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, etc. 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.

Keywords: Dynamic Networks, 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 dynamic of the network. *Scalability* is achieved by designing protocols with performances growing sub-linearly with the number of nodes (or, processors, participants). *Dynamic Network* 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

* This work is supported by ANR SPADES grant.

into the two following problems: (*i*) the *routing* problem, *i.e.,* the determination 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.,* [2–4].

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 tables). 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 dynamic 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 the network dynamic, 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-stabilization 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, etc. 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. Network dynamic 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:

$$< \textit{Label} >::< \textit{Guard} > \rightarrow < \textit{Statement} >$$

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.

We denote by C the set of all configurations 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 at least one of their actions that are enabled in each algorithm. 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. Messages transit in the network in the Store and Forward model *i.e.,* they are stored temporally in each processor before being transmitted. Once the message is transmitted it can be deleted from the previous processor. Note that in order to store messages, each processor use a space memory called buffer. We assume in our case that each buffer can store a whole message and each message needs only one buffer to be stored.

It is clear that each processor uses a finite number of buffers for the message forwarding. Thus the aim is to bound these resources avoiding deadlocks (a configuration is which in every execution some messages can not be transmitted) and starvation (a configuration from which, in every execution, some processors are no longer able to generate messages). Thus some control mechanisms must be introduced in order to avoid these kind of situations.

The message forwarding problem is formally 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. *every message can be generated in finite time;*
2. *every valid message (generated by a processor) is delivered to its destination once and only once in 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):
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 $q, q' \in N_p$ (N_p is the set of identities of the neighbours of 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:

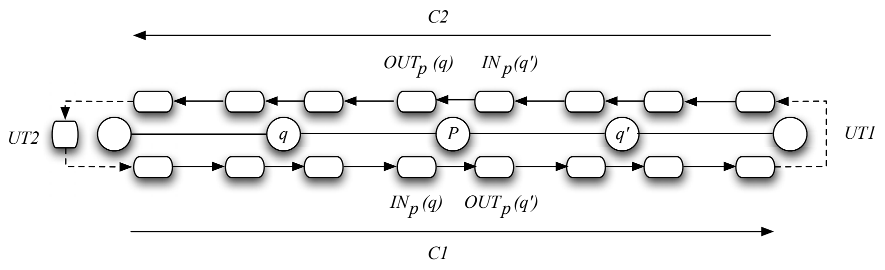


Fig. 1. Buffer Graph

1. $nb(m, IN_p(q)) = OUT_q(p)$, *i.e.,* the next buffer of the message m that is in the input buffer of p on the link (p, q) is the output buffer of q connected to p,
2. $nb(m, OUT_p(q)) = IN_q(p)$, *i.e.,* the next buffer of the message m that is in the output buffer of p on the link (p, q) is the input buffer of q connected to 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 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).

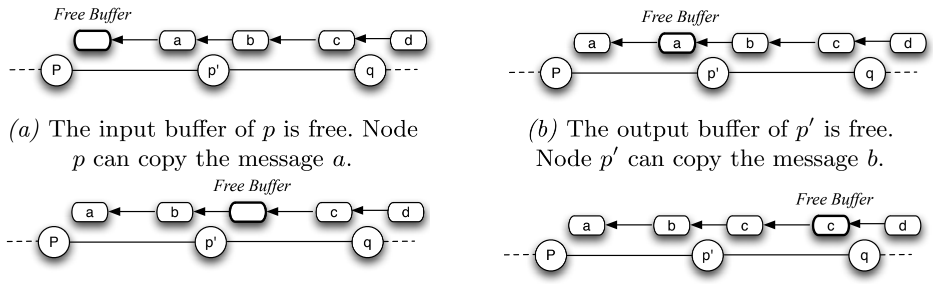
To avoid duplicated deliveries, each message is alternatively labelled by a color. If the messages can progress enough in the system (move) then it will either meet its destination and hence it will be consumed in finite time or it will reach the input buffer of one of the processors that are at the 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). Let p_0 be the processor that is at the extremity of the chain that has an internal buffer that we call Extra buffer.

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 movement of free slots is shown in Figure 2[1]. 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.

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 and becomes 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

[1] Note that in the algorithm, the actions (b) and (c) are executed in the same step.



(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 movement

the worst case, the 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, and $d \in I$ is the identity of the message recipient. In addition to m and d each message carries an extra field, c, which is a color number in $\{0, 1\}$ alternatively given to the messages 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 (B, F, and C refer to Broadcast, Feedback, and Clean, respectively), q is a pointer to a neighbour of p.

- **Input/Output**
  - $Request_p$: Boolean, allows the communication with the higher layer, it is set to 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 to true by the forwarding algorithm and false by the PIF algorithm.
  - The variables of the PIF algorithm are the input of the forwarding algorithm.

- **Procedures**
  - $Next_p(d)$: 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.

- **Predicates**
  - $Consumption_p(q,m)$: $IN_p(q) = (m,d,c) \wedge d = p \wedge OUT_q(p) \neq (m,d,c)$
  - $leaf_p(q)$: $S_q = (B,?) \wedge (\forall\, q' \in N_p/\{q\}, S_{q'} \neq (B,p) \wedge (consumption_p(q) \vee OUT_p(q') = \epsilon \vee OUT_p(q') = IN_{q'}(p)))$.
  - $NO\text{-}PIF_p$: $S_p = (C, NULL) \wedge \forall q \in N_p, S_q \neq (B,?)$.
  - $init\text{-}PIF$: $S_p = (C, NULL) \wedge (\forall q \in N_p, S_q = (C, NULL)) \wedge$ $PIF\text{-}Request_p = true$.
  - $Inter\text{-}trans_p(q)$: $IN_p(q) = (m,d,c) \wedge d \neq p \wedge OUT_q(p) \neq IN_p(q) \wedge (\exists q' \in N_p/\{q\}, OUT_p(q') = \epsilon \vee OUT_p(q') = IN_{q'}(p))$.
  - $internal_p(q)$: $p \neq p_0 \wedge \neg\, leaf_p(q)$.
  - $Road\text{-}Change_p(m)$: $p = p_0 \wedge IN_p(q) = (m,d,c) \wedge d \neq p \wedge EXT_p = \epsilon \wedge OUT_q(p) \neq IN_p(q)$.
  - $\forall\, TAction \in C, B$, 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:: $init\text{-}PIF \to S_p := (B,-1)$, $PIF\text{-}Request_p := false$.
    • C-Action:: $S_p = (B,-1) \wedge \forall q \in N_p, S_q = (F,?) \to S_p := (C, NULL)$.

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

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

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

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**:: $Road\text{-}Change_p(m) \wedge OUT_p(q) \neq \epsilon \wedge OUT_p(q) \neq IN_q(p) \wedge PIF\text{-}Request_p = false \rightarrow PIF\text{-}Request_p := true.$
  - **R8**:: $Road\text{-}Change_p(m) \wedge OUT_p(q) \neq \epsilon \wedge OUT_p(q) \neq IN_q(p) \wedge PIF\text{-}Request_p \wedge B\text{-}initiator \rightarrow EXT_p := IN_p(q),\ IN_p(q) := OUT_q(p).$
  - **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**:: $p = p_0 \wedge S_p = (C, ?) \wedge PIF\text{-}Request = true \wedge [(IN_p(q) = (m, d, c) \wedge d = p) \vee IN_p(q) = \epsilon] \rightarrow PIF\text{-}Request = false.$

4 Proof of Correctness

In this section, we prove the correctness of our algorithm—due to the lack of space, the formal proofs are omitted[2] 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 finite time.

Let us first state the following lemma:

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

[2] The complete proof can be found in `http://arxiv4.library.cornell.edu/abs/1006.3432`

The proof of Lemma 1 is based on the fact that the PIF algorithm introduced here is similar to the one proposed in [7]. The only effect of the message forwarding algorithm on the PIF algorithm (w.r.t. [7]) is that a leaf is no more only defined in terms of a topology property. Here, a leaf is a dynamic property of any node. It is easy to check that this change keeps the property of snap-stabilization.

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. 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: If the output buffer of p_0 is free, then message in the extra buffer is copied in this buffer. Otherwise (the output buffer is busy), the message in the extra buffer is deleted. In both cases, the extra buffer becomes free (a free slot is created).

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.*

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), the following lemma is proven by induction:

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

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 Outline. 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.) □

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 us consider messages that are not deleted only. Let m be such 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. Hence, 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 if it not deleted before.*

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.*

From there, it is important to show that any processor can generate a message in finite time. From Lemma 6, all the messages are in suitable buffers in finite time. Since the PIF waves are used for route changes only, then:

Lemma 7. *When the routing tables are stabilized and all the messages are in suitable buffer, no PIF wave is initiated.*

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 8. *every message can be generated in finite time under a weakly fair daemon.*

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

Lemma 9. *The forwarding protocol never duplicates a valid message even if Rtables runs simultaneously.*

From Lemma 8, every message can be generated in 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 10. *Every valid message (that is generated by a processor) is never deleted unless it is delivered to its destination even if Rtables runs simultaneously.*

From Lemma 8, every message can be generated in finite time. From Lemma 10, every valid message is never deleted unless it is delivered to its destination even if *Rtables* runs simultaneously. From Lemma 9, no valid message is duplicated. Hence, the following theorem holds:

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

Note that for any processor p, the protocol delivers at most $4n - 3$ invalid messages. Indeed, the system contains only $4n - 3$ buffers and in the worst case, initially, all the buffers are busy with different invalid messages (that were not generated).

5 Network Dynamic

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 to 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 does not admit any solution. The only way is to redefine the problem in dynamic settings. 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 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 [14–16] 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, Cambridge (2000)
2. Huang, S.T., Chen, N.S.: A self-stabilizing algorithm for constructing breadth-first trees. Inf. Process. Lett. 41(2), 109–117 (1992)
3. Kosowski, A., Kuszner, L.: A self-stabilizing algorithm for finding a spanning tree in a polynomial number of moves. In: Wyrzykowski, R., Dongarra, J., Meyer, N., Waśniewski, J. (eds.) PPAM 2005. LNCS, vol. 3911, pp. 75–82. Springer, Heidelberg (2006)
4. Johnen, C., Tixeuil, S.: Route preserving stabilization. In: Huang, S.-T., Herman, T. (eds.) SSS 2003. LNCS, vol. 2704, pp. 184–198. Springer, Heidelberg (2003)
5. Awerbuch, B., Patt-Shamir, B., Varghese, G.: Self-stabilizing end-to-end communication. Journal of High Speed Networks 5(4), 365–381 (1996)
6. Kushilevitz, E., Ostrovsky, R., Rosén, A.: Log-space polynomial end-to-end communication. In: STOC 1995: Proceedings of the twenty-seventh annual ACM symposium on Theory of computing, pp. 559–568. ACM, New York (1995)
7. Cournier, A., Dubois, S., Villain, V.: Snap-stabilization and PIF in tree networks. Distributed Computing 20(1), 3–19 (2007)
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), pp. 1–11 (2009)
9. Cournier, A., Dubois, S., Villain, V.: How to improve snap-stabilizing point-to-point communication space complexity? In: Guerraoui, R., Petit, F. (eds.) SSS 2009. LNCS, vol. 5873, pp. 195–208. Springer, Heidelberg (2009)
10. Edsger, W., Dijkstra: Self-stabilizing systems in spite of distributed control. ACM Commum. 17(11), 643–644 (1974)
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, pp. 577–581 (1978)
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, pp. 384–393 (January 2003)
15. Caron, E., Desprez, F., Petit, F., Tedeschi, C.: Snap-stabilizing Prefix Tree for Peer-to-Peer Systems. In: Masuzawa, T., Tixeuil, S. (eds.) SSS 2007. LNCS, vol. 4838, pp. 82–96. Springer, Heidelberg (2007)
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), pp. 207–216. IEEE, Los Alamitos (2008)