes have been proposed. For instance, *self-stabilization* [3] allows one to design a system tolerating an arbitrary number of transient faults. A self-stabilizing system, regardless of the initial state of the system, is guaranteed to converge into the intended behavior in finite time. Another paradigm is *snap-stabilization* [4]. A snap-stabilizing protocol guarantees that, starting from any configuration, it always behaves according to its specification. Hence, a snap-stabilizing protocol is a self-stabilizing protocol which stabilizes in 0 time units.

In a distributed system, it is commonly assumed that each processor can exchange messages only with its *neighbors* (*i.e.* processors with which it shares a communication link) but processors may need to exchange messages with *any* processor of the network. To perform this goal, processors have to solve two problems: the determination of the path which messages have to follow in the network to reach their destinations (called the *routing problem*) and the management of network resources in order to forward messages (called the *message forwarding problem*). These two problems have received great attention in literature. The routing problem is studied, for example in [5–7], and self-stabilizing approaches can be found in [8–10]. The forwarding problem has also been well studied, see [11–16]. As far we know, only [1,2] deal with this problem using a stabilizing approach.

Informally, the goal of forwarding is to design a protocol which allows all processors of the network to send messages to any destination of the network (knowing that a routing algorithm computes the path that messages have to follow to

☆ A preliminary version of this work appears in the proceedings of the 11th International Symposium on Stabilization, Safety and Security of Distributed Systems (SSS'09), (Cournier et al. 2009 [2]).

* Corresponding author. Tel.: +33 1 44 27 73 46.
E-mail addresses: alain.cournier@u-picardie.fr (A. Cournier), swan.dubois@lip6.fr (S. Dubois), vincent.villain@u-picardie.fr (V. Villain).

0304-3975/$ – see front matter © 2010 Elsevier B.V. All rights reserved.
doi:10.1016/j.tcs.2010.11.014

reach their destinations). The problems arise when messages traveling through a message-switched network [17] must be stored in each processor of their path before being forwarded to the next processor on this path. This temporary storage of messages is performed with reserved memory spaces called buffers. Obviously, each processor of the network reserves only a finite number of buffers for the message forwarding. So, the problem of bounded resources management exposes the network to deadlocks and livelocks if no control is performed. In this paper, we focus on designing a protocol which deals with the message forwarding problem using a snap-stabilizing approach. The goal is to allow the system to forward messages regardless of the state of the routing tables. Obviously, we need the routing tables to be able to repair themselves within finite time. So, we assume the existence of a self-stabilizing protocol to initially compute routing tables [8–10].

In the following, a message sent out by a processor is called valid. An invalid message is present in the initial configuration. We propose a specification of the message forwarding problem where duplicates (*i.e.* the same message arrives many times at its destination while it has been sent out only once) are forbidden:

Specification 1 ($\mathcal{SP}$)**.** *Specification of the message forwarding problem.*

- *Any message can be sent out in finite time.*
- *Any valid message is delivered to its destination once and only once in finite time.*

In [1], we show that it is possible to transform the forwarding algorithm of [12] into a snap-stabilizing one without any significant excess cost (with respect to time of forwarding and amount of memory per processor). But this algorithm needs $\Theta(n)$ buffers per processor (where n is the number of processors of the network). The scope of this paper is the improvement of this space complexity. We achieve this goal by providing a snap-stabilizing forwarding algorithm which requires $\Theta(D)$ buffers per processor (where D is the diameter of the network). Since n and D are close values in the worst case, this improvement is quite useful from a theoretical point of view. However, we believe that it could be very interesting from a practical point of view. Indeed, practical networks have in general a diameter which is significantly smaller than the number of nodes (for example, [18] shows that in 2000 the diameter of the Internet is near to 6 although it had near to 14,000 nodes).

However, we show in this paper that this space improvement leads to an increase of time complexity in the worst case with respect to the protocol of [1]. But we are happy to say that the amortized complexity (a measure that give us the mean time of delivery of messages) is the same for both protocols. That means that, from a practical point of view, both protocols have the same cost in time.

The remainder of this paper is organized as follows. We present first our model (Section 2). We give and prove our solution in the state model, respectively, in Sections 3 and 4. Next, we study the time complexities of our solution in Section 5. Finally, we give our conclusions in Section 6.

2. Preliminaries

We consider a network to be an undirected connected graph $G = (V, E)$, where V is a set of processors and E is the set of bidirectional asynchronous communication links. In the network, a communication link (p, q) exists if and only if p and q are neighbors. We assume that the labels of neighbors of p are stored in the set N_p. We also use the following notations: respectively, n is the number of processors, Δ the maximal degree, and D the diameter of the network. If p and q are two processors of the network, we denote by $dist(p, q)$ the length of the shortest path between p and q (that is, the distance between p and q). In the following, we assume that each processor has an identity which is unique on the network. Moreover, we assume that all processors know the set I of all identities of the network.

2.1. State model

We consider the classical local shared memory model of computation (see [17]) in which communications between neighbors are modeled by direct reading of variables instead of exchange of messages. In this model, the program of every processor consists of a set of shared variables (henceforth, referred to as variables) and a finite set of actions. A processor can write to its own variables only, and read its own variables and those of its neighbors. Each action is of the form: $< label >::< guard > \longrightarrow < statement >$. The guard of an action in the program of p is a Boolean expression involving variables of p and its neighbors. The statement of an action of p updates one or more variables of p. An action can be executed only if its guard is satisfied. The state of a processor is defined by the value of its variables. The state of a system is the Cartesian product of the states of all processors. We refer to the state of a processor and the system as a (local) state and (global) configuration, respectively. We denote $\mathcal{C}$ as the set of all configurations of the system. Let $\gamma \in \mathcal{C}$ and A be an action of p ($p \in V$). A is enabled at p in γ if and only if the guard of A is satisfied by p in γ. Processor p is said to be enabled in γ if and only if at least one action is enabled at p in γ. Let a distributed protocol $\mathcal{P}$ be a collection of binary transition relations denoted by $\rightarrow$, on $\mathcal{C}$. An execution of a protocol $\mathcal{P}$ is a maximal sequence of configurations $\Gamma = (\gamma_0, \gamma_1, \ldots, \gamma_i, \gamma_{i+1}, \ldots)$ such that, $\forall i \geq 0$, $\gamma_i \rightarrow \gamma_{i+1}$ (called a step) if γ_{i+1} exists, else γ_i is a terminal configuration. *Maximality* means that the sequence is either finite (and no action of $\mathcal{P}$ is enabled in the terminal configuration) or infinite. All executions considered here are assumed to be maximal. $\mathcal{E}$ is the set of all executions of $\mathcal{P}$. As we already said, each execution is decomposed into steps. Each step is split into three sequential phases atomically executed: (i) every processor evaluates its guards, (ii) a daemon

chooses some enabled processors, (iii) each chosen processor executes its enabled action. When the three phases are done, the next step begins. A daemon can be defined in terms of fairness and distribution. There exists several kinds of fairness assumption. Here, we use only the weak fairness assumption, meaning that we assume that every continuously enabled processor is eventually chosen by the daemon. We assume that the daemon is distributed, meaning that, at each step, if one or more processors are enabled, then the daemon chooses at least one of these processors to execute an action. We consider that any processor p is neutralized in the step $\gamma_i \rightarrow \gamma_{i+1}$ if p was enabled in γ_i and not enabled in γ_{i+1}, but did not execute any action in $\gamma_i \rightarrow \gamma_{i+1}$. To compute the time complexity, we use the definition of round [19]. This definition captures the execution rate of the slowest processor in any execution. The first round of $\Gamma \in \mathcal{E}$, noted Γ', is the minimal prefix of Γ containing the execution of one action or the neutralization of every enabled processor from the initial configuration. Let Γ'' be the suffix of Γ such that $\Gamma = \Gamma'\Gamma''$. The second round of Γ is the first round of Γ'', and so on.

2.2. Message-switched network

Today, most computer networks use a variant of the *message-switching* method (also called *store-and-forward* method). It is why we have chosen to work with this switching model. In this section, we briefly present this method (see [17] for a detailed presentation). The model assumes that each buffer can store a whole message and that each message needs only one buffer to be stored. The switching method is modeled by four types of move:

1. Generation: when a processor is ready to send out a new message, it "creates" a new message in one of its empty buffers. We assume that the network may allow this move as soon as at least one buffer of the processor is empty.
2. Forwarding: a message m is forwarded (copied) from a processor p to an empty buffer of the next processor q on its route (determined by the routing algorithm). We assume that the network may allow this move as soon as at least one buffer of the processor q is empty.
3. Consumption: A message m occupying a buffer in its destination is erased and delivered to this processor. We assume that the network may always allow this move.
4. Erasing: a message m is erased from a buffer. We assume that the network may allow this move as soon as the message has been forwarded at least one time or delivered to its destination.

2.3. Stabilization

Definition 1 (*Self-Stabilization [3]*)**.** Let $\mathcal{T}$ be a task and $\mathcal{S}_{\mathcal{T}}$ a specification of $\mathcal{T}$. A protocol $\mathcal{P}$ is self-stabilizing for $\mathcal{S}_{\mathcal{T}}$ if and only if $\forall \Gamma \in \mathcal{E}$, there exists a finite prefix $\Gamma' = (\gamma_0, \gamma_1, \ldots, \gamma_l)$ of Γ such that any execution starting from γ_l satisfies $\mathcal{S}_{\mathcal{T}}$.

Definition 2 (*Snap-Stabilization [4]*)**.** Let $\mathcal{T}$ be a task and $\mathcal{S}_{\mathcal{T}}$ a specification of $\mathcal{T}$. A protocol $\mathcal{P}$ is snap-stabilizing for $\mathcal{S}_{\mathcal{T}}$ if and only if $\forall \Gamma \in \mathcal{E}$, Γ satisfies $\mathcal{S}_{\mathcal{T}}$.

Definition 2 has the two following consequences. We can see that a snap-stabilizing protocol for $\mathcal{S}_{\mathcal{T}}$ is a self-stabilizing protocol for $\mathcal{S}_{\mathcal{T}}$ with a stabilization time of 0 time units. A common method used to prove that a protocol is snap-stabilizing is to distinguish an action as a "starting action" (*i.e.* an action which initiates a computation) and to prove the following properties for every execution of the protocol: if a processor requests it, the computation is initiated by a starting action in a finite time and every computation initiated by a starting action satisfies the specification of the task. We will use these two remarks to prove snap-stabilization of our protocol.

3. Description of the proposed protocol

To simplify the presentation, we assume that the routing algorithm induces only minimal paths in number of edges. We have seen in Section 2.2 that, by default, the network always allows message moves between buffers. But, if we do not make any control on these moves, the network may reach unacceptable situations such as deadlocks, livelocks or message losses. If such situations appear, specifications of message forwarding are not respected. Now, we briefly present solutions given by the literature in the case when the routing tables are correct in the initial configuration. We must define an algorithm which permits or forbids various moves in the network (depending on the current occupation of buffers) in order to prevent the network reaching a deadlock. Such algorithms are called deadlock-free controllers (see [17] for a detailed description). Livelocks can be avoided by fairness assumptions on the controller for the generation and forwarding of messages. Message loss is avoided using identifier on messages (for example, the concatenation of the identity of source and a two-value flag). [12] introduces a generic method to design deadlock-free controllers. The key idea is to restrict moves of messages along edges of an oriented graph BG (called buffer graph) defined on the network buffers. The authors show that cycles on BG can lead to deadlocks and that, if BG is acyclic, they can define a deadlock-free controller on this buffer graph. The main idea in [1] is to adapt the graph buffer of [12] to obtain a snap-stabilizing forwarding protocol.

In this paper, we are interested in another buffer graph introduced in [12]. Each processor has $D + 1$ buffers ranked from 1 to $D + 1$. New messages are always generated in the buffer of rank 1 of the sender processor. When a message occupying a buffer of rank i is forwarded to a neighbor q, it is always copied in the buffer of rank $i + 1$ of q. It is easy to see that this graph is acyclic since messages always "ascend" the buffer rank (the reader can find an example of such a graph in Fig. 1). We need

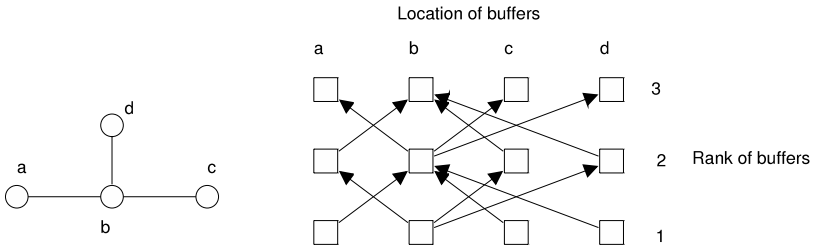

Fig. 1. Example of our buffer graph (on the right) for the network on the left.

$D+1$ buffers per processor since, in the worst case, a message is forwarded at most D times between its generation and its consumption.

Our idea is to adapt this scheme in order to tolerate transient faults. To that goal, we assume that a self-stabilizing silent algorithm $\mathcal{A}$ computes routing tables (see e.g. [8–10]). Our message forwarding protocol is provided in Algorithm 1 ($\mathcal{SSMFP}$ means $\mathcal{S}$nap-$\mathcal{S}$tabilizing $\mathcal{M}$essage $\mathcal{F}$orwarding $\mathcal{P}$rotocol). Moreover, we assume that $\mathcal{A}$ has priority over $\mathcal{SSMFP}$ (*i.e.* a processor which has enabled actions for both algorithms always chooses the action of $\mathcal{A}$). This guarantees us that routing tables are correct and stable within finite time. We assume that $\mathcal{SSMFP}$ can have access to the routing table via a function, called $nextHop_p(d)$. This function returns the identity of the neighbor of p to which p must forward messages of destination d. Our idea is as follows: we allow the erasing of a message only if we have evidence that the message has been delivered to its destination. In this goal, we use a scheme with acknowledgment which guarantees the reception of the message. More precisely, we associate to each copy of the message a type which has 3 values: E (Emission), A (Acknowledgment) and F (Fail). Forwarding of a valid message m (to destination d) follows the above scheme:

1. Generation of m with type E in a buffer of rank 1 by $(\boldsymbol{R_1})$.
2. Forwarding[1] of m with type E without any erasing by $(\boldsymbol{R_8})$ or $(\boldsymbol{R_{12}})$.
3. If m reaches d:
   (a) It is delivered and the copy of m takes type A by $(\boldsymbol{R_4})$ or $(\boldsymbol{R_{10}})$.
   (b) Type A is spread to the sink following the incoming path by $(\boldsymbol{R_7})$.
   (c) Buffers are allowed to free themselves once the type A is propagated to the previous buffer on the path by $(\boldsymbol{R_9})$, $(\boldsymbol{R_{11}})$, or $(\boldsymbol{R_{14}})$.
   (d) The sink erases its copy by $(\boldsymbol{R_3})$ or $(\boldsymbol{R_5})$, thus m is erased.
4. If m reaches a buffer of rank $D+1$ without crossing d:
   (a) The copy of m takes type F by $(\boldsymbol{R_{13}})$.
   (b) Type F is spread to the sink following the incoming path by $(\boldsymbol{R_7})$.
   (c) Buffers are allowed to free themselves once the type F is propagated to the previous buffer on the path by $(\boldsymbol{R_9})$, $(\boldsymbol{R_{11}})$, or $(\boldsymbol{R_{14}})$).
   (d) Then, the sink of m gives the type E to its copy by $(\boldsymbol{R_2})$ or $(\boldsymbol{R_6})$, that begin a new cycle: m is sending once again.

Obviously, it is necessary to take in account invalid messages: we have chosen to let them follow the forwarding scheme and to erase them if they reach Step 4.d (by rules from $(\boldsymbol{R_{15}})$ to $(\boldsymbol{R_{18}})$). The key idea of the snap-stabilization of our algorithm is the following: since a valid message is never erased, it is sent again after the stabilization of routing tables (if it never reaches its destination before) and then it is normally forwarded. To avoid livelocks, we use a fair scheme of selection of processors allowed to forward a message for each buffer. We can manage this fairness by a queue of requesting processors. Finally, we use a specific flag to prevent message loss. It is composed of the identity of the next processor on the path of the message, the identity of the last processor crossed over by the message, the identity of the destination of the message and the type of the message (E, A or F).

We must manage a communication between our algorithm and processors in order to know when a processor has a message to send. We have chosen to create a Boolean shared variable $request_p$ (for any processor p). Processor p can set it at *true* when it is at *false* and when p has a message to send. Otherwise, p must wait until our algorithm sets the shared variable to *false* (when a message is sent out).

4. Proof of the snap-stabilization

In order to simplify the proof, we introduce a second specification of the problem. This specification allows message duplications.

Specification 2 ($\mathcal{SP}'$)**.** *Specification of message forwarding problem allowing duplication.*
- Any message can be send out in finite time.
- Any valid message is delivered to its destination in finite time.

[1] With copy in buffers of increasing rank.

Algorithm 1 $\mathcal{SSMFP}$: protocol for processor p.

Data:
- n, D : natural integers equal respectively to the number of processors and to the diameter of the network.
- $I = \{0, \ldots, n-1\}$: set of processor identities of the network.
- N_p : set of neighbors of p.

Message:
- (m, r, q, d, c) with m useful information of the message, $r \in N_p$ identity of the next processor to cross for the message (when it reaches the node), $q \in N_p$ identity of the last processor crossed over by the message, $d \in I$ identity of the destination of the message, $c \in \{E, A, F\}$ type of the message.

Variables:
- $\forall i \in \{1, \ldots, D+1\}$, $buf_p(i)$: buffer which can contain a message or be empty (denoted by ε)

Input/Output:
- $request_p$: Boolean. The higher layer can set it to "true" when its value is "false" and when there is a waiting message. We consider that this waiting is blocking.
- $nextMes_p$: gives the message waiting in the higher layer.
- $nextDest_p$: gives the destination of $nextMes_p$ if it exists, *null* otherwise.

Procedures:
- $nextHop_p(d)$: neighbor of p computed by the routing for destination d (if $d = p$, we choose arbitrarily $r \in N_p$).
- $\forall i \in \{2, \ldots, D+1\}$, $choice_p(i)$: fairly chooses one of the processors which can send a message in $buf_p(i)$, *i.e.* $choice_p(d)$ satisfies predicate $((choice_p(i) \in N_p) \wedge (buf_{choice_p(i)}(i-1) = (m, p, q, d, E)) \wedge (choice_p(i) \neq d))$. We can manage this fairness with a queue of length $\Delta + 1$ of processors which satisfies the predicate.
- $deliver_p(m)$: delivers the message m to the higher layer of p.

Rules:

/* Rules for the buffer of rank 1 */

/* Generation of messages */

$(\boldsymbol{R_1})$ :: $request_p \wedge (buf_p(1) = \varepsilon) \wedge (nextDest_p = d) \wedge (nextMes_p = m) \wedge (buf_{nextHop_p(d)}(2) \neq (m, r', p, d, c)) \longrightarrow buf_p(1) := (m, nextHop_p(d), r, d, E)$ with $r \in N_p$; $request_p := false$

/* Processing of acknowledgment */

$(\boldsymbol{R_2})$ :: $(buf_p(1) = (m, r, q, d, F)) \wedge (d \neq p) \wedge (buf_r(2) \neq (m, r', p, d, F)) \longrightarrow buf_p(1) := (m, nextHop_p(d), q, d, E)$

$(\boldsymbol{R_3})$ :: $(buf_p(1) = (m, r, q, d, A)) \wedge (d \neq p) \wedge (buf_r(2) \neq (m, r', p, d, A)) \longrightarrow buf_p(1) := \varepsilon$

/* Management of messages which reach their destinations */

$(\boldsymbol{R_4})$ :: $buf_p(1) = (m, r, q, p, E) \longrightarrow deliver_p(m)$; $buf_p(1) := (m, r, q, p, A)$

$(\boldsymbol{R_5})$ :: $buf_p(1) = (m, r, q, p, A) \longrightarrow buf_p(1) := \varepsilon$

$(\boldsymbol{R_6})$ :: $buf_p(1) = (m, r, q, p, F) \longrightarrow buf_p(1) := (m, r, q, p, E)$

In this section, we prove that $\mathcal{SSMFP}$ is a snap-stabilizing message forwarding protocol for specification $\mathcal{SP}$. For that, we prove successively that:

1. Copies of the same message have a particular structure (Definitions 3 and 4). Then, we prove some properties of the behavior of these structures under $\mathcal{SSMFP}$ (Lemmas 1–4).
2. $\mathcal{SSMFP}$ is a snap-stabilizing message forwarding protocol for specification $\mathcal{SP}'$ if the routing tables are correct in the initial configuration (Lemmas 5–7 and Proposition 1).
3. $\mathcal{SSMFP}$ is a self-stabilizing message forwarding protocol for specification $\mathcal{SP}'$ even if the routing tables are corrupted in the initial configuration (Proposition 2).
4. $\mathcal{SSMFP}$ is a snap-stabilizing message forwarding protocol for specification $\mathcal{SP}$ even if the routing tables are corrupted in the initial configuration (Lemma 8, 9 and Theorem 1).

In this proof, we consider that the notion of message is different from the notion of useful information. This implies that two messages with the same useful information sent by the same processor are considered as two different messages. We must prove that the algorithm does not lose one of them due to the use of the flag.

4.1. Preliminaries

Firstly, we define a particular structure of messages and we study the behavior of this structure under $\mathcal{SSMFP}$.

Let γ be a configuration of the network. We say that a message m exists in γ if at least one buffer contains m in γ. If we observe the copies of any existing message m in any configuration, we can see that these copies can be always organized as directed sub-chains of the buffer graph with the following properties (i) only the last buffer can be located on the destination of m, (ii) the content (pointer to the next and to the previous processor) of each buffer is consistent with the one of the previous and the following buffer on the subchain and (iii) first buffers are of type E and last buffers are all of type A or F. We call these particular structures caterpillars. Here is the formal definition.

End of Algorithm 1:

/* Rule for buffers of rank 1 to D : propagation of acknowledgment */

$(\boldsymbol{R_7})$ :: $\exists i \in \{1,\ldots,D\}, ((buf_p(i) = (m,r,q,d,E)) \wedge (p \neq d) \wedge (buf_r(i+1) = (m,r',p,d,c)) \wedge (c \in \{F,A\})) \longrightarrow buf_p(i) := (m,r,q,d,c)$

/* Rules for buffers of rank 2 to D */

/* Forwarding of messages */

$(\boldsymbol{R_8})$ :: $\exists i \in \{2,\ldots,D\}, ((buf_p(i) = \varepsilon) \wedge (choice_p(i) = s) \wedge (buf_s(i-1) = (m,p,q,d,E)) \wedge (buf_{nextHop_p(d)}(i+1) \neq (m,r,p,d,c))) \longrightarrow buf_p(i) := (m, nextHop_p(d), s, d, E)$ /* Erasing of messages for which acknowledgment has been forwarded */

$(\boldsymbol{R_9})$ :: $\exists i \in \{2,\ldots,D\}, ((buf_p(i) = (m,r,q,d,c)) \wedge (c \in \{F,A\}) \wedge (d \neq p) \wedge (buf_q(i-1) = (m,p,q',d,c)) \wedge (buf_r(i+1) \neq (m,r',p,d,c)) \longrightarrow buf_p(i) := \varepsilon$

/* Rules for buffers of rank 2 to $D+1$ */

/* Consumption of a message and generation of the acknowledgment A */

$(\boldsymbol{R_{10}})$ :: $\exists i \in \{2,\ldots,D+1\}, buf_p(i) = (m,r,q,p,E) \longrightarrow deliver_p(m);\ buf_p(i) := (m,r,q,p,A)$

/* Erasing of messages for p for which acknowledgment has been forwarded */

$(\boldsymbol{R_{11}})$ :: $\exists i \in \{2,\ldots,D+1\}, ((buf_p(i) = (m,r,q,p,c)) \wedge (c \in \{F,A\}) \wedge (buf_q(i-1) = (m,p,q',p,c))) \longrightarrow buf_p(i) := \varepsilon$

/* Rules for the buffer of rank $D+1$ */

/* Forwarding of messages */

$(\boldsymbol{R_{12}})$ :: $(buf_p(D+1) = \varepsilon) \wedge (choice_p(D+1) = s) \wedge (buf_s(D) = (m,p,q,d,E)) \longrightarrow buf_p(D+1) := (m, nextHop_p(d), s, d, E)$

/* Generation of the acknowledgment F */

$(\boldsymbol{R_{13}})$ :: $(buf_p(D+1) = (m,r,q,d,E)) \wedge (d \neq p) \longrightarrow buf_p(D+1) := (m,r,q,d,F)$

/* Erasing of messages for which the acknowledgment has been forwarded */

$(\boldsymbol{R_{14}})$ :: $(buf_p(D+1) = (m,r,q,d,c)) \wedge (c \in \{F,A\}) \wedge (d \neq p) \wedge (buf_q(D) = (m,p,q',d,c)) \longrightarrow buf_p(D+1) := \varepsilon$

/* Correction rules: erasing of tail of abnormal caterpillars of type F, A */

$(\boldsymbol{R_{15}})$ :: $\exists i \in \{2,\ldots,D\}, ((buf_p(i) = (m,r,q,d,c)) \wedge (c \in \{F,A\}) \wedge (buf_r(i+1) \neq (m,r',p,d,c)) \wedge (buf_q(i-1) \neq (m,p,q',d,c'))) \longrightarrow buf_p(i) := \varepsilon$

$(\boldsymbol{R_{16}})$ :: $\exists i \in \{2,\ldots,D\}, ((buf_p(i) = (m,r,q,d,c)) \wedge (c \in \{F,A\}) \wedge (buf_r(i+1) \neq (m,r',p,d,c)) \wedge (buf_q(i-1) = (m,p,q',d,c')) \wedge (c' \in \{F,A\}\backslash\{c\} \vee q = d)) \longrightarrow buf_p(i) := \varepsilon$

$(\boldsymbol{R_{17}})$ :: $(buf_p(D+1) = (m,r,q,d,c)) \wedge (c \in \{F,A\}) \wedge (buf_q(D) \neq (m,p,q',d,c')) \longrightarrow buf_p(D+1) := \varepsilon$

$(\boldsymbol{R_{18}})$ :: $(buf_p(D+1) = (m,r,q,d,c)) \wedge (c \in \{F,A\}) \wedge (buf_q(D) = (m,p,q',d,c')) \wedge (c' \in \{F,A\}\backslash\{c\} \vee q = d) \longrightarrow buf_p(D+1) := \varepsilon$

Definition 3 (*Caterpillar of a Message m*)**.** Let m be a message of destination d existing in a configuration γ. We define a caterpillar associated to m (noted C_m) as the longest sequence of buffers $C_m = buf_{p_1}(i) \ldots buf_{p_t}(i+t-1)$ (with $t \geq 1$) which satisfies:

- $\forall j \in \{1,\ldots,t-1\}, p_j \neq d$ and $p_{j+1} \neq p_j$.
- $\forall j \in \{1,\ldots,t\}, buf_{p_j}(i+j-1) = (m, r_j, q_j, d, c_j)$.
- $\forall j \in \{1,\ldots,t-1\}, r_j = p_{j+1}$.
- $\forall j \in \{2,\ldots,t\}, q_j = p_{j-1}$.
- $\exists k \in \{1,\ldots,t+1\}, \begin{cases} \forall j \in \{1,\ldots,k-1\},\ c_j = E \text{ and} \\ (\forall j \in \{k,\ldots,t\},\ c_j = A) \vee (\forall j \in \{k,\ldots,t\},\ c_j = F). \end{cases}$

We call respectively $buf_{p_1}(i)$, $buf_{p_t}(i+t-1)$, and $lg_{C_m} = t$ the tail, the head, and the length of C_m.

We give now some characterization of caterpillars.

Definition 4 (*Characterization of Caterpillar of a Message m*)**.** Let m be a message of destination d in a configuration γ and $C_m = buf_{p_1}(i) \ldots buf_{p_t}(i+t-1)$ $(t \geq 1)$ a caterpillar associated to m. Then,

- C_m is a normal caterpillar if $i = 1$. It is abnormal otherwise ($i \geq 2$).
- C_m is a caterpillar of type E if $\forall j \in \{1,\ldots,t\}, c_j = E$ (*i.e.* $k = t+1$).
- C_m is a caterpillar of type A if $\exists j \in \{1,\ldots,t\}, c_j = A$ (*i.e.* $k < t+1$).
- C_m is a caterpillar of type F if $\exists j \in \{1,\ldots,t\}, c_j = F$ (*i.e.* $k < t+1$).

It is obvious that, for each caterpillar C_m, C_m is either normal or abnormal. In the same way, C_m is of type E, A or F only. The reader can find in Fig. 2 an example for some types of caterpillar.

Lemma 1. *Let γ be a configuration and m be a message of destination d existing in γ. Under a weakly fair daemon, every abnormal caterpillar of type F (resp. A) associated to m disappears in finite time or become a normal caterpillar of type F (resp. A).*

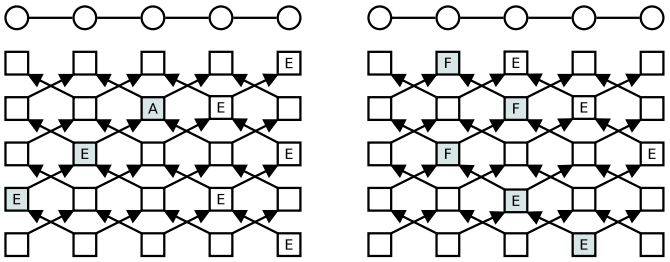

Fig. 2. Examples of caterpillars. On left: abnormal of type *A* (gray) and normal of type *E* (white). On right: normal of type F (gray) and abnormal of type *E* (white).

Proof. Let γ be a configuration of the network. Let m be an existing message (of destination d) in γ. Let $C_m = buf_{p_1}(i) \dots buf_{p_t}(i+t-1)$ ($t \geq 1$ and $i > 1$) be a normal caterpillar of type F or A associated to m. Let c be the type of C_m.

Step 1: By the definition of caterpillar of type c, we have $1 \leq k \leq t$. We can deduce that $i+k-2 < i+t-1 \leq D+1$ and then $(\boldsymbol{R_7})$ is enabled for p_{k-1}. This rule cannot be neutralized since processor p_k is not enabled by a rule affecting its buffer of rank $i+k$. As the daemon is weakly fair, p_{k-1} executes this rule in finite time. We can repeat this reasoning $k-1$ times on the processors $p_{k-1}, \dots, p_1$. Then, we obtain a caterpillar for which all buffers are of type c in finite time.

Step 2: If $t = 1$, we can directly go to Step 4. Otherwise ($t \geq 2$), we must distinguish the following cases:

Case 1: $p_t = d$.

Processor p_t is enabled for rule $(\boldsymbol{R_{11}})$ by the definition of a caterpillar and the fact that all buffers of C_m are of type c. Note that processor p_{t-1} is not enabled. Consequently, this rule remains continuously enabled for p_t. Since the daemon is weakly fair, p_t executes this rule in finite time. Then, $buf_{p_t}(i+t-1)$ is empty in finite time.

Case 2: $p_t \neq d$.

Case 2.1: $i+t-1 = D+1$.

Then, processor p_t is enabled for rule $(\boldsymbol{R_{14}})$ by the definition of a caterpillar and the fact that all buffers of C_m are of type c. Note that processor p_{t-1} is not enabled. Consequently, this rule remains continuously enabled for p_t. Since the daemon is weakly fair, p_t executes this rule in finite time. Then, $buf_{p_t}(i+t-1)$ is empty in finite time.

Case 2.2: $i+t-1 \leq D$.

Assume that $buf_{p_t}(i+t-1) = (m, r, q, d, c)$. Then, processor p_t is enabled for rule $(\boldsymbol{R_9})$ by the definition of a caterpillar and the fact that all buffers of C_m are of type c. Note that processor p_{t-1} is not enabled and that processor r cannot forward a message (m, r', p_t, d, c) in its buffer of rank $i+t$ (since $buf_{p_t}(i+t-1)$ is of type $c \neq E$). Consequently, this rule remains continuously enabled for p_t. Since the daemon is weakly fair, p_t executes this rule in finite time. Then, $buf_{p_t}(i+t-1)$ is empty in finite time.

Step 3: By following a reasoning similar to the one of Case 2.2 (Step 2), we can prove that $p_{t-1}, \dots, p_2$ executes $(\boldsymbol{R_9})$ sequentially in finite time.

Step 4: Then, we obtain a caterpillar of type c of length 1 satisfying $i > 1$. Assume that $buf_{p_1}(i) = (m, r, q, d, c)$. We can distinguish the following cases:

Case 1: $buf_q(i-1) = (m, p_1, q', d, c')$.

Case 1.1: $q = d$.

By the definition of a caterpillar of type c of length 1 and the hypothesis, p_1 is enabled for rule $(\boldsymbol{R_{16}})$ (if $i \leq D$) or $(\boldsymbol{R_{18}})$ (if $i = D+1$). By a reasoning similar to the one of Case 2.2 (Step 2), this rule remains continuously enabled. Since the daemon is weakly fair, p_1 executes this rule in finite time. Consequently, $buf_{p_1}(i)$ becomes empty in finite time. Then, C_m disappears.

Case 1.2: $q \neq d$.

Assume that $c' = E$. Then, $buf_q(i-1)$ belongs to C_m. This contradicts the fact that C_m is of type c. Consequently, $c' \in \{F, A\}$.

If $c' = c$, then the execution of rule $(\boldsymbol{R_7})$ by p_1 leads to the merging of two caterpillars of type c. Then, consider the new caterpillar $C'_m = buf_{p'_1}(i') \dots buf_{p'_{t'}}(i'+t'-1)$ (with $buf_{p'_{t'}}(i'+t'-1) = buf_{p_1}(i)$). If $i' = 1$, then we have a normal caterpillar of type c. Otherwise, we can start the reasoning from the beginning. Note that we are ensured that this reasoning is finite since $1 \leq i' < i$ at each step.

Consider now the case $c' \neq c$. By definition of a caterpillar of type c of length 1 and the hypothesis, p_1 is enabled by rule $(\boldsymbol{R_{16}})$ (if $i \leq D$) or $(\boldsymbol{R_{18}})$ (if $i = D+1$). By a reasoning similar to the one of Case 2.2 (Step 2), this rule remains continuously enabled. Since the daemon is weakly fair, p_1 executes this rule in finite time. Consequently, $buf_{p_1}(i)$ becomes empty in finite time. Then, C_m disappears.

Case 2: $buf_q(i-1) \neq (m, p_1, q', d, c')$.

By definition of a caterpillar of type c of length 1 and the hypothesis, p_1 is enabled by rule $(\boldsymbol{R_{15}})$ (if $i \leq D$) or $(\boldsymbol{R_{17}})$ (if $i = D+1$). By a reasoning similar to the one of Case 2.2 (Step 2), this rule remains continuously enabled. Since the daemon is weakly fair, p_1 executes this rule in finite time. Consequently, $buf_{p_1}(i)$ becomes empty in finite time. Then, C_m disappears.

In all cases, C_m disappears or becomes a normal caterpillar of type c in finite time. That leads us to the proof of the lemma. □

Lemma 2. *Let γ be a configuration and m be a message of destination d existing in γ. Under a weakly fair daemon, every normal caterpillar of type A associated to m disappears in finite time.*

Proof. Let γ be a configuration and m be a message of destination d existing in γ. Let $C_m = buf_{p_1}(1) \dots buf_{p_t}(t)$ $(t \geq 1)$ be a normal caterpillar of type A associated to m. We must distinguish the following cases:
Case 1: $t = 1$.
Case 1.1: $p_1 = d$.
Then, rule $(\boldsymbol{R_5})$ is enabled for p_1. Since the guard of this rule involves only local variables, it remains continuously enabled. Since the daemon is weakly fair, p_1 executes this rule in finite time. Consequently, C_m disappears.
Case 1.2: $p_1 \neq d$.
By the definition of a caterpillar and the hypothesis, p_1 is enabled by rule $(\boldsymbol{R_3})$. By a reasoning similar to the one of Case 2.2 (Step 2) of the proof of Lemma 1, we can prove that this rule remains continuously enabled. Since the daemon is weakly fair, p_1 executes this rule in finite time. Consequently, C_m disappears.
Case 2: $t \geq 2$.
We can apply the reasoning of steps 1, 2, and 3 of the proof of Lemma 1. That leads us to Case 1.2.
In all the cases, C_m disappears in finite time. □

Lemma 3. *Let γ be a configuration and m be a message of destination d existing in γ. Under a weakly fair daemon, every normal caterpillar of type F associated to m becomes a normal caterpillar of type E of length* 1 *in finite time.*

Proof. Let γ be a configuration and m be a message of destination d existing in γ. Let $C_m = buf_{p_1}(1) \dots buf_{p_t}(t)$ $(t \geq 1)$ be a normal caterpillar of type F associated to m. We must distinguish the following cases:
Case 1: $t = 1$.
Case 1.1: $p_1 = d$.
Then, rule $(\boldsymbol{R_6})$ is enabled for p_1. Since the guard of this rule involves only local variables, it remains continuously enabled. Since the daemon is weakly fair, p_1 executes this rule in finite time. Consequently, C_m becomes a caterpillar of type E of length 1.
Case 1.2: $p_1 \neq d$.
By the definition of a caterpillar and the hypothesis, p_1 is enabled by rule $(\boldsymbol{R_2})$. By a reasoning similar to the one of Case 2.2 (Step 2) of the proof of Lemma 1, we can prove that this rule remains continuously enabled. Since the daemon is weakly fair, p_1 executes this rule in finite time. Consequently, C_m becomes a caterpillar of type E of length 1.
Case 2: $t \geq 2$.
We can apply the reasoning of steps 1, 2, and 3 of the proof of Lemma 1. That leads us to Case 1.2.
In all cases, we proved that C_m becomes a caterpillar of type E of length 1 in finite time. This leads us to the lemma. □

Lemma 4. *Let γ be a configuration and m be a message of destination d existing in γ. Under a weakly fair daemon, every caterpillar of type E associated to m becomes a caterpillar of type A or F in finite time.*

Proof. Let γ be a configuration of the network and m be a message (of destination d) existing in γ. Let $C_m = buf_{p_1}(i) \dots buf_{p_t}(i + t - 1)$ $(t \geq 1)$ be a caterpillar of type E associated to m.
We prove this result by a decreasing induction on the rank of the buffer occupied by the head of C_m in γ. Let us define the following property:
$(\boldsymbol{P_l})$: If C_m satisfies $i + t - 1 = l$, then it becomes a caterpillar of type A or F in finite time.
Initialization: We want to prove that $(\boldsymbol{P_{D+1}})$ is true.
Let $C_m = buf_{p_1}(i) \dots buf_{p_t}(i + t - 1)$ $(t \geq 1)$ be a caterpillar of type E associated to m such that $i + t - 1 = D + 1$. We must distinguish the following cases:
Case 1: $p_t = d$.
By hypothesis, processor p_t is enabled for rule $(\boldsymbol{R_{10}})$. Since the guard of this rule involves only local variables, it remains continuously enabled. Since the daemon is weakly fair, p_t executes this rule in finite time. Consequently, $buf_{p_t}(i + t - 1)$ becomes a buffer of type A and C_m becomes a caterpillar of type A in finite time. Then, property $(\boldsymbol{P_{D+1}})$ is satisfied.
Case 2: $p_t \neq d$.
By hypothesis, processor p_t is enabled for rule $(\boldsymbol{R_{13}})$. Since the guard of this rule involves only local variables, it remains continuously enabled. Since the daemon is weakly fair, p_t executes this rule in finite time. Consequently, $buf_{p_t}(i + t - 1)$ becomes a buffer of type F and C_m becomes a caterpillar of type F in finite time. Then, property $(\boldsymbol{P_{D+1}})$ is satisfied.
Induction: Let be $l \leq D$. Assume that $(\boldsymbol{P_{l+1}}) \dots (\boldsymbol{P_{D+1}})$ are satisfied. We want to prove that $(\boldsymbol{P_l})$ is then satisfied.
Let $C_m = buf_{p_1}(i) \dots buf_{p_t}(i + t - 1)$ $(t \geq 1)$ be a caterpillar of type E associated to m such that $i + t - 1 = l < D + 1$. We must distinguish the following cases:
Case 1: $p_t = d$.
Case 1.1: $i + t - 1 = 1$.
By hypothesis, processor p_t is enabled for rule $(\boldsymbol{R_4})$. Since the guard of this rule involves only local variables, it remains continuously enabled. Since the daemon is weakly fair, p_t executes this rule in finite time. Consequently, $buf_{p_t}(i + t - 1)$ becomes a buffer of type A and C_m becomes a caterpillar of type A in finite time. Then, property $(\boldsymbol{P_l})$ is satisfied.

Case 1.2: $2 \leq i + t - 1 \leq D$.
These case is similar to the Case 1 of initialization. Consequently, C_m becomes a caterpillar of type A in finite time. Then, property $(\boldsymbol{P_l})$ is satisfied.
Case 2: $p_t \neq d$.

Assume without loss of generality that $buf_{p_t}(i + t - 1) = (m, r, q, d, E)$. We want to prove that the head of C_m goes up by one buffer in finite time. We must study the following cases:
Case 2.1: $i + t = D + 1$.
Case 2.1.1: If $buf_r(i + t) = \varepsilon$, then processor r is enabled by rule $(\boldsymbol{R_{12}})$. Since processor $choice_r(i + t)$ is not enabled, this rule remains continuously enabled for r. r executes this rule in finite time because the daemon is weakly fair. The result of this execution depends on the value of $choice_r(i + t)$:

1. If $choice_r(i + t) = p_t$, then the head of C_m goes up by one buffer when r executes rule $(\boldsymbol{R_{12}})$.
2. If $choice_r(i + t) = s \neq p_t$, then $buf_r(i + t)$ takes the value (m', r', s, d', c) when r executes rule $(\boldsymbol{R_{12}})$. This leads us to Case 2.1.2.2. Note that the fairness of $choice_r(i + t)$ ensures us that these case can appear only a finite number of times.

Case 2.1.2: Consider now that $buf_r(i + t) = (m', r', q', d', c')$.
Assume that $q' = p_t$ and $m' = m$, then $buf_r(i + t)$ belongs to C_m (the type of C_m is then identical to the one of $buf_r(i + t)$). Consequently, we have a contradiction with the definition of C_m. This implies that $q' \neq p_t$ or $m' \neq m$. Let $C_{m'}$ be the caterpillar to which $buf_r(i + t)$ belongs. Consider the three possible cases:

1. $C_{m'}$ is of type E: we can apply the induction hypothesis to $C_{m'}$ since its head stays in a buffer of rank greater than or equal to $i + t$. Consequently, $C_{m'}$ becomes a caterpillar of type F or A in finite time. That leads us to one of the following cases.
2. $C_{m'}$ is of type A: following Lemmas 1 and 2, $C_{m'}$ disappears in finite time. Then, $buf_r(i + t)$ becomes empty. That leads us to Case 2.1.1.
3. $C_{m'}$ is of type F: following Lemmas 1 and 3, $C_{m'}$ disappears or becomes a caterpillar of type E and length 1 in finite time. In all cases, $buf_r(i + t)$ becomes empty (since $i + t = D + 1 \geq 2$). That leads us to Case 2.1.1.

Case 2.2: $2 \leq i + t \leq D$.
Consider the following cases:
Case 2.2.1: $buf_r(i + t) = \varepsilon$.
Assume without loss of generality that $s = choice_r(i + t)$ and $buf_s(i + t - 1) = (m', r, q', d', c')$. By the construction of rule $(\boldsymbol{R_8})$ and the definition of a caterpillar, r is enabled if and only if $buf_{nextHop_r(d')}(i + t + 1)$ is not the tail of an abnormal caterpillar $C_{m'}$ associated to m'. Let us study the following cases:

1. $C_{m'}$ is of type E: we can apply the induction hypothesis to $C_{m'}$ since its head stays in a buffer of rank greater than or equal to $i + t + 1$. Consequently, $C_{m'}$ becomes a caterpillar of type F or A in finite time. That leads us to one of the following cases.
2. $C_{m'}$ is of type A: following Lemma 1, $C_{m'}$ disappears in finite time. Then, $buf_{nextHop_r(d')}(i + t + 1)$ becomes empty.
3. $C_{m'}$ is of type F: following Lemma 1, $C_{m'}$ disappears in finite time (it cannot become a caterpillar of type E and length 1 since $buf_r(i + t) = \varepsilon$). Consequently, $buf_{nextHop_r(d')}(i + t + 1)$ becomes empty in finite time.

Then, Rule $(\boldsymbol{R_8})$ is enabled for r in finite time. This rule remains continuously enabled since no message of type (m'', r', r, d'', c'') can be copied in $buf_{nextHop_r(d')}(i + t + 1)$ (indeed, the contrary implies that $nextHop_r(d')$ executes rule $(\boldsymbol{R_8})$ whereas $buf_r(i + t) = \varepsilon$). Since the daemon is weakly fair, r executes rule $(\boldsymbol{R_8})$ in finite time. The result of this execution is one of the following:

1. If $choice_r(i + t) = p_t$, then the head of C_m goes up by one buffer when r executes rule $(\boldsymbol{R_8})$.
2. If $choice_r(i + t) = s \neq p_t$, then $buf_r(i + t)$ takes the value (m', r', s, d', c) when r executes rule $(\boldsymbol{R_8})$. This situation is similar to the one of Case 2.1.2 below. Note that the fairness of $choice_r(i + t)$ ensures us that this case can appear only a finite number of times.

Case 2.2.2: If $buf_r(i + t) = (m', r', q', d', c')$, the reasoning is similar to the one of Case 2.1.2. Consequently, that leads us to point 1 in finite time.

In conclusion of Case 2 ($p_t \neq d$), the head of C_m goes up by one buffer in finite time. Then, the induction hypothesis allows us to state that C_m becomes a caterpillar of type F or A in finite time. Hence, $(\boldsymbol{P_l})$ is true. □

4.2. Snap-stabilization when routing tables are correct in the initial configuration

Now, we assume that routing tables are correct in the initial configuration and we prove that $\mathcal{SSMFP}$ is a snap-stabilizing algorithm for $\mathcal{SP}'$.

Lemma 5. *Let γ be a configuration in which routing tables are correct and m be a message of destination d existing in γ. Under a weakly fair daemon, every normal caterpillar of type E associated to m becomes a caterpillar of type A in finite time.*

Proof. Let γ be a configuration of the network in which routing tables are correct and m be a message (of destination d) existing in γ. Let $C_m = buf_{p_1}(1) \dots buf_{p_t}(t)$ $(t \geq 1)$ be a normal caterpillar of type E associated to m.

By Lemma 4, C_m becomes a caterpillar of type A or F in finite time. In the first case, the proof ends here. In the second case (which is possible if $D + 1 - t \leq d(p_t, d)$ in γ), it follows by Lemma 3 that C_m becomes a caterpillar of type E of length 1 in finite time. Then, we have: $C_m = buf_{p_1}(1)$.

Following Lemma 4, C_m becomes a caterpillar of type F or A in finite time. Assume that C_m becomes a caterpillar of type F. This implies that m has been forwarded D times without reaching its destination. This result is absurd since we have by definition that $dist(p_1, d) \leq D$ and we assumed that routing tables are correct and constant. Consequently, C_m becomes a caterpillar of type A in finite time. □

Lemma 6. *If routing tables are correct, any processor can generate a first message (*i.e. *execute* $(\boldsymbol{R_1})$*) in finite time under a weakly fair daemon.*

Proof. Let p be a processor of the network which has a message m (of destination d) to forward. As p has a waiting message, the higher layer puts $request_p = true$ whatever is its value in the initial configuration.

Assume that $buf_p(1)$ already contains a message. Let C_m be the caterpillar which contains this buffer.

If C_m is of type F, C_m becomes a caterpillar of type E in finite time following Lemma 3. That leads us to the case above.

If C_m is of type E, C_m becomes a caterpillar of type A in finite time following Lemma 5. That leads us to the case above.

If C_m is of type A, C_m disappears in finite time following Lemma 2.

In all cases, we obtain that $buf_p(1)$ becomes empty in finite time. It remains empty while p does not execute rule $(\boldsymbol{R_1})$ (since it is the only rule which can put a message in this buffer). In these case, $(\boldsymbol{R_1})$ is enabled for p if and only if $buf_{nextHop_p(d)}(2) \neq (m, r', p, d, c)$.

Assume that this condition is not satisfied. This implies (by definition of a caterpillar) that $buf_{nextHop_p(d)}(2)$ is the tail of an abnormal caterpillar C'_m. Following sequentially Lemmas 1 and 4, C'_m disappears in finite time (note that the merge with $buf_p(1)$ is impossible since this buffer is empty). Moreover, $buf_{nextHop_p(d)}(2)$ cannot be filled by a message of type (m, r', p, d, c) (since $buf_p(1)$ is empty). Consequently, rule $(\boldsymbol{R_1})$ is continuously enabled for processor p. As the daemon is weakly fair, p executes this rule in finite time, which leads to the lemma. □

Lemma 7. *If a message m is generated by* $\mathcal{SSMFP}$ *in a configuration in which routing tables are correct,* $\mathcal{SSMFP}$ *delivers m to its destination in finite time under a weakly fair daemon.*

Proof. The generation of a message m (of destination d) by $\mathcal{SSMFP}$ results from the execution of rule $(\boldsymbol{R_1})$ by the processor which sends m. This rule creates a normal caterpillar of type E associated to m. Following Lemma 5, this caterpillar becomes a caterpillar of type A in finite time. It is due to the execution of rule $(\boldsymbol{R_4})$ or $(\boldsymbol{R_{10}})$ by d. These rules delivers the message to the higher layer of d. This ends the proof. □

Proposition 1. $\mathcal{SSMFP}$ *is a snap-stabilizing message forwarding protocol for* $\mathcal{SP}'$ *if routing tables are correct in the initial configuration.*

Proof. Assume that routing tables are correct in the initial configuration.

In our case, the starting action of $\mathcal{SSMFP}$ is the execution of $(\boldsymbol{R_1})$. Lemma 6 proves that, if a processor p requests to send a message, then the protocol is initiated by at least one starting action on p in finite time.

If we consider that $(\boldsymbol{R_1})$ has been executed at least one time, we can prove that: the first property of $\mathcal{SP}'$ is always satisfied (following Lemma 6 and the fact that the waiting for the sending of new messages is blocking) and the second property of $\mathcal{SP}'$ is always satisfied (following Lemma 7). Consequently, the protocol is executed according to $\mathcal{SP}'$ after the execution of the first starting action.

These two properties shows us that $\mathcal{SSMFP}$ is a snap-stabilizing message forwarding protocol for specification $\mathcal{SP}'$. □

4.3. Self-stabilization

Now, we assume that routing tables are corrupted in the initial configurations and we prove that $\mathcal{SSMFP}$ is a self-stabilizing algorithm for specification $\mathcal{SP}'$.

Proposition 2. $\mathcal{SSMFP}$ *is a self-stabilizing message forwarding protocol for* $\mathcal{SP}'$ *even if routing tables are corrupted in the initial configuration when* $\mathcal{A}$ *runs simultaneously.*

Proof. Recall that $\mathcal{A}$ is a self-stabilizing silent algorithm for computing routing tables running simultaneously to $\mathcal{SSMFP}$. Moreover, we assumed that $\mathcal{A}$ has priority over $\mathcal{SSMFP}$ (*i.e.* a processor which has enabled actions for both algorithms always chooses the action of $\mathcal{A}$). This guarantees us that routing tables are correct and constant in finite time regardless of their initial states.

By Proposition 1, $\mathcal{SSMFP}$ is a snap-stabilizing message forwarding protocol for specification $\mathcal{SP}'$ when it starts from a such configuration. Consequently, we can conclude on the proposition. □

4.4. Snap-stabilization

We still assume that routing tables are corrupted in the initial configuration and we prove that $\mathcal{SSMFP}$ is a snap-stabilizing algorithm for specification $\mathcal{SP}$.

Lemma 8. *Under a weakly fair daemon, $\mathcal{SSMFP}$ does not delete a valid message without delivering it to its destination even if $\mathcal{A}$ runs simultaneously.*

Proof. When $\mathcal{SSMFP}$ accepts a new valid message m, the processor which sends m executes rule $(\boldsymbol{R_1})$. By construction of the rule, this execution creates a normal caterpillar C_m of type E associated to m.

While m is not delivered to its destination, we know, by Lemmas 4 and 3, that C_m follows infinitely often the above cycle: C_m is of type E and becomes of type F (type A is impossible since m is not delivered), C_m is of type F and becomes of type E.

This implies that there always exists at least one copy of m in $buf_p(1)$ (if p is the sending processor of m). Then, this message is not deleted without being delivered to its destination. □

Lemma 9. *Under a weakly fair daemon, $\mathcal{SSMFP}$ never duplicates a valid message even if $\mathcal{A}$ works simultaneously.*

Proof. It is obvious that the emission of a message m by rule $(\boldsymbol{R_1})$ only creates one caterpillar of type E associated to m.

Then, observe that all rules are designed to obtain the following property: if a caterpillar has one head in a configuration, it also has one head in the following configuration whatever rules have been applied. Indeed, this property is ensured by the fact that the next processor on the path of a message m is computed (and put in the second field on the message) when m is copied into a buffer $buf_p(i)$ (not when it is forwarded to a neighbor). Consequently, if there is a routing table move after the copy of m in $buf_p(i)$, the caterpillar does not fork. The head of the caterpillar remains unique.

We can conclude that, for any valid message m, there always exists a unique caterpillar C_m associated to m. When m is delivered, C_m becomes of type A by construction of rules $(\boldsymbol{R_4})$ and $(\boldsymbol{R_{10}})$. Following Lemma 2, C_m disappears in finite time. m cannot be delivered several times. □

Theorem 1. *$\mathcal{SSMFP}$ is a snap-stabilizing message forwarding protocol for $\mathcal{SP}$ even if routing tables are corrupted in the initial configuration when $\mathcal{A}$ runs simultaneously.*

Proof. Proposition 2 and Lemma 8 allows us to conclude that $\mathcal{SSMFP}$ is a snap-stabilizing message forwarding protocol for specification $\mathcal{SP}'$ even if routing tables are corrupted in the initial configuration, on condition that $\mathcal{A}$ runs simultaneously.

Then, using this remark and Lemma 9, we obtain the result. □

5. Time complexities

Since our algorithm needs a weakly fair daemon, there is no point in doing an analysis in terms of steps. It is why all the following complexity analysis is given in rounds. Let $R_{\mathcal{A}}$ be the stabilization time of $\mathcal{A}$ in terms of rounds.

Proposition 3. *In the worst case, $\Theta(nD)$ invalid messages are delivered to processor d.*

Proof. In the initial configuration, the system has at most $n(D+1)$ distinct invalid messages of destination d. Then, the number of invalid messages delivered to d is in $O(nD)$.

We can obtain the lower bound with a chain of $n = 2q+1$ processors labeled $p_1, p_2, \ldots, p_n$. Assume that all buffers of rank strictly less than $q+2$ initially contain a message of destination p_{q+1} and other buffers are empty. Moreover, assume that routing tables are initially correct. Then, $\mathcal{SSMFP}$ delivers all invalid messages of this initial configuration to p_{q+1}. This initial configuration contains $n(q+1) = n(\frac{D}{2}+1) \in \Theta(nD)$ invalid messages. The result follows. □

Proposition 4. *In the worst case, a message m (of destination d) needs $O(max(R_{\mathcal{A}}, nD\Delta^D))$ rounds to be delivered to d once it has been sent out by its source.*

Proof. Firstly, we prove by induction the following fact: if γ is a configuration in which routing tables are correct and in which a message of destination d exists and C_m is a caterpillar of type E associated to m whose head is a buffer of rank $1 \leq i+t-1 < D+1$ on $p \neq d$, then the head of C_m goes up by one buffer in at most $O(\Delta^{D+1-(i+t-1)})$ rounds if there exists no abnormal caterpillar whose tail is a buffer of rank greater than $i+t$.

Secondly, it is possible to show that $\mathcal{C}$, the set of abnormal caterpillars in γ loses at least one element during the $O(\Delta^D)$ rounds which follow γ. Then, we can say that, when routing tables are correct, an accepted message is forwarded in at most $O(nD\Delta^D)$ rounds.

Finally, we can deduce the result when m is emitted in a configuration in which routing tables are not correct, since the message is delivered in at most $O(nD\Delta^D)$ rounds after routing tables computation (which takes at most $O(R_{\mathcal{A}})$ rounds if m is not delivered during the routing tables computation, since we have assumed the priority of $\mathcal{A}$). □

Proposition 5. *The delay (waiting time before the first emission) and the waiting time (between two consecutive emissions) of $\mathcal{SSMFP}$ is $O(max(R_{\mathcal{A}}, nD\Delta^D))$ rounds in the worst case.*

Proof. Let p be a processor which has a message of destination d to emit. By the fairness of $choice_p(d)$, we can say that m is sent after at most $(\Delta - 1)$ releases of $buf_p(1)$. The result of Proposition 4 allows us to say that $buf_p(1)$ is released in $O(max(R_{\mathcal{A}}, nD\Delta^D))$ rounds at worst. Indeed, we can deduce the result. □

The complexity obtained in Proposition 4 is due to the fact that the system delivers a huge quantity of messages during the forwarding of the considered message. It is why we are interested now in the amortized complexity (in rounds) of our algorithm. For an execution Γ, this measure is equal to the number of rounds of Γ divided by the number of delivered messages during Γ (see [20] for a formal definition).

Proposition 6. *The amortized complexity to forward a message of $\mathcal{SSMFP}$ is in $O(max(R_{\mathcal{A}}, D))$ rounds when there exists no invalid messages.*

Proof. Firstly, we prove the following property: if γ is a configuration in which at least one caterpillar of type E exists, routing tables are correct, and there exists no invalid messages, then $\mathcal{SSMFP}$ delivers at least one message to a processor in the $3D + 1$ rounds following γ.

Assume now an initial configuration in which routing tables are correct and in which there exists no invalid messages. Let Γ be one execution which leads to the worst amortized complexity. Let R_Γ be the number of rounds of Γ. By the last remark, we can say that $\mathcal{SSMFP}$ delivers at least $\frac{R_\Gamma}{3D+1}$ messages during Γ. So, we have an amortized complexity of $\frac{R_\Gamma}{\frac{R_\Gamma}{3D+1}} \in \Theta(D)$.

Then, the announced result is obvious. □

6. Conclusion

In this paper, we provide an algorithm to solve the message forwarding problem in a snap-stabilizing way (when a self-stabilizing algorithm for computing routing tables runs simultaneously) for a specification which forbids message losses and duplication. This property implies the following fact: our protocol can forward any generated message to its destination regardless of the state of the routing tables in the initial configuration. Such an algorithm allows the processors of the network to send messages to each other without waiting for the routing table computation.

As in [1], we show that it is possible to adapt a fault-free protocol into a snap-stabilizing one without memory over cost. This new algorithm improves the one proposed in [1] since it needs $\Theta(D)$ buffers per processor versus $\Theta(n)$ for the former. Although we prove that this new protocol has a worse time complexity in the worst case than the one of [1], we prove that their amortized complexities are equal. That allows us to state that this new protocol has, from a practical point of view, similar time complexity than the previous one. But the following problem is still open: what is the minimal number of buffers to allow snap-stabilization on the message forwarding problem?

References

[1] A. Cournier, S. Dubois, V. Villain, A snap-stabilizing point-to-point communication protocol in message-switched networks, in: IPDPS, 2009, pp. 1–11.
[2] A. Cournier, S. Dubois, V. Villain, How to improve snap-stabilizing point-to-point communication space complexity? in: SSS, 2009, pp. 195–208.
[3] E.W. Dijkstra, Self-stabilizing systems in spite of distributed control, Commun. ACM 17 (11) (1974) 643–644.
[4] A. Bui, A.K. Datta, F. Petit, V. Villain, Snap-stabilization and pif in tree networks, Distrib. Comput. 20 (1) (2007) 3–19.
[5] K.M. Chandy, J. Misra, Distributed computation on graphs: shortest path algorithms, Commun. ACM 25 (11) (1982) 833–837.
[6] J. van Leeuwen, R.B. Tan, Compact routing methods: a survey, in: SIROCCO, 1994, pp. 99–110.
[7] P. Merlin, A. Segall, A failsafe distributed routing protocol, IEEE Trans. Commun. 27 (9) (1979) 1280–1287.
[8] S.-T. Huang, N.-S. Chen, A self-stabilizing algorithm for constructing breadth-first trees, Inform Process. Lett. 41 (2) (1992) 109–117.
[9] A. Kosowski, L. Kuszner, A self-stabilizing algorithm for finding a spanning tree in a polynomial number of moves, in: PPAM, 2005, pp. 75–82.
[10] C. Johnen, S. Tixeuil, Route preserving stabilization, in: Self-Stabilizing Systems, 2003, pp. 184–198.
[11] J. Duato, A necessary and sufficient condition for deadlock-free routing in cut-through and store-and-forward networks, IEEE Trans. Parallel Distrib. Syst. 7 (8) (1996) 841–854.
[12] P.M. Merlin, P.J. Schweitzer, Deadlock avoidance in store-and-forward networks, in: Jerusalem Conference on Information Technology, 1978, pp. 577–581.
[13] L. Schwiebert, D.N. Jayasimha, A universal proof technique for deadlock-free routing in interconnection networks, in: SPAA, 1995, pp. 175–184.
[14] S. Toueg, Deadlock- and livelock-free packet switching networks, in: STOC, 1980, pp. 94–99.
[15] S. Toueg, K. Steiglitz, Some complexity results in the design of deadlock-free packet switching networks, SIAM J. Comput. 10 (4) (1981) 702–712.
[16] S. Toueg, J.D. Ullman, Deadlock-free packet switching networks, SIAM J. Comput. 10 (3) (1981) 594–611.
[17] G. Tel, Introduction to Distributed Algorithms, 2nd edition, Cambridge University Press, Cambridge, UK, 2000.
[18] M. Faloutsos, P. Faloutsos, C. Faloutsos, On power-law relationships of the internet topology, in: SIGCOMM, 1999, pp. 251–262.
[19] S. Dolev, A. Israeli, S. Moran, Uniform dynamic self-stabilizing leader election, IEEE Trans. Parallel Distrib. Syst. 8 (4) (1997) 424–440.
[20] T. Cormen, C. Leierson, R. Rivest, C. Stein, Introduction to Algorithms, 2nd edition, MIT, 2001.