A snap-stabilizing point-to-point communication protocol in message-switched networks

Alain Cournier
MIS Laboratory,
Université de Picardie Jules Verne
33 rue Saint Leu,
80039 Amiens Cedex 1 (France)
alain.cournier@u-picardie.fr

Swan Dubois
LIP6 - UMR 7606/INRIA Rocquencourt,
Project-team REGAL
Université Pierre et Marie Curie - Paris 6
104 Avenue du Président Kennedy,
75016 Paris (France)
swan.dubois@lip6.fr

Vincent Villain
MIS Laboratory,
Université de Picardie Jules Verne
33 rue Saint Leu,
80039 Amiens Cedex 1 (France)
vincent.villain@u-picardie.fr

Abstract

A snap-stabilizing *protocol, starting from any configuration, always behaves according to its specification. In this paper, we present a snap-stabilizing protocol to solve the 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 informations given by a routing algorithm. By the context of stabilization (in particular, the system starts in any configuration), these informations can be corrupted. So, the existence of a snap-stabilizing protocol for the message forwarding problem implies that we can ask the system to begin forwarding messages even if routing informations are initially corrupted.*

In this paper, we propose a snap-stabilizing algorithm (in the state model*) for the following specification of the problem:*

- *Any message can be generated in a finite time.*
- *Any emitted message will be delivered to its destination once and only once in a finite time.*

This implies that our protocol can deliver any emitted message regardless of the state of routing tables in the initial configuration.

Keywords: Distributed protocols, snap-stabilization, fault-tolerance, message forwarding, point-to-point communication, deadlock-free routing, message-switched networks.

1 Introduction

The quality of a distributed system depends on its fault-tolerance. Many fault-tolerant schemes have been proposed. For instance, self-stabilization [8] 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. An other paradigm called snap-stabilization has been introduced in [3, 2]. A snap-stabilizing protocol guarantees that, starting from any configuration, it always behaves according to its specification. In other words, 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 [1, 4, 13, 14, 15, 29, 30, 20, 23, 25] and self-stabilizing approach can be found (directly or not) in [16, 18, 9, 17]. The forwarding problem has also been well studied, see [12, 21, 22, 26, 27, 28] for example. As far we know, the message forwarding problem was never directly studied with a snap-stabilizing approach (note that the protocol proposed by [17] can be used to perform a self-stabilizing forwarding protocol for dynamic networks since it is guaranteed that the routing tables remain loop-free even if topological changes are allowed). This is the scope of this paper.

Informally, the goal is to give a protocol which allows all processors of the network to send messages to any destination of the network knowing that a routing algorithm calculates the path that messages have to follow to reach their destinations. Problems come of the following fact: messages traveling through a message-switched network ([24]) 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, 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 on a message forwarding protocol which deals the problem with 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 that theses routing tables repair themselves within a finite time. So, we assume the existence of a self-stabilizing protocol to compute routing tables (see [16, 18, 9]).

In the following, we say that a valid message is a message which has been generated by a processor. As a consequence, an invalid message is a message which is present in the initial configuration. We can now specify the problem. We propose a specification of the problem where message duplications (*i.e.* the same message reaches its destination many time while it has been generated only once) are forbidden:

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

- *Any message can be generated in a finite time.*
- *Any valid message will be deliver to its destination once and only once in a finite time.*

The remainder of this paper is organized as follows: we present first our model (section 2), then we give, prove, and analyze our solution in the state model (section 3). Finally, we conclude by some remarks and open problems (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. Every processor p can distinguish all its links. To simplify the presentation, we refer to a link (p, q) of a processor p by the label q. We assume that the labels 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 have 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 a local shared memory model of computation (see [24]) 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 will 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 called 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 actions 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 atomic step is composed of three sequential phases: (i) every processor evaluates its guards, (ii) a daemon chooses some enabled processors, (iii) each chosen processor executes one of its enabled actions. 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 present the strong fairness, weak fairness, and unfairness assumptions. Under a strongly fair daemon, every processor that is enabled infinitively often is chosen by the daemon infinitively often to execute an action. When a daemon is weakly fair, every continuously enabled processor is eventually chosen by the daemon. Finally, the unfair daemon is the weakest scheduling assumption: it can forever prevent a processor to execute an action except if it is the only enabled processor. Concerning the distribution, we assume that the daemon is distributed meaning that, at each step, if one or several 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 (introduced in [10] and modified by [3]). 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 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 are going to present this method (see [24] for a detailed presentation).

Each processor has $\mathcal{B}$ buffers for temporarily storing messages. 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 three 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). As a result of the move the buffer previously occupied by m becomes empty. 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 processor is removed from this buffer (and delivered to the processor). We assume that the network may always allow this move.

2.3 Stabilization

In this section, we give formal definitions of self- and snap-stabilization using notations introduced in 2.1.

Definition 1 (Self-Stabilization [8]) *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 [2, 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 will use these two remarks to prove snap-stabilization of our protocol in the following of this paper.

3 Our protocol

3.1 Informal description

We have seen in section 2.2 that, by default, the network always allows message moves between buffers. But, if we do no control on these moves, the network can

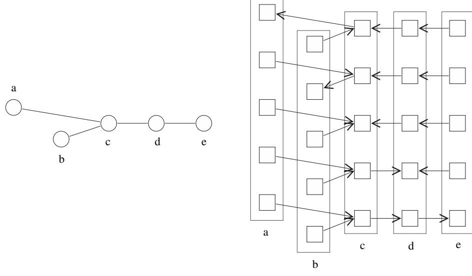

Figure 1. Example of a "destination-based" buffer graph (on the right) on the network on the left.

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 [24] for a much detailed description). [21] introduced a generic method to design deadlock-free controllers. It consists to restrict moves of messages along edges of an oriented graph BG (called buffer graph) defined on the network buffers. Then, it is easy to see that cycles on BG can lead to deadlocks. So, authors show that, if BG is acyclic, they can define a deadlock-free controller on this buffer graph. For example, we can present a "destination-based" buffer graph. In this scheme, we assume that the routing algorithm forwards all packets of Destination d via a directed tree T_d rooted in d. Each processor p of the network has a buffer $b_p(d)$ for each possible Destination d (called the target of $b_p(d)$). The buffer graph has n connected components, each of them containing all the buffers which shared their target. The connected component associated to the target d is isomorphic to T_d (the reader can find an example of such a graph in Figure 1). It is easy to see that this oriented graph is acyclic.

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, one can use the concatenation of the identity of the source and a two-value flag in order to distinguish two consecutive identical messages generated by the same processor for Destination d (since all messages follow the same path in T_d).

The main idea that leads our research is to adapt this solution in order to tolerate the corruption of routing tables in the initial configuration. To perform this goal, we assume the existence of a self-stabilizing silent (*i.e.* no actions are enabled after convergence) algorithm $\mathcal{A}$ to compute routing tables which runs simultaneously to our message forwarding protocol. Moreover, we assume that $\mathcal{A}$ has priority over our protocol (*i.e.* a processor which has enabled actions for both algorithms always chooses the action of $\mathcal{A}$). This guarantees us that routing tables will be correct and constant within a finite time. To simplify the presentation, we assume that $\mathcal{A}$ induces only minimal paths in number of edges. We assume that our protocol 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. We use a controller based on a buffer graph similar to that we presented before (once routing tables are computed). The buffer graph is composed of n connected components, each associated to a destination d and based on the oriented tree T_d. So, we are going to present only one connected component, associated to a destination noted d (others are similar). We use two buffers per processor for Destination d. The first one, noted $bufR_p(d)$ (for processor p), is reserved to the reception of messages whereas the second one, noted $bufE_p(d)$, is used to emit messages (see Figure 2). This scheme allows us to control the advance of messages. Indeed, we allow a message to be forwarded from $bufR_p(d)$ to $bufE_p(d)$ if and only if the message is only present in $bufR_p(d)$ and we erase it simultaneously. In this way, we can control the effect of routing tables moves on messages (duplication or merge which can involve message losses).

To avoid livelocks, we use a fair scheme of selection of processors allowed to forward or to emit a message for each reception 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 last processor cross over by the message and a *color* which is dynamically given to the message when it reaches an emission buffer. In order to distinguish a such incoming message of these contained in reception buffers of neighbors of the considered processor, we give to this incoming message a *color* which is not carry by a such message. It is why a message is considered as a triplet (m, p, c) in our algorithm where m is the useful information of the message, p is the identity of the last processor crossed over by the message, and c is a color (a natural integer between 0 and Δ). We must manage a communication between our algorithm and processors in order to know when a processor have 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

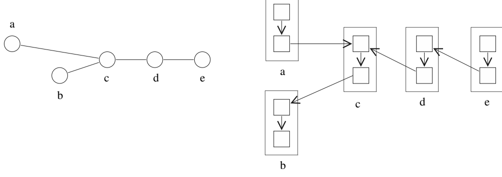

Figure 2. Example of our buffer graph (on the right) for Destination b on the network (on the left).

has a message to send. Otherwise, p must wait that our algorithm sets the shared variable to $false$ (that is done when a message is generated).

The reader can find a complete example of the execution of our algorithm in Figure 3. Diagram (N) shows the network and diagram (0) shows the initial configuration for the connected component associated to b of the buffer graph. We observe that $\Delta = 3$, so we need 4 different values for the variable $color$, we have chosen to represent them by a natural integer in $\{0, 1, 2, 3\}$. Remark that routing tables are incorrect (in particular there exists a cycle involving buffers of a and c) and that there exists an invalid message m' in the reception buffer of b (its $color$ is 0). Then, Processor c emits a message m (its $color$ is 0) in the reception buffer of c to obtain configuration (1). When the message m is forwarded to the emission buffer of c, we associate it the $color$ 1 (since 0 is forbidden, see configuration (2)). During the next step, message m is forwarded to the reception buffer of a (remark that it keeps its $color$) and c emits (in its reception buffer) a new message m' which has the same useful information as the invalid message present on b. So, we obtain configuration (3). Message m can now be erased from the emission buffer of c and m' can be forwarded into this buffer (we associate it the $color$ 2). These two steps lead to configuration (4). Assume that routing tables are repaired during the next step. Simultaneously, processor a is allowed to forward m into its emission buffer. We obtain configuration (5). Remark that the use of $color$ forbids the merge between the two messages which have m' for useful information. Then, the system is able to deliver these three messages by the repetition of moves that we have described:

- forwarding from reception buffer to emission buffer of the same processor.
- forwarding from emission buffer to reception buffer of two processors.
- erasing from emission buffer or delivering.

The sequence of configuration (6) to (12) shows an example of the end of our execution.

3.2 Algorithm

We now present formally our protocol in Algorithm 1. We call it $\mathcal{SSMFP}$ for $\mathcal{S}$nap-$\mathcal{S}$tabilizing $\mathcal{M}$essage $\mathcal{F}$orwarding $\mathcal{P}$rotocol. In order to simplify the presentation, we write the algorithm for Destination d only. Obviously, each destination of the network needs a similar algorithm. Moreover, we assume that all these algorithms run simultaneously (as they are mutually independent, this assumption has no effect on the provided proof).

3.3 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 generated in a finite time.*
- *Any valid message will be deliver to its destination in a finite time.*

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

1. $\mathcal{SSMFP}$ is a snap-stabilizing message forwarding protocol for specification $\mathcal{SP}'$ if routing tables are correct in the initial configuration (Proposition 1).
2. $\mathcal{SSMFP}$ is a self-stabilizing message forwarding protocol for specification $\mathcal{SP}'$ even if routing tables are corrupted in the initial configuration (Proposition 2).
3. $\mathcal{SSMFP}$ 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 generated 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. We say that a message m is existing in γ if at least one buffer contains m in γ. We say that m is existing on p in γ if at least one buffer of p contains m in γ.

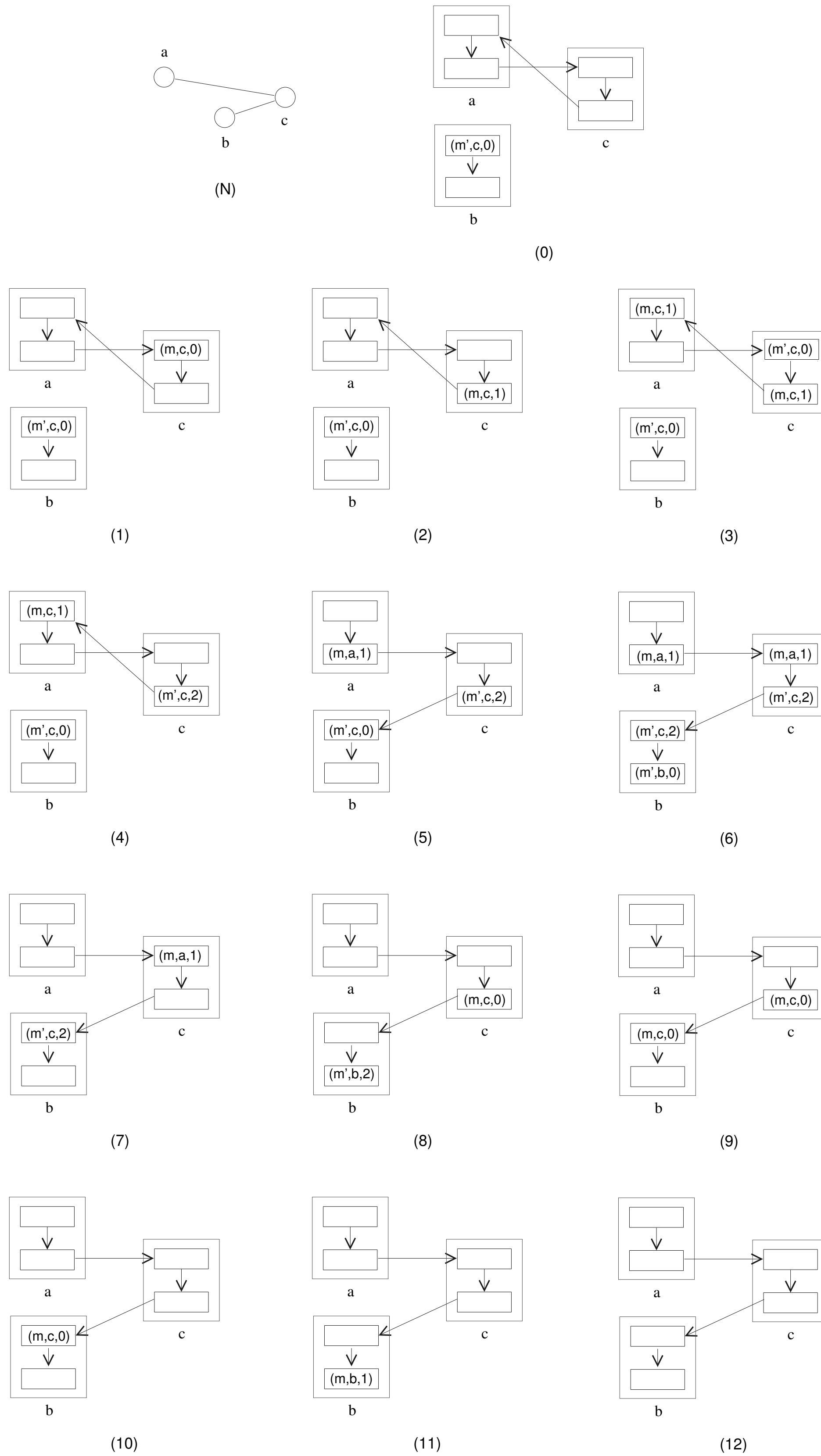

Figure 3. An example of execution of our algorithm.

Algorithm 1 ($\mathcal{SSMFP}$): Message forwarding protocol for Processor p with Destination d.

Data:

- n: natural integer equals to the number of processors of the network.
- $I = \{0, ..., n-1\}$: set of processor identities of the network.
- N_p: set of neighbors of p.
- Δ: natural integer equals to the maximal degree of the network.

Message:

- (m, q, c) with m useful information of the message, $q \in N_p \cup \{p\}$ identity of the last processor crossed over by the message, and $c \in \{0, ..., \Delta\}$ a color. The message destination is the buffer index.

Variables:

- $bufR_p(d)$, $bufE_p(d)$: buffers which can contain a message.

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.

Macros:

- $nextMessage_p$: gives the message waiting in the higher layer.
- $nextDestination_p$: gives the destination of $nextMessage_p$ if it exists, $null$ otherwise.

Procedures:

- $nextHop_p(d)$: neighbor of p given by the routing algorithm for Destination d.
- $choice_p(d)$: fairly chooses one of the processors which can forward or generate a message in $bufR_p(d)$, *i.e.* $choice_p(d)$ satisfies predicate $(choice_p(d) \in N_p \wedge bufE_{choice_p(d)}(d) = (m, q, c) \wedge nextHop_{choice_p(d)}(d) = p) \vee (choice_p(d) = p \wedge request_p)$. 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.
- $color_p(d)$: gives a natural integer c between 0 and Δ such as $\forall q \in N_p$, $bufR_q(d)$ does not contain a message with c as color.

Rules:

/* Rule for the generation of a message */
$(\boldsymbol{R_1})$ :: $request_p \wedge (nextDestination_p = d) \wedge (bufR_p(d) = empty) \wedge (choice_p(d) = p) \longrightarrow bufR_p(d) := (nextMessage_p, p, 0); request_p := false$
/* Rule for the internal forwarding of a message */
$(\boldsymbol{R_2})$ :: $(bufE_p(d) = empty) \wedge (bufR_p(d) = (m, q, c)) \wedge ((q = p) \vee (bufE_q(d) \neq (m, q', c))) \longrightarrow bufE_p(d) := (m, p, color_p(d)); bufR_p(d) := empty$
/* Rule for the forwarding of a message */
$(\boldsymbol{R_3})$ :: $(bufR_p(d) = empty) \wedge (choice_p(d) = s) \wedge (s \neq p) \wedge (bufE_s(d) = (m, q, c)) \longrightarrow bufR_p(d) := (m, s, c)$[1]
/* Rule for the erasing of a message after its forwarding */
$(\boldsymbol{R_4})$ :: $(bufE_p(d) = (m, q, c)) \wedge (p \neq d) \wedge (bufR_{nextHop_p(d)}(d) = (m, p, c)) \wedge (\forall r \in N_p \backslash \{nextHop_p(d)\}, bufR_r(d) \neq (m, p, c)) \longrightarrow bufE_p(d) := empty$
/* Rule for the erasing of a message after its duplication */
$(\boldsymbol{R_5})$ :: $(bufR_p(d) = (m, q, c)) \wedge (bufE_q(d) = (m, q', c)) \wedge (nextHop_q(d) \neq p) \longrightarrow bufR_p(d) := empty$
/* Rule for the consumption of a message */
$(\boldsymbol{R_6})$ :: $(bufE_p(p) = (m, q, c)) \longrightarrow deliver_p(m); bufE_p(p) := empty$

[1] The fact that q may be different of s implies that the message was in the system at the initial configuration. We could locally delete this message but that will not improve the performance of $\mathcal{SSMFP}$.

Definition 3 (Caterpillar of a message m) *Let m be a message of Destination d existing on a processor p in a configuration γ. We define a caterpillar associated to m as the longest sequence of buffers that satisfies one of the three definitions below:*

1. *Caterpillar of type 1:* $(bufR_p(d) = (m, q, c)) \wedge ((bufE_q(d) \neq (m, q', c)) \vee (q = p))$.
2. *Caterpillar of type 2:* $(bufE_p(d) = (m, q, c)) \wedge (bufR_{nextHop_p(d)}(d) \neq (m, p, c))$.
3. *Caterpillar of type 3:* $(bufE_p(d) = (m, q', c)) \wedge \exists q \in N_p,\ (bufR_q(d) = (m, p, c))$.

The reader can find in Figure 4 an example for each type of caterpillar. Remark that an emission buffer can belong to several caterpillars of type 3.

Assume that routing tables are correct in the initial configuration. When we observe the execution of Protocol $\mathcal{SSMFP}$ under a weakly fear daemon, we can see that a caterpillar of type 1 associated to a message m (of Destination d) on a processor p becomes a caterpillar C of type 2 associated to m on p within a finite time (by rule $(\boldsymbol{R_2})$). If p is the destination of m, the message is delivered (by rule $(\boldsymbol{R_6})$) else, C becomes a caterpillar of type 3 on p (by rule $(\boldsymbol{R_3})$) thanks to the fairness of $choice_{nextHop_p(d)}(d)$. If several neighbors are implied in caterpillars of type 3 which share $bufR_p(d)$ (due to invalid messages in the initial configuration), $(\boldsymbol{R_5})$ is enabled for each neighbor q of p such that $nextHop_p(d) \neq q$. It's why rule $(\boldsymbol{R_4})$ is enabled in a finite time for $nextHop_p(d)$, its activation transform C into a caterpillar of type 1 associated to m on $nextHop_p(d)$. Then, we can give the following lemma:

Lemma 1 *Let γ be a configuration in which routing tables are correct. Let m be a message existing on p in γ. Under a weakly fair daemon, the execution of $\mathcal{SSMFP}$ will product within a finite time one of the following effects for any caterpillar of type 1 associated to m:*

- *m is delivered to its destination.*
- *the caterpillar disappeared on p and there exists a caterpillar of type 1 associated to the same message on $nextHop_p(d)$.*

Assume that routing tables are correct in the initial configuration. If a buffer $bufR_p(d)$ contains a caterpillar of type 3, we have seen (in the proof of lemma 1) that this caterpillar become a caterpillar of type 1 on p or disappear. Moreover, Lemma 1 implies that a buffer $bufR_p(d)$ which contains a caterpillar of type 1 is erased within a finite time. Then, the fairness of $choice_p(d)$ allows us to give the following result:

Lemma 2 *Under a weakly fair daemon when routing tables are correct, every processor can generate a first message (*i.e. *it can execute $(\boldsymbol{R_1})$).*

Assume that routing tables are correct in the initial configuration and that a processor p has generated a message m of Destination d (with rule $(\boldsymbol{R_1})$). This implies the creation of a caterpillar of type 1 associated to m in $bufR_p(d)$ when m has been generated. The following result is obtained by $dist(p, d) + 1$ applications of Lemma 1:

Lemma 3 *Once a message is accepted by $\mathcal{SSMFP}$, it will be correctly forwarded to its destination under a weakly fair daemon if routing tables are correct (when the message was accepted).*

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 2 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 2 and the fact that the waiting for the generation of new messages is blocking) and the second property of $\mathcal{SP}'$ is always verified (by Lemma 3). 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 will be correct and constant 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 such a configuration by Proposition 1, we can conclude on the following property:

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.*

By the construction of the algorithm, it is obvious that a message cannot be erased from two distinct buffers simultaneously. Then, the construction of $color_p(d)$ and

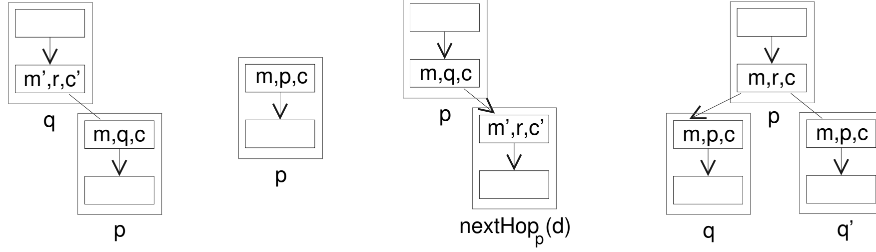

Figure 4. Examples of caterpillar associated to m on p (from left to right: two of type 1, one of type 2 and one of type 3).

of rule $(\boldsymbol{R_4})$ guarantees us that, when $(\boldsymbol{R_4})$ is applied by p, the message in $bufR_{nextHop_p(d)}(d)$ is a copy of that in $bufE_p(d)$. So, the message have been copied at least once before it is erased, that allows us to give the following lemma:

Lemma 4 *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 $(\boldsymbol{R_1})$ creates only one caterpillar of type 1. Then, the construction of rules $(\boldsymbol{R_6})$ and $(\boldsymbol{R_4})$ implies the following property: if a caterpillar of type 1 associated to a message m is present on a processor p and this message is erased from all buffers of p, then only one neighbor of p contains a caterpillar of type 1 associated to m or m have been delivered to its destination (perhaps m have been copied several times but $(\boldsymbol{R_5})$ ensure us that there exists a unique copy of m when $(\boldsymbol{R_4})$ is enabled). This allows us to give the following lemma:

Lemma 5 *Under a weakly fair daemon, $\mathcal{SSMFP}$ never duplicates a valid message even if $\mathcal{A}$ runs simultaneously.*

Proposition 2 and Lemma 4 allows us to conclude that $\mathcal{SSMFP}$ is a snap-stabilizing message forwarding protocol with specification $\mathcal{SP}'$ even if routing tables are corrupted in the initial configuration on condition that $\mathcal{A}$ run simultaneously. Then, using this remark and Lemma 5, we can claim:

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.*

3.4 Time complexities

Let $R_{\mathcal{A}}$ be the stabilization time of $\mathcal{A}$ in terms of rounds.

Proposition 4 *In the worst case, $2n$ invalid messages will be delivered to Processor d.*

Sketch of proof. In the initial configuration, the system has at most $2n$ distinct invalid messages of Destination d (since the connected component of the buffer graph associated to d has $2n$ buffers). In the worst case, all these messages will be delivered to their destination, that allows us to reach the announced bound. $\Box$

Proposition 5 *In the worst case, a message m (of Destination d) needs $O(max(R_{\mathcal{A}}, \Delta^D))$ rounds to be delivered to d once it has been generated by its source.*

Sketch of proof. In a first time, we can show by induction the following result: if γ is a configuration in which routing tables are correct and C is a caterpillar of type 1 associated to a message m (of Destination d) on a processor p such as $dist(p, d) = \delta$, then m is delivered to d or there exists a caterpillar of type 1 associated to m on $nextHop_p(d)$ in at most $O(\Delta^\delta)$ rounds. This result is due to the fairness of $choice_p(d)$ which can allow at most Δ messages to "pass" m. Then, consider that s is the source of a message m of Destination d. We have $dist(s, d) \leq D$ by definition. We can conclude that m is delivered in at most $\sum_{\delta=D}^{0} O(\Delta^\delta) \in O(\Delta^D)$ rounds if routing tables are correct when m is emitted. 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 $O(\Delta^D)$ 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 $\mathcal{A}$ over $\mathcal{SSMFP}$). $\Box$

Proposition 6 *The delay (waiting time before the first emission) and the waiting time (between two consecutive emissions) of $\mathcal{SSMFP}$ is $O(max(R_{\mathcal{A}}, \Delta^D))$ rounds in the worst case.*

Sketch of 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 will be generated after at most $(\Delta - 1)$ releases of $bufR_p(d)$. The result of Proposition 5 allows us to say that $bufR_p(d)$ is released in $O(max(R_{\mathcal{A}}, \Delta^D))$ rounds at worst. Indeed, we can deduce the result. □

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 interest 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 [5] for a formal definition).

Proposition 7 *The amortized complexity (to forward a message) of $\mathcal{SSMFP}$ is $O(max\ (R_{\mathcal{A}}, D))$ rounds.*

Sketch of proof. In a first time, we must prove the following property: if γ is a configuration in which at least one message of Destination d is present and in which routing tables are correct, then $\mathcal{SSMFP}$ delivers at least one message to d in the $3D$ rounds following γ. Assume now an initial configuration in which routing tables are correct. Let Γ be one execution 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 $\frac{R_\Gamma}{3D}$ messages during Γ. So, we have an amortized complexity of $\frac{R_\Gamma}{\frac{R_\Gamma}{3D}} \in \Theta(D)$. Then, the announced result is obvious. □

4 Conclusion

In this paper, we provide the first algorithm (at our knowledge) 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 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. We use a tool called "buffer graph" which has been introduced in [21]. This paper proposed a "destination-based" buffer graph that we have adapted in order to control the effect of routing table moves on messages. Our analysis shows that we ensure snap-stabilization without significant over cost in space or in time with respect to the fault-free algorithm.

[21] also proposed other buffer graphs. So, it is natural to wonder if they could be adapted to tolerate transient faults. In particular, one of them (based on the acyclic covering of the network, see also [24]) is very interesting since it needs less buffers per processor in general (3 for a ring, 2 for a tree...). But, authors of [19] show that it is NP-hard to compute the size of the acyclic covering of any graph. So, this buffer graph cannot be easily applied to any network. An open problem is the following: what is the minimal number of buffers per processor to allow snap-stabilization on the message forwarding problem ?

Another way to improve our protocol is to speed up the message forwarding in the worst case (without increasing amortized complexity). In this goal, we believe that we can keep our protocol and modify the fair scheme of selection of messages $choice_p(d)$. In fact, the complexity of our algorithm depends on the number of messages which can "pass" a specific message at each hop.

Our protocol has the following drawback: when a message m is delivered to a processor p, p cannot determine if m is valid or not. This can bring some problems for applications which use these messages. So, an interesting way of future researches could be to design a protocol which solves this problem. In [6] the authors propose an efficient solution for the PIF problem that deals with a similar problem, unfortunately their approach does not seem suitable for our problem.

Finally, it will be interesting to carry our protocol in the message passing model (a more realistic model of distributed system) in order to enable snap-stabilizing message forwarding in a real network. To our knowledge, in this model, only two snap-stabilizing protocols exist in the literature ([7, 11]). The problem to carry automatically a protocol from the state model to the message passing model is still open.

References

[1] E. M. Bakker, J. van Leeuwen, and R. B. Tan. Prefix routing schemes in dynamic networks. *Computer Networks and ISDN Systems*, 26(4):403–421, 1993.

[2] A. Bui, A. K. Datta, F. Petit, and V. Villain. State-optimal snap-stabilizing pif in tree networks. In *WSS*, pages 78–85, 1999.

[3] A. Bui, A. K. Datta, F. Petit, and V. Villain. Snap-stabilization and pif in tree networks. *Distributed Computing*, 20(1):3–19, 2007.

[4] K. M. Chandy and J. Misra. Distributed computation on graphs: Shortest path algorithms. *Commun. ACM*, 25(11):833–837, 1982.

[5] T. Cormen, C. Leierson, R. Rivest, and C. Stein. *Introduction à l'algorithmique*. Eyrolles, seconde edition, 2002.

[6] A. Cournier, S. Devismes, and V. Villain. Snap-stabilizing pif and useless computations. In *ICPADS (1)*, pages 39–48, 2006.

[7] S. Delaët, S. Devismes, M. Nesterenko, and S. Tixeuil. Snap-stabilization in message-passing systems. *CoRR*, abs/0802.1123, 2008.

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

[9] S. Dolev. Self-stabilizing routing and related protocol. *J. Parallel Distrib. Comput.*, 42(2):122–127, 1997.

[10] S. Dolev, A. Israeli, and S. Moran. Uniform dynamic self-stabilizing leader election. *IEEE Trans. Parallel Distrib. Syst.*, 8(4):424–440, 1997.

[11] S. Dolev and N. Tzachar. Empire of colonies: Self-stabilizing and self-organizing distributed algorithms. In *OPODIS*, pages 230–243, 2006.

[12] 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):841–854, 1996.

[13] M. Flammini and G. Gambosi. On devising boolean routing schemes. *Theor. Comput. Sci.*, 186(1-2):171–198, 1997.

[14] P. Fraigniaud and C. Gavoille. Interval routing schemes. *Algorithmica*, 21(2):155–182, 1998.

[15] C. Gavoille. A survey on interval routing. *Theor. Comput. Sci.*, 245(2):217–253, 2000.

[16] S.-T. Huang and N.-S. Chen. A self-stabilizing algorithm for constructing breadth-first trees. *Inf. Process. Lett.*, 41(2):109–117, 1992.

[17] C. Johnen and S. Tixeuil. Route preserving stabilization. In *Self-Stabilizing Systems*, pages 184–198, 2003.

[18] A. Kosowski and L. Kuszner. A self-stabilizing algorithm for finding a spanning tree in a polynomial number of moves. In *PPAM*, pages 75–82, 2005.

[19] R. Kralovic and P. Ruzicka. Ranks of graphs: The size of acyclic orientation cover for deadlock-free packet routing. *Theor. Comput. Sci.*, 374(1-3):203–213, 2007.

[20] P. Merlin and A. Segall. A failsafe distributed routing protocol. *IEEE Trans. Communications*, 27(9):1280–1287, 1979.

[21] P. M. Merlin and P. J. Schweitzer. Deadlock avoidance in store-and-forward networks. In *Jerusalem Conference on Information Technology*, pages 577–581, 1978.

[22] L. Schwiebert and D. N. Jayasimha. A universal proof technique for deadlock-free routing in interconnection networks. In *SPAA*, pages 175–184, 1995.

[23] W. D. Tajibnapis. A correctness proof of a topology information maintenance protocol for a distributed computer network. *Commun. ACM*, 20(7):477–485, 1977.

[24] G. Tel. *Introduction to Distributed Algorithms*. Cambridge University Press, Cambridge, UK, 2nd edition, 2001.

[25] S. Toueg. An all-pairs shortest-path distributed algorithm. RC 8327 10598, IBM T. J. Watson Research Center, Yorktown Heights, NY, 1980.

[26] S. Toueg. Deadlock- and livelock-free packet switching networks. In *STOC*, pages 94–99, 1980.

[27] S. Toueg and K. Steiglitz. Some complexity results in the design of deadlock-free packet switching networks. *SIAM J. Comput.*, 10(4):702–712, 1981.

[28] S. Toueg and J. D. Ullman. Deadlock-free packet switching networks. *SIAM J. Comput.*, 10(3):594–611, 1981.

[29] J. van Leeuwen and R. B. Tan. Interval routing. *Comput. J.*, 30(4):298–307, 1987.

[30] J. van Leeuwen and R. B. Tan. Compact routing methods: A survey. In *SIROCCO*, pages 99–110, 1994.