How to Improve Snap-Stabilizing Point-to-Point Communication Space Complexity?*

Alain Cournier[1], Swan Dubois[2], and Vincent Villain[1]

[1] MIS Laboratory, Université de Picardie Jules Verne
[2] LIP6 - UMR 7606 Université Pierre et Marie Curie - Paris 6/INRIA Rocquencourt

Abstract. A *snap-stabilizing* protocol, starting from any configuration, always behaves according to its specification. In this paper, we are interested in message forwarding problem in a message-switched network. In this problem, we must manage resources of the system to deliver messages to any processor of the network. In this purpose, we use information given by a routing algorithm. By the context of stabilization (in particular, the system starts in any configuration), this information can be corrupted. In [1], authors show that there exists snap-stabilizing algorithms for this problem (in the *state model*). That implies that we can ask the system to begin forwarding messages without losses even if routing informations are initially corrupted.

In this paper, we propose another snap-stabilizing algorithm for this problem which improves the space complexity of the one of [1].

1 Introduction

The quality of a distributed system depends on its *fault-tolerance*. Many fault-tolerant schemes have been proposed. For instance, *self-stabilization* [2] allows to design a system tolerating arbitrary transient faults. A self-stabilizing system, regardless of the initial state of the system, is guaranteed to converge into the intended behavior in a finite time. Another paradigm is *snap-stabilization* ([3]). 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 unit.

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 (it is the *routing problem*) and the management of network resources in order to forward messages (it is the *message forwarding problem*). These two problems received a great attention in literature. The routing problem is studied for example in [4,5,6] and self-stabilizing

* This work has been supported in part by the ANR project SPADES (08-ANR-SEGI-025).

R. Guerraoui and F. Petit (Eds.): SSS 2009, LNCS 5873, pp. 195–208, 2009.

approaches can be found in [7,8,9]. The forwarding problem has also been well studied, see [10,11,12,13,14,15]. As far we know, only [1] deals 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 reach their destinations). Problems come of the following fact: messages traveling through a message-switched network ([16]) 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 to the message forwarding. So, it is a problem of bounded resources management which exposes the network to deadlocks and livelocks if no control is performed. In this paper, we focus about a message forwarding protocol which deals withthe problem using a stabilizing approach. The goal is to allow the system to forward messages regardless of the state of the routing tables. Obviously, we need that these routing tables repair themselves within a finite time. So, we assume the existence of a self-stabilizing protocol to compute routing tables.

In the following, a valid message is a message which has been sent out by a processor. As a consequence, an invalid message is present in the initial configuration. We can now specify the problem. We propose a specification of the problem where duplications (*i.e.* the same message reaches many time 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 a finite time.
- Any valid message is delivered to its destination once and only once in a finite time.

In [1], authors show that it is possible to transform a forwarding algorithm of [11] into a snap-stabilizing one without any significant over 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 networks). 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). Even if this improvement can be seen as quite useful from a theoretical point of view (since n and D are close values in the worst case), we believe that it could be very interesting from a practical point of view. Indeed, practical networks have in general a diameter which is very smaller than the number of nodes (for example, [17] shows that the diameter of Internet is near to 6 in 2000 although it had near to 14,000 nodes).

The remaining of this paper is organized as follows: we present first our model (section 2), then we give and prove our solution in the state model (section 3). Finally, we conclude in section 4.

2 Preliminaries

We consider a network as 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. In the following, we assume that the network is identified, *i.e.* 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.

State model. We consider the classical local shared memory model of computation (see [16]) 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 in 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 constituted as follows: $<$ 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 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 note $\mathcal{C}$ the set of all configurations of the system. Let $\gamma \in \mathcal{C}$ and A an action of p ($p \in V$). A is said 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 $\Gamma = (\gamma_0, \gamma_1, ..., \gamma_i, \gamma_{i+1}, ...)$ 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 shared 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 weakly fairness assumption, meaning that we assume that every continuously enabled processor is eventually chosen by the daemon. Concerning the distribution, 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 [18]. 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.

Message-switched network. Today, most of computer networks use a variant of the *message-switching* method (also called *store-and-forward* method). It's why we have chosen to work with this switching model. In this section, we quickly present this method (see [16] 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 moves:

1- ***Generation:***when a processor sends 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 in 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 buffer of the processor 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.

Stabilization. We recall here some formal definitions.

Definition 1 (Self-Stabilization [2]).
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 $\Gamma' = (\gamma_0, \gamma_1, ..., \gamma_l)$ of Γ such that all executions starting from γ_l satisfies $\mathcal{S}_{\mathcal{T}}$.

Definition 2 (Snap-Stabilization [3]).
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 $\mathcal{S}_{\mathcal{T}}$.

This definition 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 unit. 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 property 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 use these two remarks to prove snap-stabilization of our protocol in the following.

3 Proposed Protocol

3.1 Description

To simplify the presentation, we assume that the routing algorithm induces only minimal paths in number of edges. We have seen that, by default, the network always allows message moves between buffers. But, if we do no control on these moves, the network can reach unacceptable situations such as deadlocks, livelocks or message losses. If such situations appear, specifications of message forwarding are not respected. Now, we quickly present solutions brought by the literature in the case where routing tables are correct in the initial configuration. In order to avoid deadlocks, we must define an algorithm which permits or forbids various moves in the network (functions of the current occupation of buffers) in order to prevent the network to reach a deadlock. Such algorithms are called deadlock-free controllers (see [16] for a much detailed description). Livelocks can be avoided by fairness assumptions on the controller for the generation and the forwarding of messages. Message losses are avoided by the using of identifier on messages (for example, the concatenation of the identity of source and a two-value flag). [11] introduced 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. 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 a graph buffer of [11] in order to obtain a snap-stabilizing forwarding protocol.

In this paper, we are interested in another buffer graph introduced in [11]. Each processor have $D + 1$ 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 "come upstairs" the buffer rank (the reader can find an example of such a graph in Figure 1). We need $D+1$ buffers per processor since, in the worst case, there are D forwarding of a message between its generation and its consumption.

Our idea is to adapt this scheme in order to tolerate transient faults. To perform this goal, we assume the existence of a self-stabilizing silent algorithm $\mathcal{A}$ to compute routing tables (see e.g. [7,8,9]) which runs simultaneously to our message forwarding protocol 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 a 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 are ensured 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

have 3 values: E (Emission), A (Acknowledgment) and F (Fail). Forwarding of a valid message m (of 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:
3.1- It is delivered and the copy of m takes type A by $(\boldsymbol{R_4})$ or $(\boldsymbol{R_{10}})$.
3.2- Type A is spread to the sink following the incoming path by $(\boldsymbol{R_7})$.
3.3- 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 $(\boldsymbol{R_{14}})$.
3.4- 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:
4.1- The copy of m takes type F by $(\boldsymbol{R_{13}})$.
4.2- Type F is spread to the sink following the incoming path by $(\boldsymbol{R_7})$.
4.3- 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 $(\boldsymbol{R_{14}})$).
4.4- Then, the sink of m gives the type E to its copy, that begin a new cycle: m is sending once again by $(\boldsymbol{R_2})$ or $(\boldsymbol{R_6})$.

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 4.4 (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 losses. 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 that our algorithm sets the shared variable to $false$ (when a message is sent out).

3.2 Proof of the Snap-Stabilization

In this section, we give ideas[2] to prove that $\mathcal{SSMFP}$ is a snap-stabilizing message forwarding protocol for specification $\mathcal{SP}$. We introduce a second specification $\mathcal{SP}'$ of the problem. This specification is identical to $\mathcal{SP}$ but it allows message duplications. Our proof has the following map. First, we prove that

[1] With copy in buffers of increasing rank.
[2] Due to the lack of place, formal proofs are omitted. A full version of this work is available in [19].

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

Data:

- n, D : natural integers equal resp. to the number of processors and to the diameter of the network.
- $I = \{0, ..., n-1\}$: 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 cross over by the message, $d \in I$ identity of the destination of the message, $c \in \{E, A, F\}$ color of the message.

Variables:

- $\forall i \in \{1, ..., D+1\}$, $buf_p(i)$: 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 $r \in N_p$).
- $\forall i \in \{2, ..., D+1\}, choice_p(i)$: 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 */

(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 */

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

(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 */

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

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

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

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

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

End of Algorithm 1:

/* Rules for buffers of rank 2 to D */
/* Forwarding of messages */
$(\boldsymbol{R_8})$ :: $\exists i \in \{2,...,D\}, ((buf_p(i) = \varepsilon) \wedge (choice_p(i) = s) \wedge (buf_s(i-1) = (m,p,q,d,E)) \wedge (buf_{nextHop_p(d)}(i+1) \neq (m,r,p,d,c))) \longrightarrow buf_p(i) := (m, nextHop_p(d), s, d, E)$ /* Erasing of messages which acknowledgment has been forwarded */
$(\boldsymbol{R_9})$ :: $\exists i \in \{2,...,D\}, ((buf_p(i) = (m,r,q,d,c)) \wedge (c \in \{F,A\}) \wedge (d \neq p) \wedge (buf_q(i-1) = (m,p,q',d,c)) \wedge (buf_r(i+1) \neq (m,r',p,d,c)) \longrightarrow buf_p(i) := \varepsilon$
/* Rules for buffers of rank 2 to $D+1$ */
/* Consumption of a message and generation of the acknowledgment A */
$(\boldsymbol{R_{10}})$ :: $\exists i \in \{2,...,D+1\}, buf_p(i) = (m,r,q,p,E) \longrightarrow deliver_p(m);\ buf_p(i) := (m,r,q,p,A)$
/* Erasing of messages for p which acknowledgment has been forwarded */
$(\boldsymbol{R_{11}})$ :: $\exists i \in \{2,...,D+1\}, ((buf_p(i) = (m,r,q,p,c)) \wedge (c \in \{F,A\}) \wedge (buf_q(i-1) = (m,p,q',p,c))) \longrightarrow buf_p(i) := \varepsilon$
/* 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 of 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 */
$(\boldsymbol{R_{15}})$ :: $\exists i \in \{2,...,D\}, ((buf_p(i) = (m,r,q,d,c)) \wedge (c \in \{F,A\}) \wedge (buf_r(i+1) \neq (m,r',p,d,c)) \wedge (buf_q(i-1) \neq (m,p,q',d,c'))) \longrightarrow buf_p(i) := \varepsilon$
$(\boldsymbol{R_{16}})$ :: $\exists i \in \{2,...,D\}, ((buf_p(i) = (m,r,q,d,c)) \wedge (c \in \{F,A\}) \wedge (buf_r(i+1) \neq (m,r',p,d,c)) \wedge (buf_q(i-1) = (m,p,q',d,c')) \wedge (c' \in \{F,A\} \backslash \{c\} \vee q = d)) \longrightarrow buf_p(i) := \varepsilon$
$(\boldsymbol{R_{17}})$ :: $(buf_p(D+1) = (m,r,q,d,c)) \wedge (c \in \{F,A\}) \wedge (buf_q(D) \neq (m,p,q',d,c')) \longrightarrow buf_p(D+1) := \varepsilon$
$(\boldsymbol{R_{18}})$ :: $(buf_p(D+1) = (m,r,q,d,c)) \wedge (c \in \{F,A\}) \wedge (buf_q(D) = (m,p,q',d,c')) \wedge (c' \in \{F,A\} \backslash \{c\} \vee q = d) \longrightarrow buf_p(D+1) := \varepsilon$

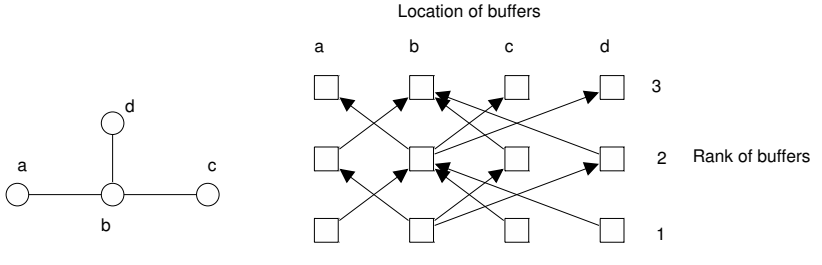

Fig. 1. Example of our buffer graph (on the right) for the network on the left

$\mathcal{SSMFP}$ is a snap-stabilizing message forwarding protocol for specification $\mathcal{SP}'$ if routing tables are correct in the initial configuration (Proposition 1). Then, we can show that $\mathcal{SSMFP}$ is a self-stabilizing message forwarding protocol for specification $\mathcal{SP}'$ even if routing tables are corrupted in the initial configuration (Proposition 2). Finally, we obtain that $\mathcal{SSMFP}$ is a snap-stabilizing message forwarding protocol for specification $\mathcal{SP}$ even if routing tables are corrupted in the initial configuration (Proposition 3). 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 thanks to the use of the flag. Let γ be a configuration of the network. A message m is existing in γ if at least one buffer contains m in γ.

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 $C_m = buf_{p_1}(i)...buf_{p_t}(i+t-1)$ (with $t \geq 1$) which satisfies:*

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

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.

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

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

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

When we study the behavior of these caterpillars from some configurations, we obtain the following properties:

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

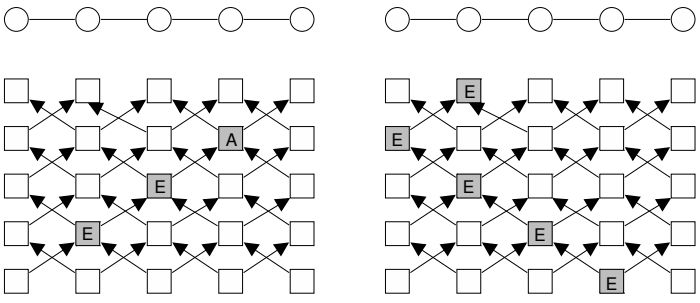

Fig. 2. Examples of caterpillar: abnormal of type A (left) and normal of type E (right)

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

Lemma 3. *Let γ be a configuration and m be a message 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 a finite time.*

Lemma 4. *Let γ be a configuration and m be a message 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 a finite time.*

Assume that there exists a normal caterpillar of type E in a configuration γ in which routing tables are correct, then we can observe that, under a weakly fair daemon:
- by lemma 4, C_m becomes a caterpillar of type A or F in a finite time.
- in the latter case, lemma 3 allows us to say that C_m becomes a caterpillar of type E in a finite time. Then, C_m becomes of type A in a finite time by lemma 4 and the fact that routing tables are correct. So, we have the lemma:

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

Assume that a processor p has a message m to forward in a configuration in which routing tables are correct. Lemmas 3, 5 and 2 allow us to say that the buffer of rank 1 of p is empty in a finite time. Then, it is easy to see that the rule $(\boldsymbol{R_1})$ is enabled in a finite time and remains forever. So, the weakly fair daemon allows us to state:

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

Assume that a processor generate a message in a configuration in which routing tables are correct. This implies the creation of a normal caterpillar of type E.

By the lemma 5, this caterpillar become of type A in a finite time. That means that the message has been delivered to its destination by rule $(\boldsymbol{R_1})$ or $(\boldsymbol{R_4})$. Then, we have:

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 a finite time under a weakly fair daemon.*

Assume that routing tables are correct in the initial configuration. To prove that our algorithm is a snap-stabilizing message forwarding protocol for specification $\mathcal{SP}'$, we must prove that $(\boldsymbol{R_1})$ (the starting action) is executed within a finite time if a computation is requested. Lemma 6 proves this. After a starting action, the protocol is executed in accordance to $\mathcal{SP}'$. If we consider that $(\boldsymbol{R_1})$ have been executed at least one time, we can prove that: the first property of $\mathcal{SP}'$ is always verified (by 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 verified (by Lemma 7). By the remark which follows the definition 2, this implies the following result:

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

We recall that a self-stabilizing silent algorithm $\mathcal{A}$ for computing routing tables is running simultaneously to $\mathcal{SSMFP}$. Moreover, we assume that $\mathcal{A}$ has priority over $\mathcal{SSMFP}$ (*i.e.* a processor which have enabled actions for both algorithms always chooses the action of $\mathcal{A}$). This guarantees us that routing tables are correct and stable within a finite time regardless of their initial states. As we are guaranteed that $\mathcal{SSMFP}$ is a snap-stabilizing message forwarding protocol for specification $\mathcal{SP}'$ from a such configuration by Proposition 1, we can conclude:

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.*

Assume that a processor p generate a message m. This implies the creation of a normal caterpillar of type E. While m is not deliver to its destination, we know by lemma 4 and 3 that C_m is continuously transforming in type F (not A since m is not deliver) then in type E. This implies that there exists a copy of m in the buffer of rank 1 of p until m is deliver to its destination, that proves:

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

It is obvious that the emission of a message by rule $(\boldsymbol{R_1})$ only creates one caterpillar of type E. Then, we can observe that all rules are designed to obtain the following property: if a caterpillar has one head in a configuration, it has also one head in the following configuration whatever rules have been applied. Indeed, it is important to remark that the next processor on the path of a message is computed when the message is copied into a buffer not when it is forwarded to a neighbor (this why routing table moves have no effects on caterpillars). Then, we have:

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

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 $\mathcal{A}$ run simultaneously. Using this remark and Lemma 9, we have:

Proposition 3. *$\mathcal{SSMFP}$ is a snap-stabilizing message forwarding protocol for $\mathcal{SP}$ even if routing tables are corrupted in the initial configuration when $\mathcal{A}$ run simultaneously.*

3.3 Time Complexities

In this section, we give time complexities results[3]. Since $\mathcal{SSMFP}$ needs a weakly fair daemon, there is no points to study complexities in terms of steps. It's why all results of this section are given in terms of rounds. Let $R_{\mathcal{A}}$ be the stabilization time of $\mathcal{A}$ in terms of rounds.

Proposition 4. *In the worst case, $\Theta(nD)$ invalid messages are delivered to a processor.*

Proposition 5. *In the worst case, an accepted message needs $O(max\{R_{\mathcal{A}}, nD\Delta^D\})$ rounds to be delivered to its destination.*

Proposition 6. *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.*

The complexity obtained in Proposition 5 is due to the fact that the system delivers a huge quantity of messages during the forwarding of the considered message. It's why we are now interested 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 7. *The amortized complexity (to forward a message) of $\mathcal{SSMFP}$ is $O(max\{R_{\mathcal{A}}, D\})$ rounds if there is no invalid messages.*

4 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

[3] Due to the lack of space, proofs are omitted but available in [19].

any emitted message to its destination regardless of the state of routing tables in the initial configuration. Such an algorithm allows the processors of the network to send messages to 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 improve the one proposed in [1] since it needs $\Theta(D)$ buffers per processor versus $\Theta(n)$ for the former. 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. Cournier, A., Dubois, S., Villain, V.: A snap-stabilizing point-to-point communication protocol in message-switched networks. In: IPDPS (to appear, technical report available [19]) (2009)
2. Dijkstra, E.W.: Self-stabilizing systems in spite of distributed control. Commun. ACM 17(11), 643–644 (1974)
3. Bui, A., Datta, A.K., Petit, F., Villain, V.: Snap-stabilization and pif in tree networks. Distributed Computing 20(1), 3–19 (2007)
4. Chandy, K.M., Misra, J.: Distributed computation on graphs: Shortest path algorithms. Commun. ACM 25(11), 833–837 (1982)
5. van Leeuwen, J., Tan, R.B.: Compact routing methods: A survey. In: SIROCCO, pp. 99–110 (1994)
6. Merlin, P., Segall, A.: A failsafe distributed routing protocol. IEEE Trans. Communications 27(9), 1280–1287 (1979)
7. Huang, S.T., Chen, N.S.: A self-stabilizing algorithm for constructing breadth-first trees. Inf. Process. Lett. 41(2), 109–117 (1992)
8. Kosowski, A., Kuszner, L.: A self-stabilizing algorithm for finding a spanning tree in a polynomial number of moves. In: Wyrzykowski, R., Dongarra, J., Meyer, N., Waśniewski, J. (eds.) PPAM 2005. LNCS, vol. 3911, pp. 75–82. Springer, Heidelberg (2006)
9. Johnen, C., Tixeuil, S.: Route preserving stabilization. In: Huang, S.-T., Herman, T. (eds.) SSS 2003. LNCS, vol. 2704, pp. 184–198. Springer, Heidelberg (2003)
10. Duato, J.: A necessary and sufficient condition for deadlock-free routing in cut-through and store-and-forward networks. IEEE Trans. Parallel Distrib. Syst. 7(8), 841–854 (1996)
11. Merlin, P.M., Schweitzer, P.J.: Deadlock avoidance in store-and-forward networks. In: Jerusalem Conference on Information Technology, pp. 577–581 (1978)
12. Schwiebert, L., Jayasimha, D.N.: A universal proof technique for deadlock-free routing in interconnection networks. In: SPAA, pp. 175–184 (1995)
13. Toueg, S.: Deadlock- and livelock-free packet switching networks. In: STOC, pp. 94–99 (1980)
14. Toueg, S., Steiglitz, K.: Some complexity results in the design of deadlock-free packet switching networks. SIAM J. Comput. 10(4), 702–712 (1981)
15. Toueg, S., Ullman, J.D.: Deadlock-free packet switching networks. SIAM J. Comput. 10(3), 594–611 (1981)
16. Tel, G.: Introduction to Distributed Algorithms, 2nd edn. Cambridge University Press, Cambridge (2000)
17. Faloutsos, M., Faloutsos, P., Faloutsos, C.: On power-law relationships of the internet topology. In: SIGCOMM, pp. 251–262 (1999)

18. Dolev, S., Israeli, A., Moran, S.: Uniform dynamic self-stabilizing leader election. IEEE Trans. Parallel Distrib. Syst. 8(4), 424–440 (1997)
19. Cournier, A., Dubois, S., Villain, V.: Two snap-stabilizing point-to-point communication protocols in message-switched networks. Technical Report arXiv:0905.2540, INRIA (2009)
20. Cormen, T., Leierson, C., Rivest, R., Stein, C.: Introduction à l'algorithmique, 2nd edn. Eyrolles (2002)