Snap-Stabilizing Linear Message Forwarding[*]

Alain Cournier[1], Swan Dubois[2], Anissa Lamani[1], Franck Petit[2], and Vincent Villain[1]

[1] MIS, Université of Picardie Jules Verne, France
[2] LiP6/CNRS/INRIA-REGAL, Université Pierre et Marie Curie - Paris 6, France

Abstract. In this paper, we present the first snap-stabilizing message forwarding protocol that uses a number of buffers per node being independent of any global parameter, that is 4 buffers per link. The protocol works on a linear chain of nodes, that is possibly an overlay on a large-scale and dynamic system, *e.g.,* Peer-to-Peer systems, Grids... Provided that the topology remains a linear chain and that nodes join and leave "neatly", the protocol tolerates topology changes. We expect that this protocol will be the base to get similar results on more general topologies.

Key words: Dynamicity, Message Forwarding, Peer-to-Peer, Scalability, Snap-stabilization

1 Introduction

These last few years have seen the development of large-scale distributed systems. Peer-to-peer (P2P) architectures belong to this category. They usually offer computational services or storage facilities. Two of the most challenging issues in the development of such large-scale distributed systems are to come up with scalability and dynamicity. *Scalability* is achieved by designing protocols with performances growing sub-linearly with the number of nodes (or, processors, participants). *Dynamicity* refers to distributed systems in which topological changes can occur, *i.e.,* nodes may join or leave the system.

Self-stabilization [1] is a general technique to design distributed systems that can tolerate arbitrary transient faults. Self-stabilization is also well-known to be suitable for dynamic systems. This is particularly relevant whenever the distributed (self-stabilizing) protocol does not require any global parameters, like the number of nodes (n) or the diameter (D) of the network. With such a self-stabilizing protocol, it is not required to change global parameters in the program (n, D, etc) when nodes join or leave the system. Note that this property is also very desirable to achieve scalability.

The *end-to-end communication* problem consists in delivery in finite time across the network of a sequence of data items generated at a node called the sender, to a designated node called the receiver. This problem is generally split into the two following problems: (i) the *routing* problem, *i.e.,* the determination

[*] This work is supported by ANR SPADES grant.

of the path followed by the messages to reach their destinations; (ii) the *message forwarding* problem that consists in the management of network resources in order to forward messages. The former problem is strongly related to the problem of spanning tree construction. Numerous self-stabilizing solutions exist for this problem, *e.g.,* [2,3,4].

In this paper, we concentrate on the latter problem, *i.e.,* the message forwarding problem. More precisely, it consists in the design of a protocol managing the mechanism allowing the message to move from a node to another on the path from the sender A to the receiver B. To enable such a mechanism, each node on the path from A to B has a reserved memory space called buffers. With a finite number of buffers, the message forwarding problem consists in avoiding deadlocks and livelocks (even assuming correct routing table). Self-stabilizing solutions for the message forwarding problem are proposed in [5,6]. Our goal is to provide a snap-stabilizing solution for this problem. A *snap-stabilizing protocol* [7] guarantees that, starting from any configuration, it always behaves according to its specification, *i.e.,* it is a self-stabilizing algorithm which is optimal in terms of stabilization time since it stabilizes in 0 steps. Considering the message-forwarding problem, combined with a self-stabilizing routing protocol, snap-stabilization brings the desirable property that every message sent by the sender is delivered in finite time to the receiver. By contrast, any self-stabilizing (but not snap-stabilizing) solution for this problem ensures the same property, "eventually".

The problem of minimizing the number of required buffers on each node is a crucial issue for both dynamicity and scalability. The first snap-stabilizing solution for this problem can be found in [8]. Using n buffers per node, this solution is not suitable for large-scale system. The number of buffers is reduced to D in [9], which improves the scalability aspect. However, it works by reserving the entire sequence of buffers leading from the sender to the receiver. Furthermore, to tolerate dynamicity, each time a topology change occurs in the system, both of them would have to rebuild required data structures, maybe on the cost of loosing the snap-stabilisation property.

In this paper, we present a snap-stabilizing message forwarding protocol that uses a number of buffers per node being independent of any global parameter, that is 4 buffers per link. The protocol works on a linear chain of nodes, that is possibly an overlay on a large-scale and dynamic system *e.g.,* Peer-to-Peer systems, Grids... Provided that (i) the topology remains a linear chain and (ii) that nodes join and leave "neatly", the protocol tolerates topology changes. By "*neatly*", we mean that when a node leaves the system, it makes sure that the messages it has to send are transmitted, *i.e.,* all its buffers are free. We expect that this protocol will be the base to get similar results on more general topologies.

The paper is structured as follow: In Section 2, we define our model and some useful terms that are used afterwards. In Section 3, we first give an informal overview of our algorithm, followed by its formal description. In Section 4, we prove the correctness of our algorithm. Dynamicity is discussed in Section 5. We conclude the paper in Section 6.

2 Model and definitions

Network. We consider a network as an undirected connected graph $G = (V, E)$ where V is the set of nodes (processors) and E is the set of bidirectional communication links. A link (p, q) exists if and only if the two processors p and q are neighbours. Note that, every processor is able to distinguish all its links. To simplify the presentation we refer to the link (p, q) by the label q in the code of p. In our case we consider that the network is a chain of n processors.

Computational model. We consider in our work the classical local shared memory model introduced by Dijkstra [10] known as the state model. In this model communications between neighbours are modelled by direct reading of variables instead of exchange of messages. The program of every processor consists in a set of shared variables (henceforth referred to as variable) and a finite number of actions. Each processor can write in its own variables and read its own variables and those of its neighbours. Each action is constituted as follow:

$$< Label >::< Guard > \rightarrow < Statement >$$

The guard of an action is a boolean expression involving the variables of p and its neighbours. The statement is an action which updates one or more variables of p. Note that an action can be executed only if its guard is true. Each execution is decomposed into steps.

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. The local state refers to the state of a processor and the global state to the state of the system.

Let $y \in C$ and A an action of p $(p \in V)$. A is *enabled* for p in y if and only if the guard of A is satisfied by p in y. Processor p is enabled in y if and only if at least one action is enabled at p in y. Let P be a distributed protocol which is a collection of binary transition relations denoted by $\rightarrow$, on C. An execution of a protocol P is a maximal sequence of configurations $e = y_0 y_1 ... y_i y_{i+1} \ldots$ such that, $\forall\ i \geq 0$, $y_i \rightarrow y_{i+1}$ (called a step) if y_{i+1} exists, else y_i is a terminal configuration. *Maximality* means that the sequence is either finite (and no action of P is enabled in the terminal configuration) or infinite. All executions considered here are assumed to be maximal. ξ is the set of all executions of P. Each step consists on two sequential phases atomically executed: (i) Every processor evaluates its guard; (ii) One or more enabled processors execute its enabled actions. When the two phases are done, the next step begins. This execution model is known as the *distributed daemon* [11]. We assume that the daemon is *weakly fair*, meaning that if a processor p is continuously *enabled*, then p will be eventually chosen by the daemon to execute an action.

In this paper, we use a composition of protocols. We assume that the above statement (ii) is applicable to every protocol. In other words, each time an enabled processor p is selected by the daemon, p executes the enabled actions of every protocol.

Snap-Stabilization. Let Γ be a task, and S_Γ a specification of Γ . A protocol P is snap-stabilizing for S_Γ if and only if $\forall \Gamma \in \xi$, Γ satisfies S_Γ.

Message Forwarding Problem. The message forwarding problem is specified as follows:

Specification 1 **(SP)** *A protocol P satisfies SP if and only if the following two requirements are satisfied in every execution of P:*

1. *Any message can be generated in a finite time;*
2. *Any valid message is delivered to its destination once and only once in a finite time.*

Buffer Graph In order to conceive our snap stabilizing algorithm we will use a structure called Buffer Graph introduced in [12]. A Buffer Graph is defined as a directed graph where nodes are a subset of the buffers of the network and links are arcs connecting some pairs of buffers, indicating permitted message flow from one buffer to another one. Arcs are permitted only between buffers in the same node, or between buffers in distinct nodes which are connected by communication link.

Let us define our buffer graph (refer to Figure 1):

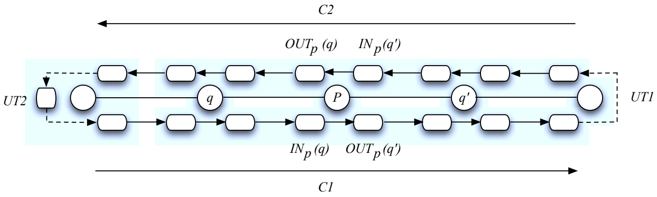


Fig. 1. Buffer Graph

Each processor p has four buffers, two for each link (p, q) such as $q \in N_p$ (except for the processors that are at the extremity of the chain that have only two buffers, since they have only one link). Each processor has two input buffers denoted by $IN_p(q)$, $IN_p(q')$ and two output buffers denoted by $OUT_p(q)$, $OUT_p(q')$ such as $q, q' \in N_p$ and $q \neq q'$ (one for each neighbour). The generation of a message is always done in the output buffer of the link (p, q) so that, according to the routing tables, q is the next processor for the message in order to reach the destination. Let us refer to $nb(m, b)$ as the next buffer of Message m stored in b, $b \in \{IN_p(q) \vee OUT_p(q)\}$, $q \in N_p$. We have the following properties:

1. $nb(m, IN_p(q)) = OUT_q(p)$
2. $nb(m, OUT_p(q) = IN_q(p)$

3 Message Forwarding

In this section, we first give the idea of our snap stabilizing message forwarding algorithm in the informal overview, then we give the formal description followed by the correctness proofs.

3.1 Overview of the algorithm

In this section, we provide an informal description of our snap stabilizing message forwarding algorithm that tolerates the corruption of the routing tables in the initial configuration.

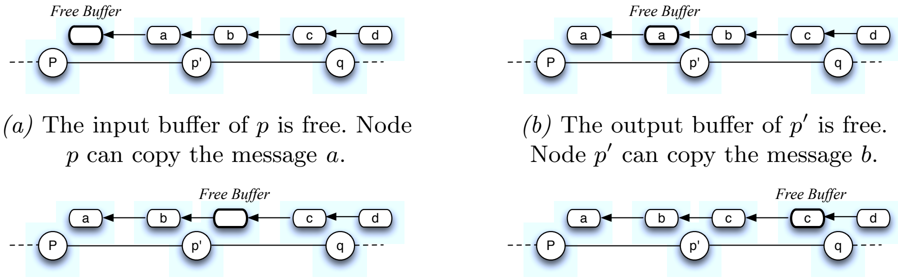
To ease the reading of the section, we assume that there is no message in the system whose the destination is not in the system. This restriction is not a problem as we will see in Section 5.

We assume that there is a self-stabilizing algorithm, $Rtables$, that calculates the routing tables and runs simultaneously to our algorithm. We assume that our algorithm has access to the routing tables via the function $Next_p(d)$ which returns the identity of the neighbour to which p must forward the message to reach the destination d. To reach our purpose we define a buffer graph on the chain which consists of two chains, one in each direction ($C1$ and $C2$ refer to Figure 1).

The overall idea of the algorithm is as follows: When a processor wants to generate a message, it consults the routing tables to determine the next neighbour by which the message will transit in order to reach the destination. Note that the generation is always done in the Output buffers. Once the message is on the chain, it follows the buffer chain (according to the direction of the buffer graph) and if the messages can progress enough in the system (move) then it will either meet its destination and hence it will be consumed in a finite time or it will reach the input buffer of one of the processors that are at the extremity of the chain. In the latter case, if the processor that is at the extremity of the chain is not the destination then, that means that the message was in the wrong direction. The idea is to change the direction of the message by copying it in the output buffer of the same processor (directly (UT1) or using the extra buffer (UT2), refer to Figure 1).

Note that if the routing tables are stabilized and if all the messages are in the right direction then all the messages can move on $C1$ or $C2$ only and no deadlock happens. However, in the opposite case (the routing tables are not stabilized or some messages are in the wrong direction), deadlocks may happen if no control is introduced. For instance, suppose that in the initial configuration all the buffers, uncluding the extra buffer of $UT2$, contain different messages such that no message can be consumed. It is clear that in this case no message can move and the system is deadlocked. Thus in order to solve this problem we have to delete at least one message. However, since we want a snap stabilizing solution we cannot delete a message that has been generated. Thus we have to introduce some control mechanisms in order to avoid this situation to appear dynamically (after the first configuration). In our case we decided to use the PIF algorithm that comprises two main phases: Broadcast (Flooding phase) and Feedback (acknowledgement phase) to control and avoid deadlock situations.

Before we explain how the PIF algorithm is used, let us focus on the message progression again. A buffer is said to be *free* if and only if it is empty (it contains no message) or contains the same message as the input buffer before it in the buffer graph buffer. For instance, if $IN_p(q) = OUT_q(p)$ then $OUT_q(p)$ is a free buffer. In the opposite case, a buffer is said to *busy.* The transmission of messages produces the filling and the cleaning of each buffer, *i.e.,* each buffer is alternatively free and busy. This mechanism clearly induces that *free slots* move into the buffer graph, a free slot corresponding to a free buffer at a given instant. The moving of free slots is shown in Figure 2[3]. Notice that the free slots move in the opposite direction of the message progression. This is the key feature on which the PIF control is based.



(a) The input buffer of p is free. Node p can copy the message a.

(b) The output buffer of p' is free. Node p' can copy the message b.

(c) The input buffer of p' is free. Node p' can copy the message c.

(d) The output buffer of q is free. Node q can copy the message d.

Fig. 2. An example showing the free slot moving.

When there is a message that is in the wrong direction in the Input buffer of the processor p_0, p_0 copies this message in its extra buffer releasing its Input buffer and it initiates a PIF wave at the same time. The aim of the PIF waves is to escort the free slot that is in the input buffer of p_0 in order to bring it in the Output buffer of p_0. Hence the message in the extra buffer can be copied in the output buffer to become in the right direction. Once the PIF wave is initiated no message can be generated on this free slot, at each time the Broadcast progresses on the chain the free slot moves as well following the PIF wave (the free slot moves by transmitting messages on $C1$ (refer to Figure 1). In the worst case, the

[3] Note that in the algorithm, the actions (b) and (c) are executed in the same step (refer to the guarded action $R3$).

free slot is the only one, hence by moving the output buffer of the other extremity of the chain p becomes free. Depending on the destination of the message that is in the input buffer of p, either this message is consumed or copied in the Output buffer of p. In both cases the input buffer of p contains a free slot.

In the same manner during the feedback phase, the free slot that is in the input buffer of the extremity p will progress at the same time as the feedback of the PIF wave. Note that this time the free slot moves on $C2$ (see Figure 1). Hence at the end of the PIF wave the output buffer that comes just after the extra buffer contains a free slot. Thus the message that is in the extra buffer can be copied in this buffer and deleted from the extra buffer. Note that since the aim of the PIF wave is to bring the free slot in the output buffer of p_0 then when the PIF wave meets a processor that has a free buffer on $C2$ the PIF wave stops escorting the previous free slot and starts the feedback phase with this second free slot (it escorts the new free slot on $C2$). Thus it is not necessary to reach the other extremity of the chain.

Now, in the case where there is a message in the extra buffer of p_0 such as no PIF wave is executed then we are sure that this message is an invalid message and can be deleted. In the same manner if there is a PIF wave that is executed such that at the end of the PIF wave the Output buffer of p_0 is not free then like in the previous case we are sure that the message that is in the extra buffer is invalid and thus can be deleted. Thus when all the buffers are full such as all the messages are different and cannot be consumed, then the extra buffer of p_0 will be released.

Note that in the description of our algorithm, we assumed the presence of a special processor p_0. This processor has an Extra buffer used to change the direction of messages that are in the input buffer of p_0 however their destination is different from p_0. In addition it has the ability to initiate a PIF wave. Note also that the other processors of the chain do not know where this special processor is. A symmetric solution can also be used (the two processors that are at the extremity of the chain execute the same algorithm) and hence both have an extra buffer and can initiate a PIF wave. The two PIF wave initiated at each extremity of the chain use different variable and are totally independent.

3.2 Formal description of the algorithm

We first define in this section the different data and variables that are used in our algorithm. Next, we present the PIF algorithm and give a formal description of the linear snap stabilizing message forwarding algorithm.

Character `'?'` in the predicates and the algorithms means *any value.*

- **Data**
  - n is a natural integer equal to the number of processors of the chain.
  - $I = \{0, ..., n-1\}$ is the set of processors' identities of the chain.
  - N_p is the set of identities of the neighbours of the processor p.
- **Message**

  - (m, d, c): m contains the message by itself, *i.e.,* the data carried from the sender to the recipient, $d \in I$ is the identity of the message recipient, and c is a color number given to the message to avoid duplicated deliveries.
- **Variable**
  - *In the forwarding algorithm*
    - $IN_p(q)$: The input buffer of p associated to the link (p, q).
    - $OUT_p(q)$: The output buffer of p associated to the link (p, q).
    - EXT_p: The Extra buffer of processor p which is at the extremity of the chain.
  - *In the PIF algorithm*
    - $S_p = (B \vee F \vee C, q)$ refers to the state of processor p, q is a pointer to a neighbour of p.

- **Input/Output**
  - $Request_p$: Boolean, allows the communication with the higher layer, it is set at true by the application and false by the forwarding protocol.
  - $PIF\text{-}Request_p$: Boolean, allows the communication between the PIF and the forwarding algorithm, it is set at true by the forwarding algorithm and false by the PIF algorithm.
  - The variables of the PIF algorithm are the input of the forwarding algorithm.
- **Procedure**
  - $Next_p(d)$: refers to the neighbour of p given by the routing table for the destination d.
  - $Deliver_p(m)$: delivers the message m to the higher layer of p.
  - $Choice(c)$: chooses a color for the message m which is different from the color of the message that are in the buffers connected to the one that will contain m.
- **Predicate**
  - $Consumption_p(q, m)$: $IN_p(q) = (m, d, c) \wedge d = p \wedge OUT_q(p) \neq (m, d, c)$
  - $leaf_p(q)$: $S_q = (B, ?) \wedge (\forall\, q' \in N_p/\{q\}, S_{q'} \neq (B, p) \wedge (consumption_p(q) \vee OUT_p(q') = \epsilon \vee OUT_p(q') = IN_{q'}(p)))$.
  - $NO\text{-}PIF_p$: $S_p = (C, NULL) \wedge \forall q \in N_p,\ S_q \neq (B, ?)$.
  - $init\text{-}PIF$: $S_p = (C, NULL) \wedge (\forall q \in N_p,\ S_q = (C, NULL)) \wedge PIF\text{-}Request_p = true$.
  - $Inter\text{-}trans_p(q)$: $IN_p(q) = (m, d, c) \wedge d \neq p \wedge OUT_q(p) \neq IN_p(q) \wedge (\exists q' \in N_p/\{q\},\ OUT_p(q') = \epsilon \vee OUT_p(q') = IN_{q'}(p))$.
  - $internal_p(q)$: $p \neq p_0 \wedge \neg\ leaf_p(q)$.
  - $Road\text{-}Change_p(m)$: $p = p_0 \wedge IN_p(q) = (m, d, c) \wedge d \neq p \wedge EXT_p = \epsilon \wedge OUT_q(p) \neq IN_p(q)$.
  - $\forall\ TAction \in C, B$, we define $TAction\text{-}initiator_p$ the predicate: $p = p_0 \wedge$ (the garde of TAction in p is enabled).
  - $\forall\ Tproc \in \{internal, leaf\}$ and $TAction \in \{B, F\}$, $T\text{-}Action\text{-}Tproc_p(q)$ is defined by the predicate: $Tproc_p(q)$ is true $\wedge$ TAction of p is enabled.
  - $PIF\text{-}Synchro_p(q)$: $(B_q\text{-}internal_p \vee F_q\text{-}leaf_p \vee F_q\text{-}internal_p) \wedge S_q = (B, ?)$.

– We define a fair pointer that chooses the actions that will be performed on the output buffer of a processor p. (Generation of a message or an internal transmission).

Algorithm 1 PIF

```
– For the initiator ($p_0$)
    • B-Action:: $init\text{-}PIF \rightarrow S_p := (B, -1),\ PIF\text{-}Request_p := false.$
    • C-Action:: $S_p = (B, -1) \wedge \forall q \in N_p,\ S_q = (F, ?) \rightarrow S_p := (C, NULL).$

– For the leaf processors: $leaf_p(q) = true \vee |N_p| = 1$
    • F-Action:: $S_p = (C, NULL) \rightarrow S_p := (F, q).$
    • C-Action:: $S_p = (F, ?) \wedge \forall q \in N_p,\ S_q = (F \vee C, ?) \rightarrow S_p := (C, NULL).$

– For the processors
    • B-Action:: $\exists! q \in N_p,\ S_q = (B, ?) \wedge S_p = (C, ?) \wedge \forall q' \in N_p/\{q\},\ S_{q'} = (C, ?) \rightarrow$
      $S_p := (B, q).$
    • F-Action:: $S_p = (B, q) \wedge S_q = (B, ?) \wedge \forall q' \in N_p/\{q\},\ S_{q'} = (F, ?) \rightarrow S_p := (F, q).$
    • C-Action:: $S_p = (F, ?) \wedge \forall q' \in N_p,\ S_{q'} = (F \vee C, ?) \rightarrow S_p := (C, NULL).$

– Correction (For any processor)
    • $S_p = (B, q) \wedge S_q = (F \vee C, ?) \rightarrow S_p := (C, NULL).$
    • $leaf_p(q) \wedge S_p = (B, q) \rightarrow S_p := (F, q).$
```

4 Proof of Correctness

In this section, we prove the correctness of our algorithm. We first show that starting from an arbitrary configuration, our protocol is deadlock free. Next, we show that no node can be starved of generating a new message. Next, we show the snap-stabilizing property of our solution by showing that, starting from any arbitrary configuration and even if the routing tables are not stabilized, every valid message is delivered to its destination once and only once in a finite time.

Let us first state the following lemma:

Lemma 1 *The PIF protocol (Algorithm 1) is snap-stabilizing.*

Proof. Note that the PIF algorithm introduced here is similar to the one proposed in [7] which is a snap stabilizing algorithm. The new thing is that we introduced the idea of dynamic leafs, processors that satisfy some properties and act like a physical leaf (they execute the F-action once they have a neighbor in a broadcast phase). Hence instead of reaching all the nodes of the chain, the PIF wave stops advancing when it meets a dynamic leaf. Note that once an internal processor p executes the B-Action, it cannot execute the F-Action unless is has a neighbor q such as $S_q = (F, p)$ (it cannot become a leaf) since to execute the F-action by any processor p, $S_p = (C, NULL)$ or for the internal processor that executes the B-Action $S_p = (B, q)$ $(q \in N_p,\ S_q = (B, p'))$. Thus no processor becomes a dynamic leaf of the PIF wave once it executed the B-Action of the

Algorithm 2 Message Forwarding

```
– Message generation (For every processor)
  R1:: $Request_p \wedge Next_p(d) = q \wedge [OUT_p(q) = \epsilon \vee OUT_p(q) = IN_q(p)] \wedge NO\text{-}PIF_p \rightarrow OUT_p(q) := (m, d, choice(c)),\ Request_p := false.$

– Message consumption (For every processor)
  R2:: $\exists q \in N_p,\ \exists m \in M;\ Consumption_p(q, m) \rightarrow deliver_p(m),\ IN_p(q) := OUT_q(p).$

– Internal transmission (For processors having 2 neighbors)
  R3:: $\exists q \in N_p,\ \exists m \in M,\ \exists d \in I;\ Inter\text{-}trans_p(q, m, d) \wedge (NO\text{-}PIF_p \vee PIF\text{-}Synchro_p(q)) \rightarrow OUT_p(q') := (m, d, choice(c)),\ IN_p(q) := OUT_q(p).$

– Message transmission from $q$ to $p$ (For processors having 2 neighbors)
  R4:: $IN_p(q) = \epsilon \wedge OUT_q(p) \neq \epsilon \wedge (NO\text{-}PIF_p \vee PIF\text{-}Synchro_p(q)) \rightarrow IN_p(q) := OUT_q(p).$

– Erasing a message after its transmission (For processors having 2 neighbors)
  R5:: $\exists q \in N_p,\ OUT_p(q) = IN_q(p) \wedge (\forall q' \in N_p \setminus \{q\},\ IN_p(q') = \epsilon) \wedge (NO\text{-}PIF_p \vee PIF\text{-}Synchro_p(q)) \rightarrow OUT_p(q) := \epsilon,\ IN_p(q') := OUT_{q'}(p).$

– Erasing a message after its transmission (For the extremities)
  R5':: $N_p = \{q\} \wedge OUT_p(q) = IN_q(p) \wedge IN_p(q) = \epsilon \wedge ((p = p_0) \Rightarrow (EXT_p = \epsilon)) \wedge (NO\text{-}PIF_p \vee PIF\text{-}Synchro_p(q)) \rightarrow OUT_p(q) := \epsilon,\ IN_p(q) := OUT_q(p).$

– Road change (For the extremities)
  • R6:: $Road\text{-}Change_p(m) \wedge [OUT_p(q) = \epsilon \vee OUT_p(q) = IN_q(p)] \rightarrow OUT_p(q) := (m, d, choice(c)),\ IN_p(q) := OUT_q(p).$
  • R7:: $Road\text{-}Change_p(m) \wedge OUT_p(q) \neq \epsilon \wedge PIF\text{-}Request_p = false \rightarrow PIF\text{-}Request_p := true.$
  • R8:: $Road\text{-}Change_p(m) \wedge OUT_p(q) \neq \epsilon \wedge PIF\text{-}Request_p \wedge B\text{-}initiator \rightarrow EXT_p := IN_p(q),\ IN_p(q) := OUT_q(p).$
  • R9:: $p = p_0 \wedge EXT_p \neq \epsilon \wedge [OUT_p(q) = \epsilon \vee OUT_p(q) = IN_q(p)] \wedge C\text{-}Initiator \rightarrow OUT_p(q) := EXT_p,\ EXT_p := \epsilon.$
  • R10:: $p = p_0 \wedge EXT_p \neq \epsilon \wedge OUT_p(q) \neq \epsilon \wedge OUT_p(q) \neq IN_q(p) \wedge C\text{-}Initiator \rightarrow EXT_p := \epsilon.$
  • R11:: $|N_p| = 1 \wedge p \neq 0 \wedge IN_p(q) = (m, d, c) \wedge d \neq p \wedge OUT_p(q) = \epsilon \wedge OUT_q(p) \neq IN_p(q) \rightarrow OUT_p(q) := (m, d, choice(c)),\ IN_p(q) := OUT_q(p).$

– Correction (For $p_0$)
  • R12:: $p = p_0 \wedge EXT_p \neq \epsilon \wedge S_p \neq (B, -1) \rightarrow EXT_p = \epsilon.$
  • R13:: $p = p_0 \wedge S_p = (B, ?) \wedge PIF\text{-}Request = true \rightarrow PIF\text{-}Request = false.$
  • R14:: $p = p_0 \wedge S_p = (C, ?) \wedge PIF\text{-}Request = true \wedge [(IN_p(q) = (m, d, c) \wedge d = p) \vee IN_p(q) = \epsilon] \rightarrow PIF\text{-}Request = false.$
```

same PIF wave. In another hand, note that the variable PIF-Request is a shared variable between the PIF algorithm and the forwarding algorithm, its role is to give the signal to the initiator to initiate the PIF wave. Hence we can deduct by analogy that the PIF algorithm proposed here is a snap stabilizing algorithm. □

We now show (Lemma 2) that the extra buffer located at p_0 cannot be infinitely continuously busy. As explained in Section 3, this solves the problem of deadlocks.

Lemma 2 *If the extra buffer of the processor p_0 (EXT_{p_0}) which is at the extremity of the chain contains a message then this buffer becomes free after a finite time.*

Proof. We know from Lemma 1 that each time p_0 launches a PIF wave, then this wave terminates. When this happens, there are two cases:

1. $S_p = (C, NULL)$. In this case $R12$ is enabled on p. Since the daemon is weakly fair we are sure that $R12$ will be executed in a finite time. Thus EXT_p will be free in a finite time too.
2. $S_p = (B, ?)$. In this case, a PIF wave is executed on the chain. Note that p is the initiator (it is eventually considered as the initiator). According to Lemma 1 the PIF is a Snap stabilizing algorithm. Hence, there will be a time when $S_q = (F, p)$, $q \in N_p$. Two sub cases are possible:
   - Either $OUT_p(q) = \epsilon$ or $OUT_p(q) = IN_q(p)$. In this case $R9$ is enabled on p. Since the daemon is weakly fair, this rule will be executed in a finite time. Hence the message that is in the extra buffer will be copied in $OUT_p(q)$ and deleted from EXT_p (see Rule $R9$). Hence $EXT_p = \epsilon$.
   - $OUT_p(q) \neq \epsilon$ and $OUT_p(q) \neq IN_q(p)$. Since $R10$ is enabled on p and the daemon is weakly fair, $R10$ will be executed in a finite time. Thus, the message that is in the extra buffer is deleted.

From the cases above, we deduct that in the case where the extra buffer of p contains a message, then this message will be either deleted or copied in $OUT_p(q)$. Hence EXT_p will be free in a finite time and the lemma holds. □

We deduce from Lemma 2 that if the routing tables are not stabilized and if there is a message locking the Input buffer of p_0, then this message is eventually copied in the extra buffer. Since the latter is infinitely often empty (Lemma 2 again).

From now on, we suppose that no generated message is deleted (we prove this property latter).

Lemma 3 *All the messages progress in the system even if the routing tables are not stabilized.*

Proof. Note that if $\exists q \in N_p$, $IN_p(q)$ is free then if there is a message in $OUT_q(p)$, then this message is automatically copied in $IN_p(q)$. Hence it is sufficient to prove that the input buffer are free in a finite time. Thus Let's prove that $\forall p \in I$, when there is a message in $IN_p(q)$, this message is deleted from $IN_p(q)$ in a finite time ($q \in N_p$).

Note that the input-buffers are all at an even distance from the input buffer of the processor p_0. Let define δ as the distance between the input buffer of the processor p and the input buffer of processor p_0 (In the direction of the buffer graph taken in account $UT1$). The lemma is proved by induction on δ. We define for this purpose the following predicate P_δ:

If there is a message mh in $IN_p(q)$ such as $IN_p(q)$ is at distance δ from the input buffer of p_0 then one of these two cases happens:

- m is consumed and hence delivered to its destination.
- m is deleted from the input buffer and copied either in EXT_p or $OUT_p(q')$ in a finite time.

Initialization. Let's prove the result for P_0. Suppose that there is a message m in $IN_p(q)$ such as $p = p_0$ and $q \in N_p$. Two cases are possible according to the destination (d) of m:

- $d = p$. In this case, since the daemon is weakly fair and since $R2$ keep being enabled on p then $R2$ will be executed on p in a finite time and the message m in $IN_p(q)$ is consumed. Thus P_0 is true.
- $d \neq p$. Since the daemon is weakly fair we are sure that p will be activated. Two cases are possible:
  - $OUT_p(q) = \epsilon$ or $OUT_p(q) = IN_q(p)$. In this case $R6$ is enabled on p. Hence the message m is copied in $OUT_p(q)$ and deleted from $IN_p(q)$ since a new value overwrite it (see $R6$). Thus P_0 is true.
  - $OUT_p(q) \neq \epsilon$ and $OUT_p(q) \neq IN_q(p)$. According to Lemma 2, EXT_p will be free in a finite time. In another hand since the PIF is a snap stabilizing algorithm (refer to Lemma 1, we are sure that the B-Action of the initiator will be enabled on p in a finite time). Hence the message m will be copied in this case in EXT_p and deleted from $IN_p(q)$ (Note that in the case where $PIF\text{-}Reaquest = false$ then it is set at true (see rule $R7$)). Thus P_0 is true.

In both cases either the message is consumed or it is removed from $IN_p(q)$. Thus P_0 is true.

Induction. let $\delta \geq 1$. We assume that $P_{2\delta}$ is true and we prove that $P_{2\delta+2}$ is true as well (Recall that the input buffers are at an even distance from the input buffer of p_0). Let $IN_q(p)$ be the buffer at distance 2δ from the input buffer of p_0 and $IN_p(q')$ the one that is at distance $2\delta + 2$ and contains the message m'.
In the case where the destination of m' is p then it will be consumed in a finite time (the daemon is weakly fair and $R2$ keep being enabled on p. Thus p will execute $R2$ in a finite time). Hence $P_{2\delta+2}$ is true. In the other case (the destination of m' is different from p), since $P_{2\delta}$ is true then if there is a message m in $IN_q(p)$ then we are sure that this message will be either consumed or copied in $OUT_q(q'')$. Thus $IN_q(p) = OUT_p(q)$ ($OUT_p(q)$ is free). Two cases are possible according to the rule that is executed on $OUT_p(q)$ (depending on the value of the pointer on $OUT_p(q)$) :

1. p executes $R3$. In this case the message m' is copied in $OUT_p(q)$ and deleted from $IN_p(q')$ since a new value overwrite it (refer to Rule $R3$). Hence $P_{2\delta+2}$ is true.
2. p executes $R2$ (it generates a message). Hence $OUT_p(q) = m''$ (m'' is the message generated by p). However, since $P_{2\delta}$ is true, then the message in $IN_q(p)$ will be deleted from the buffer (q performs either an internal transmission or consume the message). Hence $IN_q(p) = OUT_p(q)$ in a finite time. Nevertheless, since p generated a message in the previous step, we are sure that it will execute $R3$ (since the pointer on the output buffer $OUT_p(q)$ is fair). Thus we retrieve case 1.

From the proof above, we deduct that all the messages in the chain progress in the system and no deadlock happens even if the routing tables are corrupted. □

Let us call a *valid PIF* wave every PIF wave that is initiated by the processor p_0 at the same time as executing $R8$.

Lemma 4 *For every valid PIF wave, when the C-Action is executed in the initiator either $OUT_p(q) = IN_q(p)$ or $OUT_p(q) = \epsilon$.*

Proof. The idea of the proof is as follows:

- We prove first that during the broadcast phase there is a synchrony between the PIF and the forwarding algorithm. Note that when the message that was in the input buffer of the initiator is copied in the extra buffer, the input buffer becomes free. The free slot in that buffer progresses in the chain at the same time as the broadcast of the PIF wave.
- Once the PIF reaches a leaf, a new buffer becomes free in $C2$ (refer to Figure 1).
- As in the broadcast phase, there is a synchrony between the PIF and the forwarding algorithm during the feedback phase. (The feedback will escort the new free slot on $C2$ to the output buffer of p_0.)

Let's prove that during the broadcast phase there is a synchrony between the PIF and the forwarding algorithms. Let's define for this purpose λ the distance between the processor p and the processor p_0. We're going to prove the result by induction on λ. let's define fot this purpose the predicate P_λ as follow:

When the PIF wave is initiated and for each extra processor p $(\neg leaf \wedge p \neq p_0)$ that executes the B-action we have: $\exists!\ q \in N_p$, $S_q = (B, ?)$, $S_p = (B, q) \wedge IN_p(q') = OUT_{q'}(p) \wedge S_{q'} = (C, NULL)$ $(q' \in N_p/\{q\})$.

Initialization. Let's prove that P_1 is true. Since the PIF wave is valid, when the PIF wave was initiated, $R8$ was executed at the same time. Hence, the message that was in $IN_{p_0}(q)$ was copied in EXT_{p_0}, $IN_{p_0}(q) = OUT_q(p_0)$, $S_{p_0} = (B, -1)$ and $S_q = (C, NULL)$. Since q is not a leaf only $R3$ or $R5$ and possibly $R2$ are enabled on q. Note that after the execution of one of these rules $IN_q(q') = OUT_{q'}(q)$ $(q' \in N_q/\{p_0\})$. In another hand $R3$ and $R5$ are not enabled only if the *B-action* of the internal processor is enabled as well. Thus when the B-Action is executed (we are sure that this will happen since the PIF algorithm is snap stabilizing and the daemon is weakly fair) either $R3$ or $R5$ (possibly $R2$) are executed at the same time (Recall that when two actions from the PIF and the forwarding algorithm are enabled on the same processor at the same time they are both executed). Hence $IN_q(q') = OUT_{q'}(q)$, $S_q = (B, p_0)$ and $S_{q'} = (C, NULL)$. Thus P_1 is true. Note that if $R2$ is executed alone before the B-Action then either $R3$ or $R5$ are still enabled on q. Hence when the B-Action is executed one of these two actions are executed as well.

Induction. Let $\lambda \geq 2$. We assume that P_λ is true and we prove that $P_{\lambda+1}$ is true as well. Let q and p be the processors that are at distance λ and $\lambda + 1$ respectively from the processor p_0. Since P_λ is true, when the B-Action of the internal processor is executed on q, $S_q = (B, q'')$, $IN_q(p) = OUT_p(q)$ and $S_p = (C, NULL)$. In the same manner as the proof of P_1, $R3$ or $R5$ and possibly $R2$ is enabled on p. Note that $R3$ or $R5$ keep being enabled unless they are executed (Note that $R1$ cannot be executed since there is a PIF wave that is executed in the neighbourhood. Thus no message is generated). In another hand $R3$ and $R5$ cannot be executed unless the B-Action is enabled as well. Hence when the B-Action is executed either $R3$ or $R5$ or $R2$ is executed at the same time. Hence $IN_p(q') = OUT_{q'}(p)$, $S_p = (B, q)$ and $S_{q'} = (C, NULL)$. Thus $P_{\lambda+1}$ is true.

We can deduct that for the last processor p that is an internal processor ($p \neq p_0$ and $\neg$ leaf), $S_q = (B, q'')$ and $IN_p(q') = OUT_{q'}(p)$, $S_p = (B, q)$ and $S_{q'} = (C, NULL)$. Since p is the last processor which is an internal processor then q' is a leaf. Two cases are possible:

- The leaf is the processor q' that is at the extremity of the chain such as $q' \neq p_0$. Either $R2$ or $R11$ are enabled on q'. Note that the F-Action is enabled as well since q' is a leaf and $S_p = (B, ?)$. When one of these two rules is executed with the F-Action, $S_{q'} = (F, p)$ and $IN_{q'}(p) = OUT_p(q')$.
- The leaf q' is not the processor that is at the border of the chain. In this case either $R2$ or $R3$ or $R4$ are enabled. Recall that the F-action is enabled as well. Then once the F-Action of the internal processor and one of these rules are executed, $S_{q'} = (F, p)$ and $IN_{q'}(p) = OUT_p(q')$.

Note that in both cases, once the leaf q' executed the F-Action we have the following property: $S_{q'} = (F, p)$ and $IN_{q'}(p) = OUT_p(q')$. Now in the same manner that we proved the synchrony between the PIF and the forwarding algorithm during the broadcast phase. We prove the synchrony between these two algorithms during the feedback phase. The proof is by induction on λ using the following predicate: For every internal processor p that executes the F-Action, $S_p = F, q$, $S_{q'} = (F, p)$, $S_{q'} = (B, ?)$ and $IN_p(q) = OUT_q(p)$ ($q, q' \in N_p, q \neq q'$). Then when the last internal processor executes that F-action (note that the last processor is the one that is neighbour to the initiator) these properties are verified. Hence the Output buffer of the initiator is becomes free and the lemma holds.

□

In the remainder, we say that a message is in a *suitable* buffer if the buffer is on the right direction to its destination. A message is said to be deleted if it is removed from the system without being delivered.

Let m be a message. According to Lemma 3, m progresses in the system (no deadlock happens and no message stays in the same buffer indefinitely). So, if m is in a buffer that is not suitable for it, then m progresses in the system according to the buffer graph. Thus, it eventually reaches an extremity, which changes its direction. Now, m is ensured to reach its destination, leading to the following lemma:

Lemma 5 *For every message that is not in a suitable buffer, it will undergo exactly a single route change.*

Proof. Let m be a message. According to Lemma 3, m progresses in the system (no deadlock happens and no message stays in the same buffer indefinitely). So, if m is not in a buffer that is not suitable for it, then m progresses in the system according to the buffer graph. Two cases are possible:

- m will be in a finite time in the Input buffer of the processor p_0. Since the message is in a buffer that is not suitable for it, p_0 is not the destination of m. However we are sure that EXT_p will be free in a finite time (refer to Lemma 2) and that the B-Action will be enabled on p_0 in a finite time too (The PIF is a snap stabilizing algorithm). Hence the message in $IN_{p_0}(q)$ will be copied in EXT_{p_0} and a PIF wave is initiated at the same time. In another hand, it has been shown in Lemma 4 that in the case of a valid PIF wave (Note that this is our case) when the PIF ends ($S_q = (F, p_0)$) and C-Initiator is enabled on p_0 (we are sure that this will happen since the PIF is snap stabilizing) and $OUT_p(q)$ becomes free. Hence $R9$ is enabled on p_0 and the message that is in the extra buffer can be put in the output buffer of p_0 and deleted from the extra buffer. Note that since the network is a chain and p_0 is at the extremity of this chain, we are sure that the message will meet its destination since it can visit all the processors. Hence no other changes route are done.
- The message will reach the input buffer of the processor p' that is at the other extremity of the chain ($p' \neq 0$). Since the messages progress in the system (see Lemma 3) $OUT_{p'}(q')$ will be free in a finite time. Hence when a message that is not intended to q is in $IN_q(q')$ and since the daemon is weakly fair, we are sure that q will execute $R11$ in a finite time. Thus the message will be copied in $OUT_q(q')$ and deleted from $IN_q(q')$. Now since q is at the extremity of the chain, the message will meet its destination hence no other route change is performed and the lemma holds.

□

Once the routing tables are stabilized, every new message is generated in a suitable buffer. So, it is clear from Lemma 5 that the number of messages that are not in a suitable buffer strictly decreases. The next lemma follows:

Lemma 6 *When the routing tables are stabilized and after a finite time, all the messages are in buffers that are suitable for them.*

Proof. When the routing table are stabilized some of the messages still are in buffers that are not suitable for then. However, since the routing tables are stabilized, every message is generated in a suitable buffer, hence the number of messages that are in buffers that are not suitable for them does not increase. In another hand, According to Lemma 5, every message that is in the wrong direction will undergo exactly one route change and hence all the wrong messages that are in the system will be in the right direction in a finite time and the lemma holds.

□

From there, it is important to show that any processor can generate a message in a finite time. From Lemma 6, all the messages are in suitable buffers in a finite time. Since the PIF waves are used for route changes only, then no PIF wave will be initiated. That what we show in the two following lemmas:

Lemma 7 *In the case where PIF-Request=true, it will be set at false in a finite time.*

Proof. Note that in the case where PIF-request is true and the B-Action of the initiator is enabled on p_0, PIF-Request will be set at false when the B-Action is executed (see B-Action of the initiator). Otherwise two cases are possible according to the state of the initiator:

- $S_{p_0} = (B, ?)$. In this case PIF-Request is set at false by the forwarding algorithm by executing $R13$ ($R13$ is enabled on p_0 and the daemon is weakly fair).
- $S_{p_0} = (C, NULL)$. If IN_{p_0} contains a messages and the destination of he message is not p_0 then PIF-Request will be set at false by the PIF algorithm once the PIF-wave is initiated. However in the case where the input buffer of p_0 is empty or contains a message to p_0 then $R14$ is enabled and since the daemon is weakly fair $R14$ will be executed on p_0 and hence PIF-Request is set at false.

From the cases above we can deduct that if PIF-Request is true then it will be set at false in a finite time and the lemma holds. □

Lemma 8 *When the routing tables are stabilized and all the messages are in suitable buffer, no PIF wave is initiated.*

Proof. According to Lemma 7. PIF-Request will be set at false in a finite time. Note that the only rule that set PIF-Request at true is $R7$. However $R7$ is never enabled since all the messages on the chain are in suitable buffer and since the routing tables are correct (all messages are generated in suitable buffer). Thus the lemma holds. □

From this point, the fair pointer mechanism cannot be disrupted by the PIF waves anymore. So, the fairness of message generation guarantees the following lemma:

Lemma 9 *Any message can be generated in a finite time under a weakly fair daemon.*

Proof. According to Lemma 8, when the routing tables are stabilized and when all the messages are containing in buffers that are suitable for them no PIF and no Road-change are executed. In another hand since the routing tables are stabilized and since the buffer graph of the chain consists on two disjoint chains (it is a DAG) then no deadlock happens and all the messages progress in the system. Now suppose that the processor p wants to generate a message. Recall that the generation of a message m for the destination d is always done in the output buffer of the processor p connected to the link (p, q) such as $Next_p(d) = q$. Two cases are possible

1. $OUT_p(q) = \epsilon$. In this case, the processor executes either $R1$ or $R3$ in a finite time. the result of this execution depends on the value of the pointer. Two cases are possible:
   - the pointer refers to $R1$. Then p executes $R1$ in a finite time and we obtain the result.
   - the pointer refers to $R3$. Then p executes $R3$ in a finite time. Hence $OUT_p(q) \neq \epsilon$ and we retrieve case 2. Note that the fairness of the pointer guarantees us that this case cannot appear infinitely.
2. $OUT_p(q) \neq \epsilon$. Since all the messages move gradually in the buffer graph we are sure that $OUT_p(q)$ will be free in a finite time and we retrieve case 1.

We can deduct that every processor can generate a message in a finite time. □

Due to the color management (Function $Choice(c)$), the next lemma follows:

Lemma 10 *The forwarding protocol never duplicates a valid message even if A runs simultaneously.*

Proof. Three cases are possible:

- m is in $IN_p(q)$. According to the rules that are enabled on p, three cases are possible

  - the message is consumed ($R2$ is executed) hence the message m is deleted from $IN_p(q)$ since a new value overwrites since $IN_p(q) = OUTq(p)$ (Note that this happen only when $OUT_q(p) \neq IN_p(q)$).
  - $R8$ is executed . The message is copied in EXT_p (for the processor p_0) and deleted from $IN_p(q)$ since a new value overwrites ($IN_p(q) = OUTq(p)$) in a sequential manner.
  - $R4$ is executed. The message is put in this case in $OUT_p(q')$ and deleted from $IN_p(q)$ in a sequential manner hence only one copy is kept ($q' \in N_p/\{q\}$). Note that these two rules are not enabled only if $OUT_q(p)$ does not contain the same message.
- m is in $OUT_p(q)$. In this case the message m is copied in the input buffer of the processor q ($IN_q(p)$). Hence two copies are in the system. However the message in $IN_q(p)$ is not consumed and not transmitted unless the copy in $OUT_p(q)$ is deleted (see the rules $R2$ and $R4$).

- m is in EXT_p. In this case the message is either deleted or put in $OUT_p(q)$. Since this operation is a local operation (the copy is done between two buffer of the same processor) then the message is copied in the new buffer and deleted from the previous one in a sequential manner.

From the cases above we can deduct that no message is duplicated in the system. Hence m is delivered at most once to its destination. □

From Lemma 9, any message can be generated in a finite time. From the PIF mechanism and its synchronization with the forwarding protocol the only message that can be deleted is the message that was in the extra buffer at the initial configuration. Thus:

Lemma 11 *Every valid message (that is generated by a processor) is never deleted unless it is delivered to its destination even if Rtables runs simultaneously.*

Proof. The proof is by contradiction, suppose that there is a message m that is deleted without being delivered to its destination.

By construction of $R3$, this cannot be a result of an internal forwarding since the message m is first of all copied in the Output-buffer $OUT_p(q)$ and then erased from the Input-buffer $IN_p(q')$ since a new value overwrites it. Note that these two rules are enabled only if $OUT_p(q)$=$IN_q(p)$ or $OUT_p(q) = \epsilon$. Hence when the message m is copied in the $OUT_p(q)$ no message is deleted (one copy remains in $IN_q(p)$ in the case where $IN_q(p) = OUT_p(q)$).

By the construction of Rule $R4$, the message is only copied in the Input-Buffer and not deleted from the Output-buffer at the neighbour processor simultaneously (the only rules that delete a message from the Output-buffer are $R5$ and $R3$ and the guards of these rules are not verified when $R4$ is enabled).

If $R5$ is enabled in processes p, that means that $OUT_q(p) = IN_p(q)$ and $IN_p(q') = \epsilon, q' \in N_p/\{q\}$. When one of these two rules are enabled, $OUT_p(q) = \epsilon$. However according to the color management (Function Choice(c)), we are sure that a copy of the message that was in $OUT_p(q)$ is in $IN_q(p)$.

By the construction of the rules $R6$ and $R11$, this cannot be the result of the execution of these two rules because the message that is in $IN_p(q)$ such as p_0 and p is not the destination, is copied in the Output buffer and deleted from the Input buffer sequentially and then p copies the message that is in $OUT_q(p)$ in $IN_p(q)$, so no message is deleted.

Concerning $R12$, EXT_p such as p_0 contains the message m and $S_p = (C, ?)$, which means that no PIF is executed. However, for p_0, a message in $IN_p(q)$ is copied in EXT_p (in the case where p is not the destination) only if $R8$ is enabled, however, when $R8$ is enabled *B-intiator* is enabled as well. Since in this case the two rules are executed at the same time, hence $S_p = (B, ?)$. Now, for the processor p, Since the PIF is a valid PIF, when the *C-Action* of p is enabled at the same time as the rules $R10$ or $R9$. If $R10$ is executed then $EXT_p = \epsilon$ and $S_p = (C, NULL)$ (since $OUT_p(q) = \epsilon$ or $OUT_p(q) = IN_q(p)$), which is a contradiction, since in our case $EXT_p \neq \epsilon$. If $R9$ is executed, then the message in the extra buffer of p (EXT_p) is copied in $OUT_p(q)$, EXT_p becomes free and $S_p = (C, ?)$, which is a contradiction with our case. Hence we are sure that the message that is in the extra buffer of p is a message that was not generated by a processor. Hence when $R12$ is executed, this message is deleted (no valid message is deleted).

By the construction of the two rules $R8$ and $R9$, No valid message is deleted by the execution of the two rules, since the message is copied in the extra buffer ($R8$) or in the Output buffer ($R9$) and then it is deleted from the Input buffer ($R8$) or the extra buffer ($R9$).

Concerning the rule $R10$, according to Lemma 4, when the message that is in EXT_p is valid, when the *C-Action* of the initiator is enabled either $OUT_p(q) = \epsilon$ or $OUT_p(q) = IN_q(p)$. However no such buffers exist. Hence the message in the extra buffer of p is not a valid message (it is not generated by a processor). Hence it can be deleted.

We can deduct from all the cases above that no message that is generated by a processor is deleted, hence the lemma holds.

□

Theorem 1. *The proposed algorithm (Algorithms 1 and 2) is a snap-stabilizing message forwarding algorithm (satisfying SP) under a weakly fair daemon.*

Proof. From Lemma 9, any message can be generated in a finite time. From Lemma 11, every valid message is never deleted unless it is delivered to its destination even if *Rtables* runs simultaneously. From Lemma 10, no valid message is duplicated. Hence, the theorem holds. □

Remarque

For any processor p, Forwarding protocol delivers at most $4n-3$ invalid messages.

Proof. Assume that in the initial configuration all the buffers contain a message, since these messages were not generated by the processors of the system, they are invalid messages. Suppose that the destination of the message m in $IN_p(q)$ is the processor p such as $q = 0$ and $S_q = (B, -1)$. Suppose that the daemon activates p which executes $R2$ and the F-action (it is a leaf) . Hence the message m is consumed and $IN_p(q) = OUT_q(p)$. Hence $OUT_q(p)$ becomes free and the *C-action* of the initiator is enabled, q will copy then the message from EXT_q in $OUT_q(p)$ and will execute the C-Action. In another hand, since there is no way to know if the messages are valid or not, they all be treated as if they are valid. Since the forwarding algorithm is snap stabilizing, all the messages that were in the buffer of the chain at the beginning are delivered. Since there is 4n-3 buffers in the system, then 4n-3 invalid messages can be delivered and the lemma holds.

□

5 Dynamicity

In dynamic environments, processors may leave or join the network at any time. To keep our solution snap stabilizing we assume that there are no crashes and if a processor wants to leave the network (disconnect), it releases its buffers (it sends all the messages it has to send and wait for their reception by its neighbours) and accepts no more message before leaving.

In this discussion we assume that the rebuilt network is still a chain. It is fundamental to see that in dynamic systems the problem of keeping messages for ghost destinations with the hope they will join the network again and the lack of congestion are contradictory. If there is no bound on the number of leavings and joins this problem do not admit any solution. The only way is to redefine the problem in the context of dynamicity. For example we can modify the second point of the specification *(SP)* as follows: A valid message m generated by the processor p to the destination q is delivered to q in a finite time if m, p and q are continuously in the same connected component during the forwarding of the message m. Even if that could appear very strong, this kind of hypothesis is often implied in practice. However we can remark that this new specification is equivalent to SP in static environments. Our algorithm can easily be adapted in order to be snap stabilizing for this new specification in dynamic chains.

Thus we can now delete some messages as follows: we suppose that every message has an additional boolean field initially set to false. When a message reaches an extremity which is not its destination we have two cases: *(i)* The value of the boolean is false, then the processor sets it to true and sends it in the opposite direction. *(ii)* The value of the boolean is true, then the processor deletes it (in this case, if the message is valid, it crossed all the processors of the chain without meeting its destination).

Finally, in order to avoid starvation of some processors, the speed of joins and leavings of the processors has to be slow enough to avoid a sequence of PIF waves that could prevent some processors to generate some messages.

6 Conclusion

In this paper, we presented the first snap-stabilizing message forwarding protocol that uses a number of buffers per node being independent of any global parameter. Our protocol works on a linear chain and uses only 4 buffers per link. It tolerates topology changes (provided that the topology remains a linear chain). This is a preliminary version to get the same result on more general topologies. In particular, by combining a snap-stabilizing message forwarding protocol with any self-stabilizing overlay protocols (*e.g.,* [13] for DHT or [14,15,16] for *tries*), we would get a solution ensuring users to get right answers by querying the overlay architecture.

References

1. Dolev, S.: Self-stabilization. MIT Press (2000)

2. Huang, S.T., Chen, N.S.: A self-stabilizing algorithm for constructing breadth-first trees. Inf. Process. Lett. **41**(2) (1992) 109–117
3. Kosowski, A., Kuszner, L.: A self-stabilizing algorithm for finding a spanning tree in a polynomial number of moves. In: PPAM. (2005) 75–82
4. Johnen, C., Tixeuil, S.: Route preserving stabilization. In: Self-Stabilizing Systems. (2003) 184–198
5. Awerbuch, B., Patt-Shamir, B., Varghese, G.: Self-stabilizing end-to-end communication. Journal of High Speed Networks **5**(4) (1996) 365–381
6. Kushilevitz, E., Ostrovsky, R., Rosén, A.: Log-space polynomial end-to-end communication. In: STOC '95: Proceedings of the twenty-seventh annual ACM symposium on Theory of computing, ACM (1995) 559–568
7. Bui, A., A, D., Petit, F., Villain, V.: Snap-stabilization and PIF in tree networks. Distributed Computing **20**(1) (2007) 3–19
8. Cournier, A., Dubois, S., Villain, V.: A snap-stabilizing point-to-point communication protocol in message-switched networks. In: 23rd IEEE International Symposium on Parallel and Distributed Processing (IPDPS 2009). (2009) 1–11
9. Cournier, A., Dubois, S., Villain, V.: How to improve snap-stabilizing point-to-point communication space complexity? In: Stabilization, Safety, and Security of Distributed Systems, 11th International Symposium (SSS 2009). Volume 5873 of Lecture Notes in Computer Science. (2009) 195–208
10. Edsger, W., Dijkstra.: Self-stabilizing systems in spite of distributed control. Commum. ACM **17**(11) (1974) 643–644
11. Burns, J., Gouda, M., Miller, R.: On relaxing interleaving assumptions. In: Proceedings of the MCC Workshop on Self-Stabilizing Systems, MCC Technical Report No. STP-379-89. (1989)
12. Merlin, P.M., Schweitzer, P.J.: Deadlock avoidance in store-and-forward networks. In: Jerusalem Conference on Information Technology. (1978) 577–581
13. Bertier, M., Bonnet, F., Kermarrec, A.M., Leroy, V., Peri, S., Raynal, M.: D2HT: the best of both worlds, Integrating RPS and DHT. In: European Dependable Computing Conference. (2010)
14. Aspnes, J., Shah, G.: Skip Graphs. In: Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms. (January 2003) 384–393
15. Caron, E., Desprez, F., Petit, F., Tedeschi, C.: Snap-stabilizing Prefix Tree for Peer-to-peer Systems. In: SSS 2007, Springer Verlag Berlin Heidelberg (2007) 82–96
16. Caron, E., Datta, A., Petit, F., Tedeschi, C.: Self-stabilization in tree-structured p2p service discovery systems. In: 27th International Symposium on Reliable Distributed Systems (SRDS 2008), IEEE (2008) 207–216