Two snap-stabilizing point-to-point communication protocols in message-switched networks

Alain Cournier, Swan Dubois, Vincent Villain

Two snap-stabilizing point-to-point communication protocols in message-switched networks

Alain Cournier[*] Swan Dubois[†] Vincent Villain[‡]

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 information given by a routing algorithm. By the context of stabilization (in particular, the system starts in an arbitrary configuration), this information 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 information are initially corrupted.

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

- Any message can be generated in a finite time.
- Any emitted message is 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.

These two algorithms are based on the previous work of [21]. Each algorithm needs a particular method to be transform into a snap-stabilizing one but both of them do not introduce a significant overcost in memory or in time with respect to algorithms of [21].

[*] MIS Laboratory, Université de Picardie Jules Verne, alain.cournier@u-picardie.fr

[†] LIP6 - UMR 7606 Université Pierre et Marie Curie - Paris 6 & INRIA Rocquencourt, swan.dubois@lip6.fr

[‡] MIS Laboratory, Université de Picardie Jules Verne, vincent.villain@u-picardie.fr

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 routing tables remain loop-free even if topological changes are allowed).

Informally, a message forwarding protocol allows any processor 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* ([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 message forwarding protocols which deal the problem with a snap-stabilizing approach. The goal is to allow the system to forward messages (without looses) regardless of the state of the routing tables. Obviously, we need that theses routing tables repair themselves in a finite time. So, we assume the existence of a self-stabilizing protocol to compute routing tables (see [16, 18, 9]).

In the following, 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 message forwarding problem forbidding duplication.*

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

In this paper, we investigate the possibility to transform two known message forwarding protocols ([21]) into snap-stabilizing ones. We use a different scheme for both of them but we prove that these two schemes do not significantly modify time and space complexities of these protocols. Consequently, the main contribution of this paper is to show that it is possible to provide stronger safety properties without significant overcost.

The sequel of this paper is organized as follows: we present first our model (section 2). We quickly survey the seminal work of [21] in section 3. Then we give, prove, and analyze our two solutions (sections 4 and 5). Finally, we conclude by some remarks and open problems (section 6).

2 Model and definitions

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 (*i.e.* the *distance* 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 the classical *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 *label* is a name to refer to the rule in the discussion. 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 *enabled* for p in γ if and only if the guard of A is satisfied by p in γ. Processor p is *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 infinitely often is chosen by the daemon infinitely 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 networks

Today, most of computer networks use a variant of the *message-switching* method (also called *store-and-forward* method). It is why we have choose 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 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 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 is forwarded at least one time or delivered to its destination.

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 any 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 use these two remarks to prove snap-stabilization of our protocol in the following of this paper.

3 Fault-free protocols

In this section, we survey the seminal work of [21][1]. Remind that this work assume that routing tables are correct in the initial configuration. 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 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.

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). A such algorithm is a *controller*. If a controller $\mathcal{C}$ ensure the following property: in any execution, $\mathcal{C}$ prevents the network to reach a deadlock, then $\mathcal{C}$ is a *deadlock-free* controller.

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 a Destination d (since all messages follow the same path).

Then, a deadlock-free controller which prevents also livelocks and message losses satisfies the specification of the message forwarding problem.

In the case where routing table are initially correct, [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. In the sequel of this section, we present the two buffer graph which we use in our snap-stabilizing protocols.

"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 a such graph in Figure 1.

Since each connected component of this graph is a tree, this oriented graph is acyclic. Consequently, [21] allows us to define a deadlock-free controller on this graph. Note that this scheme use n buffers per processor. So, we need n^2 buffers on the whole network.

[1] The reader is referred to [24] to find a much detailed description of this work.

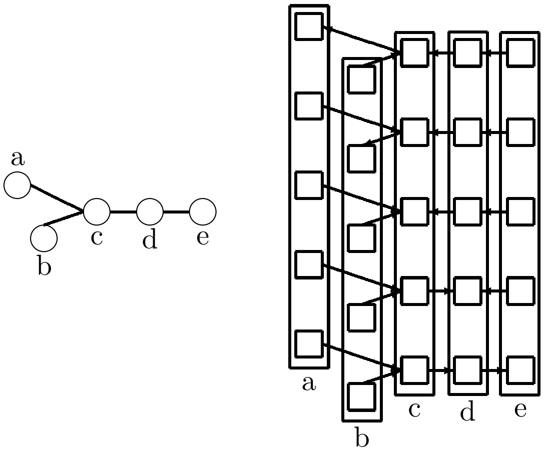

Figure 1: Example of a ”destination-based” buffer graph (on the right) on the network of the left.

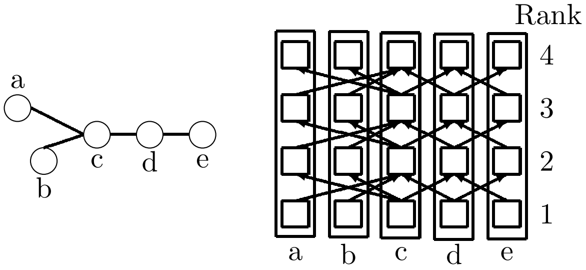

Figure 2: Example of a ”distance-based” buffer graph (on the right) on the network of the left.

”Distance-based” buffer graph. In this scheme, each processor have $D+1$ buffers ranked from 1 to $D+1$ (remind that D is the diameter of the network). New messages are always generated in the buffer of rank 1 of the sending 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. 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. The reader can find an example of such a graph in Figure 2.

Since messages always ”come upstairs” the buffer rank, this oriented graph is acyclic. Consequently, [21] allows us to define a deadlock-free controller on this graph. Note that this scheme use $D+1$ buffers per processor. So, we need $n(D+1)$ buffers on the whole network.

4 First protocol

4.1 Informal description

The main idea of this section is to adapt the ”destination-based” scheme (see Section 3) 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

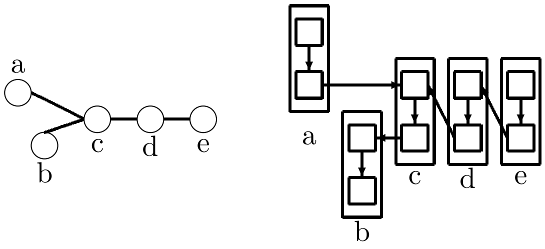

Figure 3: Example of our buffer graph (on the right) for Destination b on the network (on the left).

both algorithms always chooses the action of $\mathcal{A}$). This guarantees us that routing tables are correct and constant in 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 now describe our buffer graph adapted from the "destination-based" one. Our buffer graph is composed of n connected components, each associated to a destination d and based on the oriented tree T_d (remind that T_d is the tree induced by routing table for Destination d). Consequently, we can 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 3). 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 consequences 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 carried 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 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 4. 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.

4.2 Algorithm

We now present formally our protocol in Algorithm 1. We call it $\mathcal{SSMFP}_1$ for $\mathcal{S}$nap-$\mathcal{S}$tabilizing $\mathcal{M}$essage $\mathcal{F}$orwarding $\mathcal{P}$rotocol 1. 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).

4.3 Proof of correctness

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 is deliver to its destination in a finite time.*

In this section, we prove that $\mathcal{SSMFP}_1$ is a snap-stabilizing message forwarding protocol for specification $\mathcal{SP}$. For that, we are going to prove successively that:

1. $\mathcal{SSMFP}_1$ is a snap-stabilizing message forwarding protocol for specification $\mathcal{SP}'$ if routing tables are correct in the initial configuration (Lemmas 1, 2, 3 and Proposition 1).
2. $\mathcal{SSMFP}_1$ is a self-stabilizing message forwarding protocol for specification $\mathcal{SP}'$ even if routing tables are corrupted in the initial configuration (Proposition 2).
3. $\mathcal{SSMFP}_1$ is a snap-stabilizing message forwarding protocol for specification $\mathcal{SP}$ even if routing tables are corrupted in the initial configuration (Lemmas 4, 5 and Theorem 1).

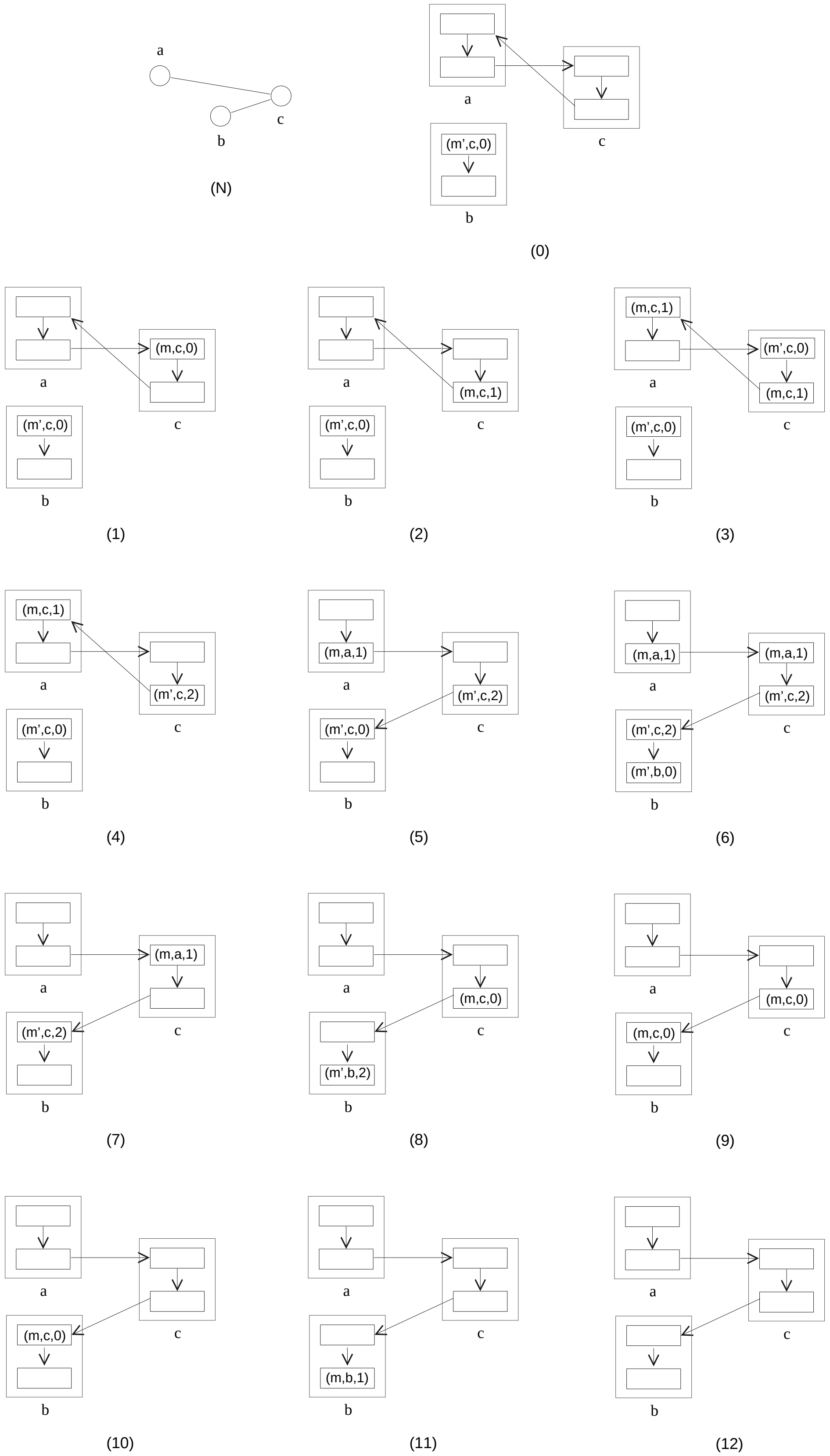

Figure 4: An example of execution of our first algorithm.

Algorithm 1 ($\mathcal{SSMFP}_1$): 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 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.

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 */

$\boldsymbol{(R_1)}$ :: $request_p \wedge (nextDestination_p = d) \wedge (bufR_p(d) = \varepsilon) \wedge (choice_p(d) = p) \longrightarrow bufR_p(d) := (nextMessage_p, p, 0); request_p := false$

/* Rule for the internal forwarding of a message */

$\boldsymbol{(R_2)}$ :: $(bufE_p(d) = \varepsilon) \wedge (bufR_p(d) = (m, q, c)) \wedge ((q = p) \vee (bufE_q(d) \neq (m, q', c))) \longrightarrow bufE_p(d) := (m, p, color_p(d)); bufR_p(d) := \varepsilon$

/* Rule for the forwarding of a message */

$\boldsymbol{(R_3)}$ :: $(bufR_p(d) = \varepsilon) \wedge (choice_p(d) = s) \wedge (s \neq p) \wedge (bufE_s(d) = (m, q, c)) \longrightarrow bufR_p(d) := (m, s, c)$[1]

/* Rule for the erasing of a message after its forwarding */

$\boldsymbol{(R_4)}$ :: $(bufE_p(d) = (m, q, c)) \wedge (p \neq d) \wedge (bufR_{nextHop_p(d)}(d) = (m, p, c)) \wedge (\forall r \in N_p \backslash \{nextHop_p(d)\}, bufR_r(d) \neq (m, p, c)) \longrightarrow bufE_p(d) := \varepsilon$

/* Rule for the erasing of a message after its duplication */

$\boldsymbol{(R_5)}$ :: $(bufR_p(d) = (m, q, c)) \wedge (bufE_q(d) = (m, q', c)) \wedge (nextHop_q(d) \neq p) \longrightarrow bufR_p(d) := \varepsilon$

/* Rule for the consumption of a message */

$\boldsymbol{(R_6)}$ :: $(bufE_p(p) = (m, q, c)) \longrightarrow deliver_p(m); bufE_p(p) := \varepsilon$

[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 this does not improve the performance of $\mathcal{SSMFP}_1$.

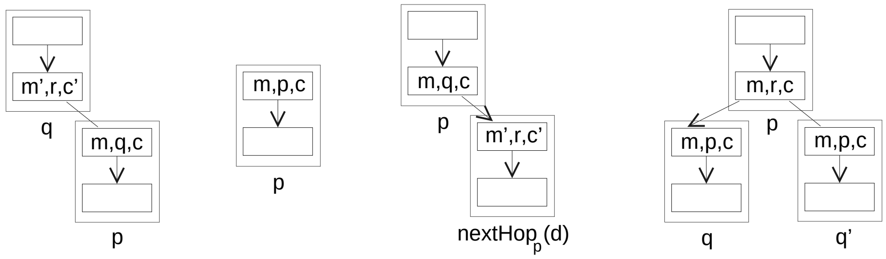

Figure 5: Examples of caterpillar associated to m on p (from left to right: two of type 1, one of type 2 and one of type 3).

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.

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 5 an example for each type of caterpillar. Remark that an emission buffer can belong to several caterpillars of type 3.

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 $\mathcal{SSMFP}_1$ products in 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 $nextHop_p(d)$.*

Proof. Let γ be a configuration in which routing tables are correct. Let m (of Destination d) be a message existing in γ. Let $C = bufR_p(d)$ be a caterpillar of type 1 associated to m. Denote by δ the distance between p and d ($\delta = dist(p, d)$). We are going to prove the result by induction on δ. We define the following predicate:

(P_δ): if $C = bufR_p(d)$ is a caterpillar of type 1 associated to m such that $dist(p, d) = \delta$, then, under a weakly fair daemon, the execution of $\mathcal{SSMFP}_1$ products one of the following effect in a finite time:

- m is delivered to d.

- C disappeared on p and there exists a caterpillar of type 1 associated to the same message on $nextHop_p(d)$.

Initialization: We are going to prove that $(\boldsymbol{P_0})$ is true.
Let $C = bufR_p(d)$ be a caterpillar of type 1 associated to m such that $dist(p, d) = 0$. This implies that $p = d$. Let be $bufR_p(d) = (m, q, id)$. We must distinguish two cases :

Case 1: $bufE_p(d) \neq \varepsilon$.
The rule $(\boldsymbol{R_6})$ is enabled for the processor p. We can observe that this rule can not be neutralized. Since we assumed a weakly fair daemon, we obtain that p executes $(\boldsymbol{R_6})$ in a finite time. We can then consider the case 2 since this rule erases the content of $bufE_p(d)$.

Case 2: $bufE_p(d) = \varepsilon$.
By the definition of a caterpillar of type 1, $(\boldsymbol{R_2})$ is enabled for p. This rule can be neutralized if and only if $bufE_q(d)$ is occupied by (m, q', id). This is impossible by the construction of $color_q(d)$. Since we assume a weakly fair daemon, we obtain that p executes $(\boldsymbol{R_2})$ in a finite time. C disappears and a new caterpillar of type 2 appears in $bufE_p(d)$. By the same reasoning of the case 1, we can say that p executes $(\boldsymbol{R_6})$ in a finite time. This implies that m is delivered to d.

We proved that $(\boldsymbol{P_0})$ is true.

Induction: Let $\delta \geq 1$. We assume that $(\boldsymbol{P_{\delta-1}})$ is true. We are going to prove that then $(\boldsymbol{P_\delta})$ is true.
Let $C = bufR_p(d)$ be a caterpillar of type 1 associated to m such that $dist(p, d) = \delta$. Let be $bufR_p(d) = (m, q, id)$. We must distinguish two cases:

Case 1: $bufE_p(d) \neq \varepsilon$.
Let be $r = nextHop_p(d)$.

Case 1.1: $bufE_p(d)$ is occupied by a caterpillar C' of type 2.
By the definition of a caterpillar of type 2, either $(\boldsymbol{R_3})$ or $(\boldsymbol{R_1})$ is enabled on r if and only if $bufR_r(d) = \varepsilon$.

Case 1.1.a: If $bufR_r(d) = \varepsilon$, then r executes $(\boldsymbol{R_3})$ or $(\boldsymbol{R_1})$ (since we assumed a weakly fair daemon and these rules cannot be neutralized). The result of this execution depends on the value of $choice_r(d)$:

- If $choice_r(d) = p$, then C' becomes a caterpillar of type 3. We are now in the case 1.2.
- If $choice_r(d) \neq p$, then a message $(m', choice_r(d), id')$ is forwarded in $bufR_r(d)$. So, C' remains a caterpillar of type 2 and we are in the case 1.1.b. It is important to remark that the fairness of $choice_r(d)$ guarantees us that this case cannot appear infinitely.

Case 1.1.b: If $bufR_r(d) = (m', q', id')$, then we can distinguish two cases:

- If $bufR_r(d)$ belongs to at least one caterpillar of type 3, we can apply the reasoning of the case 1.2 to $bufE_{q'}(d)$ and conclude that $bufR_r(d)$ belongs to a caterpillar of type 1 in a finite time.
- If $bufR_r(d)$ belongs to a caterpillar of type 1, we can say that $bufR_r(d)$ becomes empty in a finite time by application of $(\boldsymbol{P_{\delta-1}})$ ($dist(r, d) = dist(p, d) - 1 = \delta - 1$ since routing tables are correct). Then, we are on the case 1.1.a.

We can conclude that $bufE_p(d)$ belongs to a caterpillar of type 3 associated to m in a finite time. So, we are on the case 1.2.

Case 1.2: $bufE_p(d)$ belongs to at least one caterpillar of type 3.

Case 1.2.a: $bufE_p(d)$ belongs to at least two caterpillars of type 3.
This implies that there exists $x \in N_p \backslash \{r\}$, $bufR_x(d) = (m, p, id)$. The processor x is enabled by $(\boldsymbol{R_5})$ infinitely (since routing tables are correct and p cannot erase $bufE_p(d)$ by the construction of $(\boldsymbol{R_4})$). Since we assumed a weakly fair daemon, (m, p, id) is erased from $bufR_x(d)$ in a finite time. We can repeat this reasoning until $bufE_p(d)$ belongs to only one caterpillar of type 3 since the construction of $(\boldsymbol{R_3})$ guarantees us that it is impossible to create a new caterpillar of type 3 involving $bufE_p(d)$. So, we are on the case 1.2.b.

Case 1.2.b: $bufE_p(d)$ belongs to only one caterpillar of type 3.
By the definition of a caterpillar of type 3, we can say that p is enabled for $(\boldsymbol{R_4})$. The construction of $(\boldsymbol{R_3})$ guarantees us that it is impossible to create a new caterpillar of type 3 involving $bufE_p(d)$, also $(\boldsymbol{R_3})$ is not neutralized. As we assumed a weakly fair daemon, p executes $(\boldsymbol{R_4})$ in a finite time. Then, $bufE_p(d)$ is empty in a finite time, we are in the case 2.

We can conclude the case 1 by the following affirmation : we are in the case 2 in a finite time.

Case 2: $bufE_p(d) = \varepsilon$.
By the definition of a caterpillar of type 1, p is enabled by $(\boldsymbol{R_2})$. By the construction of $color_q(d)$ and of $(\boldsymbol{R_2})$ (for q), $(\boldsymbol{R_2})$ cannot neutralized for p. Since we assumed a weakly fair daemon, we can say that p executes $(\boldsymbol{R_2})$ in a finite time. This implies that C disappears and a new caterpillar C' of type 2 associated to m appears. We can now apply the reasoning of the case 1 to deduce that C' becomes a caterpillar of type 1 on r in a finite time.

We have proved that $(\mathbf{P}_\delta)$ is true, that ended this proof. □

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

Proof. Let p be a processor which has a message m (of Destination d) to send. As p has a waiting message, we have $request_p = true$ whatever its value in the initial configuration. We must now study two cases:

Case 1: $bufR_p(d) = \varepsilon$.
The processor p executes either $(\boldsymbol{R_3})$ or $(\boldsymbol{R_1})$ in a finite time (since we assumed a weakly fair daemon and these rules cannot be neutralized). The result of this execution depends on the value of $choice_p(d)$:

- If $choice_p(d) = p$, then p executes $(\boldsymbol{R_1})$ in a finite time, we obtain the result.
- If $choice_p(d) \neq p$, then p executes $(\boldsymbol{R_3})$ in a finite time. Consequently, $bufR_p(d)$ is occupied by a caterpillar of type 3. So, we are in the case 2.1. Note that the fairness of $choice_p(d)$ guarantees us that this case cannot appear infinitely.

Case 2: $bufR_p(d) = (m', q, id)$.

Case 2.1: $bufR_p(d)$ belongs to a caterpillar C of type 3.
We can apply the reasoning of the case 1.2 of the proof of Lemma 1 to $bufE_q(d)$ and conclude that C becomes a caterpillar of type 1 in a finite time. We are now in the case 2.2.

Case 2.2: $bufR_p(d)$ belongs to a caterpillar C of type 1.
We can apply Lemma 1 to C and say that $bufR_p(d)$ becomes empty in a finite time. We are now in the case 1.

By the remark of the case 1, this reasoning is finite, that proves the result. □

Lemma 3 *If a message m is generated by $\mathcal{SSMFP}_1$ in a configuration in which routing tables are correct, $\mathcal{SSMFP}_1$ delivers m to its destination in a finite time under a weakly fair daemon.*

Proof. Assume that routing tables are correct when $\mathcal{SSMFP}_1$ accepts a message m (of Destination d) on Processor p. This implies that p generated m executing rule $(\boldsymbol{R_1})$. This rule leads to the creation of a caterpillar of type 1 associated to m in $bufR_p(d)$. Since routing tables are assumed correct and constant, the result follows from $dist(p,d)+1$ applications of Lemma 1. □

Proposition 1 *$\mathcal{SSMFP}_1$ 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. To prove that $\mathcal{SSMFP}_1$ is a snap-stabilizing message forwarding protocol for specification $\mathcal{SP}'$, we must prove that :

1. If a processor p requests to send a message, then the protocol is initiated by at least one starting action on p in a finite time. In our case, the starting action is the execution of $(\boldsymbol{R_1})$. Lemma 2 proves this property.
2. After a starting action, the protocol is executed according 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 satisfied (following Lemma 2 and the fact that the waiting for the sending of new messages is blocking).
   - The second property of $\mathcal{SP}'$ is always satisfied (following Lemma 3).

Consequently, we deduce the proposition. □

Proposition 2 *$\mathcal{SSMFP}_1$ 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. Remind that $\mathcal{A}$ is a self-stabilizing silent algorithm for computing routing tables running simultaneously to $\mathcal{SSMFP}_1$. Moreover, we assumed that $\mathcal{A}$ has priority over $\mathcal{SSMFP}_1$ (*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 constant in a finite time regardless of the initial state.

By Proposition 1, $\mathcal{SSMFP}_1$ is a snap-stabilizing message forwarding protocol for specification $\mathcal{SP}'$ when it starts from a such configuration. Consequently, we obtain the proposition. □

Lemma 4 *Under a weakly fair daemon, $\mathcal{SSMFP}_1$ does not delete a valid message without deliver it to its destination even if $\mathcal{A}$ runs simultaneously.*

Proof. By contradiction, let m be a valid message which is deleted without being delivered to its destination.

By the construction of the rule $(\boldsymbol{R_2})$, this cannot be the result of an internal forwarding since the message is sequentially copied in $bufE_p(d)$ and erased from $bufR_p(d)$.

By the construction of rules $(\boldsymbol{R_5})$ and $(\boldsymbol{R_4})$, this cannot be the result of the execution of $(\boldsymbol{R_5})$ (since we are guaranteed that m is in $bufE_q(d)$ and cannot be erased from this buffer simultaneously).

By the construction of rules $(\boldsymbol{R_4})$ and $(\boldsymbol{R_2})$, m cannot be erased from $bufR_{nextHop_p(d)}(d)$ in the step in which it is erased from $bufE_p(d)$.

Since we have seen that a simultaneous erasing is impossible, the hypothesis implies that m is erased from a buffer $bufE_p(d)$ without being copied in another buffer.

The only rule which erases a message from $bufE_p(d)$ and does not deliver m is $(\boldsymbol{R_4})$. If a processor p executes this rule, then we have $bufE_p(d) = (m, q, id)$ and $bufR_{nextHop_p(d)}(d) = (m, p, id)$. Assume that the message contained by $bufR_{nextHop_p(d)}(d)$ is not the result of the application of rule $(\boldsymbol{R_3})$ on $bufE_p(d)$. If this message was in $bufR_{nextHop_p(d)}(d)$ before m came in $bufE_p(d)$, we obtain a contradiction with the definition of $color_p(d)$. This implies that this message came in $bufR_{nextHop_p(d)}(d)$ after m came in $bufE_p(d)$. Then, the construction of $(\boldsymbol{R_3})$ allows us to say that $bufR_{nextHop_p(d)}(d)$ contains a message (m, q', id) with $q' \neq p$ (since we have supposed that the message does not come from $bufE_p(d)$). We obtain a contradiction. We can conclude that, when we have $bufE_p(d) = (m, q, id)$ and $bufR_{nextHop_p(d)}(d) = (m, p, id)$, the message m has been copied at least one time. This result contradicts the existence of m. □

Lemma 5 *Under a weakly fair daemon, $\mathcal{SSMFP}_1$ never duplicates a valid message even if $\mathcal{A}$ runs simultaneously.*

Proof. Since the emission of a message creates one caterpillar of type 1 by the construction of the rule $(\boldsymbol{R_1})$, it remains to prove 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.

Let C be a caterpillar of type 1 associated to a message m (of Destination d) on a processor p. Since $(\boldsymbol{R_5})$ is not enabled for p (by definition of a caterpillar of type 1), m is erased from $bufR_p(d)$ by $(\boldsymbol{R_2})$. So, m is still present on p (since it has been copied in $bufE_p(d)$). Then, we have two cases to observe:

Case 1: $p = d$.

The only rule for erasing m which can be enabled is $(\boldsymbol{R_6})$. This rule delivers m to its destination.

Case 2: $p \neq d$.

The only rule for erasing m which can be enabled is $(\boldsymbol{R_4})$. The construction of this rule implies the announced property.

We can conclude that m is delivered at most once to its destination, that proves the result. □

Theorem 1 *$\mathcal{SSMFP}_1$ 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.*

Proof. Proposition 2 and Lemma 4 allows us to conclude that $\mathcal{SSMFP}_1$ 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 5, we obtain the result. □

4.4 Time complexities

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

In order to lighten this paper, we present only key ideas of this section proofs.

Proposition 3 *For any Processor d, $\mathcal{SSMFP}_1$ delivers $2n$ invalid messages to d in the worst case.*

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 invalid messages are delivered to their destination, that allows us to reach the announced bound. □

Proposition 4 *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 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 (see the proof of Lemma 1).

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 $\sum\limits_{\delta=D}^{0} O(\Delta^\delta) \in O(\Delta^D)$ 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 $\mathcal{SSMFP}_1$). □

Proposition 5 *The delay (waiting time before the first emission) and the waiting time (between two consecutive emissions) of $\mathcal{SSMFP}_1$ 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 is generated after at most $(\Delta - 1)$ releases of $bufR_p(d)$ (see proof of Lemma 1). The result of Proposition 4 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 4 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 6 *The amortized complexity (to forward a message) of $\mathcal{SSMFP}_1$ 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 $\mathcal{SSMFP}_1$ delivers at least one message to d in the $3D$ rounds following γ.

The proof of this property is done as follows. Let δ be the smallest number such that there exists a message of Destination d on a processor p which satisfy $dist(p, d) = \delta$. Then, we prove that, after at most three rounds, there exists a message (not necessarily m) on a processor p' which satisfies $dist(p', d) = \delta - 1$. Since $\delta \leq D$ in γ, we obtain the announced property.

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 previous property, we can say that $\mathcal{SSMFP}_1$ 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. □

4.5 Conclusion

In this section, we prove that we can adapt the "destination-based" deadlock-free controller defined in [21] to obtain a snap-stabilizing message forwarding algorithm. Our algorithm is mainly based on the control of effects of routing tables moves on message. This control is performed in two ways. Firstly, we "slow down" messages by using two buffers per processor in order to control the number of copy of a same message in the network at a given time. Secondly, we use a specific flag to avoid message merge or duplication.

The initial fault-free protocol uses n^2 buffers for the whole network and our protocol uses $2n^2$ buffers. Consequently, our protocol ensures a stronger safety and fault-tolerance with respect the initial one without a significant overcost in space. Our time analysis (see Section 4.4) shows that this stronger safety does not leads to an overcost in time.

5 Second protocol

5.1 Informal description

In this section, we give a second snap-stabilizing message forwarding protocol adapted to the "distance-based" deadlock-free controller (see Section 3). 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 (*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 are correct and constant in 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.

Our idea is as follows. We choose exactly the same graph buffer as [21] and we allow the erasing of a message only if we are assured that the message has been delivered to its destination. In this goal, we use an acknowledgment scheme which guarantees the reception of the message.

More precisely, we associate to each copy of the message a type which has 3 values: S (Sending), A (Acknowledgment) and F (Fail). Forwarding of a valid message follows the above scheme:

1. Generation with type S in a buffer of rank 1.
2. Forwarding (with copy in buffers of increasing rank) with type S without any erasing.
3. If the message reaches its destination :
   (a) It is delivered and the copy of the message takes type A.
   (b) Type A is propagated to the sink of the message following the income path.
   (c) Buffers are allowed to free themselves once the type A is propagated to the previous buffer on the path.
   (d) The sink erases its copy, that performs the erasing of the message.
4. Otherwise, (the message reaches a buffer of rank $D + 1$ without cross its destination) :
   (a) The copy of the message takes type F.
   (b) Type F is propagated to the sink of the message following the income path.
   (c) Buffers are allowed to free themselves once the type F is propagated to the previous buffer on the path.
   (d) Then, the sink of the message gives the type S to its copy, that begin a new cycle (the message 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.

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 reached its destination before) and it is then 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 cross over by the message, the identity of the destination of the message and the type of the message (S, 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).

5.2 Algorithm

We now present formally our protocol in Algorithm 2. We call it $\mathcal{SSMFP}_2$ for $\mathcal{S}$nap-$\mathcal{S}$tabilizing $\mathcal{M}$essage $\mathcal{F}$orwarding $\mathcal{P}$rotocol 2.

5.3 Proof of correctness

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

Specification 3 ($\mathcal{SP}'$) *Specification of message forwarding problem allowing duplication.*

Algorithm 2 $\mathcal{SSMFP}_2$: Message forwarding protocol for processor p.

Data:
- n, D : natural numbers 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, $c \in \{S, A, F\}$ type 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 given 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 $((choice_p(i) \in N_p) \wedge (buf_{choice_p(i)}(i-1) = (m, p, q, d, S)) \wedge (choice_p(i) \neq d))$. 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) :: $request_p \wedge (buf_p(1) = \varepsilon) \wedge (nextDest_p = d) \wedge (nextMes_p = m) \wedge (buf_{nextHop_p(d)}(2) \neq (m, r', p, d, c)) \longrightarrow buf_p(1) := (m, nextHop_p(d), r, d, S)$ with $r \in N_p$; $request_p := false$
    /* Processing of acknowledgment */
    (R_2) :: $(buf_p(1) = (m, r, q, d, F)) \wedge (d \neq p) \wedge (buf_r(2) \neq (m, r', p, d, F)) \longrightarrow buf_p(1) := (m, nextHop_p(d), q, d, S)$
    (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) :: $buf_p(1) = (m, r, q, p, S) \longrightarrow deliver_p(m);\ buf_p(1) := (m, r, q, p, A)$
    (R_5) :: $buf_p(1) = (m, r, q, p, A) \longrightarrow buf_p(1) := \varepsilon$
    (R_6) :: $buf_p(1) = (m, r, q, p, F) \longrightarrow buf_p(1) := (m, r, q, p, S)$

/* Rule for buffers of rank 1 to $D$ : propagation of acknowledgment */
    (R_7) :: $\exists i \in \{1, ..., D\}, ((buf_p(i) = (m, r, q, d, S)) \wedge (p \neq d) \wedge (buf_r(i+1) = (m, r', p, d, c)) \wedge (c \in \{R, A\})) \longrightarrow buf_p(i) := (m, r, q, d, c)$

/* Rules for buffers of rank 2 to $D$ */
    /* Forwarding of messages */
    (R_8) :: $\exists i \in \{2, ..., D\}, ((buf_p(i) = \varepsilon) \wedge (choice_p(i) = s) \wedge (buf_s(i-1) = (m, p, q, d, S)) \wedge (buf_{nextHop_p(d)}(i+1) \neq (m, r, p, d, c))) \longrightarrow buf_p(i) := (m, nextHop_p(d), s, d, S)$
    /* Erasing of messages of which the acknowledgment has been forwarded */
    (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 $A$ */
    (R_10) :: $\exists i \in \{2, ..., D+1\}, buf_p(i) = (m, r, q, p, S) \longrightarrow deliver_p(m);\ buf_p(i) := (m, r, q, p, A)$
    /* Erasing of messages of destination $p$ of which the acknowledgment has been forwarded */
    (R_11) :: $\exists i \in \{2, ..., D+1\}, ((buf_p(i) = (m, r, q, p, c)) \wedge (c \in \{F, A\}) \wedge (buf_q(i-1) = (m, p, q', p, c))) \longrightarrow buf_p(i) := \varepsilon$
```

End of Algorithm 2:

```
/* Rules for the buffer of rank $D+1$ */
    /* Forwarding of messages */
    $(\boldsymbol{R_{12}})$ :: $(buf_p(D+1) = \varepsilon) \wedge (choice_p(D+1) = s) \wedge (buf_s(D) = (m,p,q,d,S)) \longrightarrow buf_p(D+1) := (m, nextHop_p(d), s, d, S)$ /* Generation of the acknowledgment $F$ */
    $(\boldsymbol{R_{13}})$ :: $(buf_p(D+1) = (m,r,q,d,S)) \wedge (d \neq p) \longrightarrow buf_p(D+1) := (m,r,q,d,F)$
    /* 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 $F, A$ (cf. definitions below) */
    $(\boldsymbol{R_{15}})$ :: $\exists i \in \{2,...,D\}, ((buf_p(i) = (m,r,q,d,c)) \wedge (c \in \{F,A\}) \wedge (buf_r(i+1) \neq (m,r',p,d,c)) \wedge (buf_q(i-1) \neq (m,p,q',d,c'))) \longrightarrow buf_p(i) := \varepsilon$
    $(\boldsymbol{R_{16}})$ :: $\exists i \in \{2,...,D\}, ((buf_p(i) = (m,r,q,d,c)) \wedge (c \in \{F,A\}) \wedge (buf_r(i+1) \neq (m,r',p,d,c)) \wedge (buf_q(i-1) = (m,p,q',d,c')) \wedge (c' \in \{F,A\}\backslash\{c\} \vee q = d)) \longrightarrow buf_p(i) := \varepsilon$
    $(\boldsymbol{R_{17}})$ :: $(buf_p(D+1) = (m,r,q,d,c)) \wedge (c \in \{F,A\}) \wedge (buf_q(D) \neq (m,p,q',d,c')) \longrightarrow buf_p(D+1) := \varepsilon$
    $(\boldsymbol{R_{18}})$ :: $(buf_p(D+1) = (m,r,q,d,c)) \wedge (c \in \{F,A\}) \wedge (buf_q(D) = (m,p,q',d,c')) \wedge (c' \in \{F,A\}\backslash\{c\} \vee q = d) \longrightarrow buf_p(D+1) := \varepsilon$
```

- *Any message can be send out in a finite time.*
- *Any valid message is delivered to its destination in a finite time.*

In this section, we prove that $\mathcal{SSMFP}_2$ is a snap-stabilizing message forwarding protocol for specification $\mathcal{SP}$. For that, we are going to prove successively that:

1. Copies of a same message have a particular structure. Then, we prove some properties on the behavior of these structures under $\mathcal{SSMFP}_2$ (Lemmas 6, 7, 8, and 9).
2. $\mathcal{SSMFP}_2$ is a snap-stabilizing message forwarding protocol for specification $\mathcal{SP}'$ if routing tables are correct in the initial configuration (Lemmas 10, 11, 12 and Proposition 7).
3. $\mathcal{SSMFP}_2$ is a self-stabilizing message forwarding protocol for specification $\mathcal{SP}'$ even if routing tables are corrupted in the initial configuration (Proposition 8).
4. $\mathcal{SSMFP}_2$ is a snap-stabilizing message forwarding protocol for specification $\mathcal{SP}$ even if routing tables are corrupted in the initial configuration (Lemmas 13, 14 and Theorem 2).

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 loose one of them thanks to the use of the flag.

Preliminaries. In a first time, we define a particular structure of messages and we study the behavior of these structure under $\mathcal{SSMFP}_2$. 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.

Definition 4 (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:*

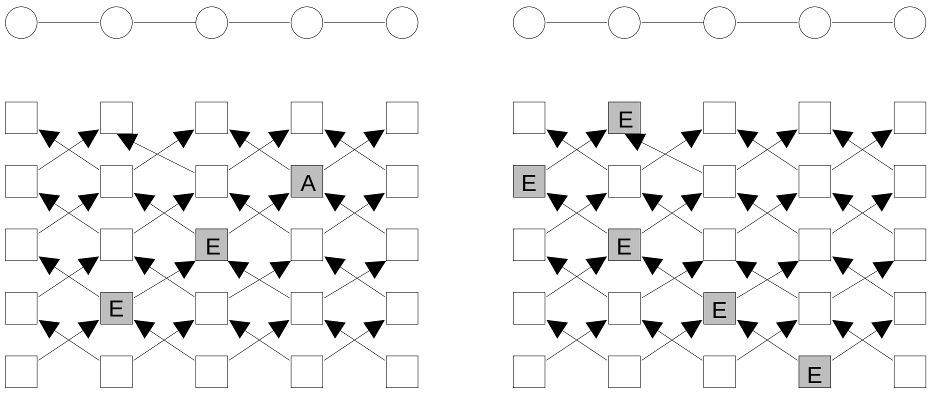

Figure 6: Examples of caterpillar (at left: abnormal of type A, at right: normal of type E).

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

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.

We give now some characterization for caterpillars.

Definition 5 (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 otherwise ($i \geq 2$).*
- *C_m is a caterpillar of type S if $\forall j \in \{1, ..., t\}$, $c_j = S$ (*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. *$k < t+1$).*

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 S, A or F. The reader can find in Figure 6 an example for some type of caterpillar.

Lemma 6 *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 a finite time or become a normal caterpillar of type F (resp. A).*

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

1. By 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 can not 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 these rule in a finite time. We can repeat this reasoning $k-1$ times on Processors $p_{k-1}, ..., p_1$. Then, we obtain a caterpillar which all buffers are on type c in a finite time.

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

   Case 1: $p_t = d$.
   Processor p_t is the enabled for rule $(\boldsymbol{R_{11}})$ by 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 infinitely enabled for p_t. Since the daemon is weakly fair, p_t executes this rule in a finite time. Then, $buf_{p_t}(i + t - 1)$ is empty in a 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 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 infinitely enabled for p_t. Since the daemon is weakly fair, p_t executes this rule in a finite time. Then, $buf_{p_t}(i + t - 1)$ is empty in a 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 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 $c \neq S$). Consequently, this rule remains infinitely enabled for p_t. Since the daemon is weakly fair, p_t executes this rule in a finite time. Then, $buf_{p_t}(i + t - 1)$ is empty in a finite time.

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

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 above, these rule remains infinitely enabled. Since the daemon is weakly fair, p_1 executes this rule in a finite time. Consequently, $buf_{p_1}(i)$ becomes empty in a finite time. Then, C_m disappears.

   Case 1.2: $q \neq d$.
   Assume that $c' = S$. 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 merge of two caterpillars of type c. Then, consider the new caterpillar $C'_m = buf_{p'_1}(i')...buf_{p'_{t'}}(i'+t'-1)$ (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 restart the reasoning (we are ensured that this reasoning is finite since we have $1 \leq i' < i$ 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 above, these rule remains infinitely enabled. Since the daemon is weakly fair, p_1 executes this rule in a finite time. Consequently, $buf_{p_1}(i)$ becomes empty in a 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 above, these rule remains infinitely enabled. Since the daemon is weakly fair, p_1 executes this rule in a finite time. Consequently, $buf_{p_1}(i)$ becomes empty in a finite time. Then, C_m disappears.

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

Lemma 7 *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 a finite time.*

Proof. Let γ be a configuration and m be a message of Destination d existing in γ. Let $C_m = buf_{p_1}(1)...buf_{p_t}(t)$ $(t \geq 1)$ 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 infinitely enabled. Since the daemon is weakly fair, p_1 executes this rule in a 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 the case 2.2.2 of the proof of Lemma 6, we can prove that this rule remains infinitely enabled. Since the daemon is weakly fair, p_1 executes this rule in a finite time. Consequently, C_m disappears.

Case 2: $t \geq 2$.
We can apply the reasoning of points 1,2, and 3 of the proof of Lemma 6. That leads us to case 1.2.

In all the cases, C_m disappears in a finite time, that leads us to the lemma. □

Lemma 8 *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 S of length 1 in a finite time.*

Proof. Let γ be a configuration and m be a message of Destination d existing in γ. Let $C_m = buf_{p_1}(1)...buf_{p_t}(t)$ $(t \geq 1)$ 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 infinitely enabled. Since the daemon is weakly fair, p_1 executes this rule in a finite time. Consequently, C_m becomes a caterpillar of type S 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 the case 2.2.2 of the proof of Lemma 6, we can prove that this rule remains infinitely enabled. Since the daemon is weakly fair, p_1 executes this rule in a finite time. Consequently, C_m becomes a caterpillar of type S of length 1.

Case 2: $t \geq 2$.

We can apply the reasoning of points 1,2, and 3 of the proof of Lemma 6. That leads us to case 1.2.

In all cases, we proved that C_m becomes a caterpillar of type S of length 1 in a finite time, that leads us to the lemma. □

Lemma 9 *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 S associated to m becomes a caterpillar of type A or F in a finite time.*

Proof. Let γ be a configuration of the network and m be a message (of Destination d) existing in γ. Let $C_m = buf_{p_1}(i)...buf_{p_t}(i+t-1)$ $(t \geq 1)$ be a caterpillar of type S 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 a finite time.

Initialization: We want to prove that $(\boldsymbol{P_{D+1}})$ is true.

Let $C_m = buf_{p_1}(i)...buf_{p_t}(i+t-1)$ $(t \geq 1)$ be a caterpillar of type S 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 infinitely enabled. Since the daemon is weakly fair, p_t executes this rule in a 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 a 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 infinitely enabled. Since the daemon is weakly fair, p_t executes this rule in a 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 a finite time. Then, Property $(\boldsymbol{P_{D+1}})$ is satisfied.

Induction: Let be $l \leq D$. Assume that $\boldsymbol{(P_{l+1})}...\boldsymbol{(P_{D+1})}$ are satisfied. We want to prove that $\boldsymbol{(P_l)}$ is then satisfied.

Let $C_m = buf_{p_1}(i)...buf_{p_t}(i+t-1)$ $(t \geq 1)$ be a caterpillar of type S 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 infinitely enabled. Since the daemon is weakly fair, p_t executes this rule in a 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 a 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 a finite time. Then, Property $\boldsymbol{(P_l)}$ is satisfied.

Case 2: $p_t \neq d$.

Assume w.l.g. that $buf_{p_t}(i+t-1) = (m, r, q, d, E)$. We want to prove that the head of C_m goes up of one buffer in a finite time. We must study the following cases:

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

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 infinitely enabled for r. Processor r executes this rule in a finite time because the daemon is weakly fair. The result of this execution depends on the value of $choice_r(i+t)$:
   (a) If $choice_r(i+t) = p_t$, then the head of C_m goes up of one buffer when r executes rule $\boldsymbol{(R_{12})}$.
   (b) 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.b. Note that the fairness of $choice_r(i+t)$ ensures us that these case can appear only a finite number of times.
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 whose $buf_r(i+t)$ belongs. Consider the three possible cases:
   (a) $C_{m'}$ is of type S: we can apply the induction hypothesis to $C_{m'}$ since its head stays in a buffer of rank greater or equals to $i+t$. Consequently, $C_{m'}$ becomes a caterpillar of type F or A in a finite time. That leads us to one of the following cases.
   (b) $C_{m'}$ is of type A: following Lemmas 6 and 7, $C_{m'}$ disappears in a finite time. Then, $buf_r(i+t)$ becomes empty. That leads us to point 1.
   (c) $C_{m'}$ is of type F: following Lemmas 6 and 8, $C_{m'}$ disappears or becomes a caterpillar of type S and length 1 in a finite time. In all cases, $buf_r(i+t)$ becomes empty (since $i+t = D+1 \geq 2$). That leads us to point 1.

Case 2.2: $2 \leq i + t \leq D$.

Consider the following cases:

1. $buf_r(i+t) = \varepsilon$.

   Assume w.l.g. 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:

   (a) $C_{m'}$ is of type S: we can apply the induction hypothesis to $C_{m'}$ since its head stays in a buffer of rank greater or equals to $i+t+1$. Consequently, $C_{m'}$ becomes a caterpillar of type F or A in a finite time. That leads us to one of the following cases.

   (b) $C_{m'}$ is of type A: following Lemma 6, $C_{m'}$ disappears in a finite time. Then, $buf_{nextHop_r(d')}(i+t+1)$ becomes empty.

   (c) $C_{m'}$ is of type F: following Lemma 6, $C_{m'}$ disappears in a finite time (it cannot become a caterpillar of type S and length 1 since $buf_r(i+t) = \varepsilon$). Consequently, $buf_{nextHop_r(d')}(i+t+1)$ becomes empty in a finite time.

   Then, Rule $(\boldsymbol{R_8})$ is enabled for r in a finite time. This rule remains infinitely 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 a finite time. The result of this execution is one of the following:

   (a) If $choice_r(i+t) = p_t$, then the head of C_m goes up of one buffer when r executes rule $(\boldsymbol{R_8})$.

   (b) 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 point 2 below. Note that the fairness of $choice_r(i+t)$ ensures us that these case can appear only a finite number of times.

2. If $buf_r(i+t) = (m', r', q', d', c')$, the reasoning is similar to the one of point 2 of case 2.1. Consequently, that leads us to point 1 in a finite time.

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

□

Snap-stabilization when routing tables are correct in the initial configuration. Now, we assume that routing tables are correct in the initial configurations and we prove that $\mathcal{SSMFP}_2$ is a snap-stabilizing algorithm for specification $\mathcal{SP}'$.

Lemma 10 *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 S associated to m becomes a caterpillar of type A in a 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 $C_m = buf_{p_1}(1)...buf_{p_t}(t)$ ($t \geq 1$) be a normal caterpillar of type S associated to m.

By Lemma 9, C_m becomes a caterpillar of type A or F in a 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 8 that C_m becomes a caterpillar of type S of length 1 in a finite time. Then, we have: $C_m = buf_{p_1}(1)$.

Following Lemma 9, C_m becomes a caterpillar of type F or A in a finite time. Assume that C_m becomes a caterpillar of type F. This implies that m have been forwarded D times without reach 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 a finite time. □

Lemma 11 *If routing tables are correct, every processor can generate a first message (*i.e. *it can execute* $(\boldsymbol{R_1})$*) in a finite time under a weakly fair daemon .*

Proof. Let p be a processor of the network which have a message m (of Destination d) to forward. As p have a waiting message, the higher layer put $request_p = true$ whatever 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. We must distinguish the following cases:

Case 1: C_m is of type F. Following Lemma 8, C_m becomes a caterpillar of type S in a finite time. That leads us to case 2.

Case 2: C_m is of type S. Following Lemma 10, C_m becomes a caterpillar of type A in a finite time. That leads us to case 3.

Case 3: C_m is of type A. Following Lemma 7, C_m disappears in a finite time.

In all cases, we obtain that $buf_p(1)$ becomes empty in a 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 9 and 6, C'_m disappear in a finite time (note that the merge with $buf_p(1)$ is impossible since this buffer is empty). Moreover, $buf_{nextHop_p(d)}(2)$ can not be fill by a message of type (m, r', p, d, c) (since $buf_p(1)$ is empty). Consequently, rule $(\boldsymbol{R_1})$ is infinitely enabled for Processor p. As the daemon is weakly fair, p executes this rule in a finite time, that leads to the lemma. □

Lemma 12 *If a message m is generated by $\mathcal{SSMFP}_2$ in a configuration in which routing tables are correct, $\mathcal{SSMFP}_2$ delivers m to its destination in a finite time under a weakly fair daemon.*

Proof. The generation of a message m (of Destination d) by $\mathcal{SSMFP}_2$ results from the execution of rule $(\boldsymbol{R_1})$ by the processor which sends m. This rule creates a normal caterpillar of type S associated to m. Following Lemma 10, this caterpillar becomes a caterpillar of type A in a 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, that ends the proof. □

Proposition 7 *$\mathcal{SSMFP}_2$ 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. To prove that $\mathcal{SSMFP}_2$ is a snap-stabilizing message forwarding protocol for specification $\mathcal{SP}'$, we must prove that :

1. If a processor p requests to send a message, then the protocol is initiated by at least one starting action on p in a finite time. In our case, the starting action is the execution of $\boldsymbol{(R_1)}$. Lemma 11 proves this property.
2. After a starting action, the protocol is executed according 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 satisfied (following Lemma 11 and the fact that the waiting for the sending of new messages is blocking).
   - The second property of $\mathcal{SP}'$ is always satisfied (following Lemma 12).

Consequently, we deduce the proposition. □

Self-stabilization. Now, we assume that routing tables are corrupted in the initial configurations and we prove that $\mathcal{SSMFP}_2$ is a self-stabilizing algorithm for specification $\mathcal{SP}'$.

Proposition 8 *$\mathcal{SSMFP}_2$ 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. Remind that $\mathcal{A}$ is a self-stabilizing silent algorithm for computing routing tables running simultaneously to $\mathcal{SSMFP}_2$. Moreover, we assumed that $\mathcal{A}$ has priority over $\mathcal{SSMFP}_2$ (*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 constant in a finite time regardless of their initial states.

By Proposition 7, $\mathcal{SSMFP}_2$ 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. □

Snap-stabilization. We still assume that routing tables are corrupted in the initial configuration and we prove that $\mathcal{SSMFP}_2$ is a snap-stabilizing algorithm for specification $\mathcal{SP}$.

Lemma 13 *Under a weakly fair daemon, $\mathcal{SSMFP}_2$ does not delete a valid message without delivering it to its destination even if $\mathcal{A}$ runs simultaneously.*

Proof. When $\mathcal{SSMFP}_2$ 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 S associated to m.

While m is not delivered to its destination, we know, by Lemmas 9 and 8, that C_m follows infinitely often the above cycle:

- C_m is of type S and becomes of type F (type A is impossible since m is not delivered).
- C_m is of type F and becomes of type S.

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 14 *Under a weakly fair daemon, $\mathcal{SSMFP}_2$ 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 S 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. Assume that m is delivered. By construction of rules $(\boldsymbol{R_4})$ and $(\boldsymbol{R_{10}})$, C_m becomes of type A. Following Lemma 7, C_m disappears in a finite time. Consequently, m cannot be delivered several times. □

Theorem 2 *$\mathcal{SSMFP}_2$ 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 8 and Lemma 13 allows us to conclude that $\mathcal{SSMFP}_2$ 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 14, we obtain the result. □

5.4 Time complexities

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

In order to lighten this paper, we present only key ideas of this section proofs.

Proposition 9 *In the worst case, $\Theta(nD)$ invalid messages are delivered to Processor d.*

Sketch of 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 deliver to d is in $O(nD)$.

We can obtain the lower boundwith a chain of $n = 2q + 1$ processors labeled $p_1, p_2, ..., p_n$. Assume that all buffers of rank least or equals to $q + 1$ initially contain a message of destination p_{q+1} and other buffers are empty. Moreover, assume that routing tables are initially correct. Then, $\mathcal{SSMFP}_2$ 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 10 *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.*

Sketch of proof. In a first time, one must 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 S associated to m which 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 of one buffer in at most $O(\Delta^{D+1-(i+t-1)})$ round if there exists no abnormal caterpillar whose tail is a buffer of rank greater than $i + t$.

In a second time, it is possible to show that $\mathcal{C}$, the set of abnormal caterpillars in γ looses 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 11 *The delay (waiting time before the first emission) and the waiting time (between two consecutive emissions) of $\mathcal{SSMFP}_2$ is $O(max(R_{\mathcal{A}}, nD\Delta^D))$ 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 is sent after at most $(\Delta - 1)$ releases of $buf_p(1)$. The result of Proposition 10 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 10 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 12 *The amortized complexity (to forward a message) of $\mathcal{SSMFP}_2$ is in $O(max$ $(R_{\mathcal{A}}, D))$ rounds when there exists no invalid messages.*

Sketch of proof. In a first time, we must prove the following property: if γ is a configuration in which at least one caterpillar of type S is present, routing tables are correct, and there exists no invalid messages, then $\mathcal{SSMFP}_2$ 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 $\mathcal{SSMFP}_2$ 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. □

5.5 Conclusion

In this section, we prove that we can adapt the "distance-based" deadlock-free controller defined in [21] to obtain a snap-stabilizing message forwarding algorithm. Our algorithm is mainly based on an acknowledgement scheme. Each message is re-emitted until it reaches its destination. As routing tables stabilize in a finite time, we are ensured that, in the worst case, the message is re-emitted after the end of computation of routing tables. Hence, it can reach its destination normally.

The initial fault-free protocol uses $n(D + 1)$ buffers for the whole network and our protocol uses exactly the same number of buffers. Consequently, our protocol ensures a stronger safety and fault-tolerance with respect the initial one without overcost in space. Our time analysis (see Section 5.4) shows that this stronger safety does not leads to an overcost in time.

6 Conclusion

In this paper, we provide the first algorithms (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 an adaptation of two "buffer graphs" in order to control the effect of routing table moves on messages. Our analysis shows that we ensure snap-stabilization without significant overcost 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. A very important 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 would 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] Erwin M. Bakker, Jan van Leeuwen, and Richard B. Tan. Prefix routing schemes in dynamic networks. *Computer Networks and ISDN Systems*, 26(4):403–421, 1993.

[2] Alain Bui, Ajoy Kumar Datta, Franck Petit, and Vincent Villain. State-optimal snap-stabilizing pif in tree networks. In *WSS*, pages 78–85, 1999.

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

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

[5] Thomas Cormen, Charles Leierson, Ronald Rivest, and Clifford Stein. *Introduction à l'algorithmique*. Eyrolles, seconde edition, 2002.

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

[7] Sylvie Delaët, Stéphane Devismes, Mikhail Nesterenko, and Sébastien Tixeuil. Snap-stabilization in message-passing systems. *CoRR*, abs/0802.1123, 2008.

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

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

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

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

[12] José 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] Michele Flammini and Giorgio Gambosi. On devising boolean routing schemes. *Theor. Comput. Sci.*, 186(1-2):171–198, 1997.

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

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

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

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

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

[19] Rastislav Kralovic and Peter 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] Philip M. Merlin and Paul J. Schweitzer. Deadlock avoidance in store-and-forward networks. In *Jerusalem Conference on Information Technology*, pages 577–581, 1978.

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

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

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

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

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

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

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

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

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