Snap-Stabilizing Message Forwarding Algorithm on Tree Topologies*

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 consider the message forwarding problem that consists in managing the network resources that are used to forward messages. Previous works on this problem provide solutions that either use a significant number of buffers (that is n buffers per processor, where n is the number of processors in the network) making the solution not scalable or, they reserve all the buffers from the sender to the receiver to forward only one message. The only solution that uses a constant number of buffers per link was introduced in [1]. However the solution works only on a chain networks. In this paper, we propose a snap-stabilizing algorithm for the message forwarding problem that uses a constant number of buffers per link as in [1] but works on tree topologies.

Keywords: Message Forwarding, Snap-stabilization, Token Circulation.

1 Introduction

It is known that the quality of a distributed system depends on its fault tolerance. Many fault-tolerance approaches have been introduced, for instance: Self-Stabilization [2] which allows the conception of systems that are tolerant of any arbitrary transient fault. A system is said to be self-stabilizing if starting from any arbitrary configuration, the system converges into the intended behavior in a finite time. Another instance of the fault-tolerance scheme is the snap-stabilization [3]. Snap-stabilizing systems always behave according to their specification, and this regardless of the starting configuration. Thus, a snap-stabilizing solution can be seen as a self-stabilizing solution that stabilizes in zero time.

In distributed systems, 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 another node called the receiver. This problem comprises the following two sub-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. In this paper, we focus on the second problem whose aim is to design a protocol that manages the mechanism

* This work has been supported in part by the ANR project SPADES (08-ANR-SEGI-025). Details of the project on `http://graal.ens-lyon.fr/SPADES`

L. Bononi et al. (Eds.): ICDCN 2012, LNCS 7129, pp. 46–60, 2012.

allowing the message to move from a node to another one on the path from a sender to a receiver. Each node on this path has a reserved memory space called buffer. We assume that each buffer is large enough to contain any message. With a finite number of buffers, the message forwarding problem consists in avoiding deadlock and livelock situations.

The message forwarding problem has been well investigated in a non faulty setting [4–7]. In [8, 9] self-stabilizing solutions were proposed. Both solutions deal with network dynamic, *i.e.,* systems in which links can be added or removed. However, message deliveries are not ensured while the routing tables are not stabilized. Thus, the proposed solutions cannot guaranty the absence of message loss during the stabilization time.

In this paper, we address the problem of providing a snap-stabilizing protocol for this problem. Snap-stabilization provides the desirable property of delivering to its recipient every message generated after the faults, once and only once even if the routing tables are not (yet) stabilized. Some snap-stabilizing solutions have been proposed to solve the problem [10, 11, 1]. In [10], the problem was solved using n buffers per node (where n denotes the number of processors in the network). The number of buffers was reduced in [11] to D buffers per node (where D refers to the diameter of the network). However, the solution works by reserving the entire sequence of buffers leading from the sender to the receiver. Note that the first solution is not suitable for large-scale systems whereas the second one has to reserve all the path from the source to the destination for the transmission of only one message. In [1], a snap-stabilizing solution was proposed using a constant number of buffers per link. However the solution works only on chain topologies.

We provide a snap-stabilizing solution that solves the message forwarding problem in tree topologies using the same complexity on the number of buffers as in [1] *i.e.,* two buffers per link for each processor plus one internal buffer, thus, $2\delta + 1$ buffers by processor, where δ is the degree of the processor in the system.

Road Map. The rest of the paper is organized as follow: Our Model is presented in Section 2. In Section 3, we provide our snap-stabilizing solution for the message forwarding problem. Due to the lack of space, the complete proofs of correctness are omitted however, some sketches of proofs are given in Sub-Section 3.3. Finally we conclude the paper in Section 4.

2 Model and Definitions

Network. We consider in this paper 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. Each process has a unique *id*. Two processors p and q are said to be neighbours if and only if there is a communication link (p, q) between the two processors. 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 tree of n processors.

Computational Model. In this paper we consider the classical local shared memory model introduced by Dijkstra [12] 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* [13]. 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 $\forall\ E \in \xi$, E satisfies S_Γ.

Message Forwarding Problem. In the following, a message is said to be valid if it has been emitted after the faults. Otherwise it is said to be invalid.

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: (i) Any message can be generated in a finite time. (ii) Any valid message is delivered to its destination once and only once in a finite time.*

Buffer Graph. A Buffer Graph [14] is defined as a directed graph on the buffers of the graph *i.e.,* the 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 only permitted between buffers in the same node, or between buffers in distinct nodes which are connected by a communication link.

3 Message Forwarding

In this section, we first give an overview of our snap stabilizing Solution for the message forwarding problem, then we present the formal description followed by some sketches of the proofs of correctness.

3.1 Overview of the Solution

In this section, we provide an informal description of our snap stabilizing solution that solves the message forwarding problem and tolerates the corruption of the routing tables in the initial configuration. We assume that there is a self-stabilizing algorithm 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. In the following we assume that there is no message in the system whose destination is not in the system.

Before detailing our solution let us define the buffer graph used in our solution. Let $\delta(p)$ be the degree of the processor p in the tree structure. Each processor p has (i) one internal buffer that we call Extra buffer denoted EXT_p. (ii) $\delta(p)$ input buffers allowing p to receive messages from its neighbors. Let $q \in N_p$, the input buffer of p connected to the link (p, q) is denoted by $IN_p(q)$. (iii) $\delta(p)$ output buffers allowing it to send messages to its neighbors. Let $q \in N_p$, the output buffer of p connected to the link (p, q) is denoted by $OUT_p(q)$. In other words, each processor p has $2\delta(p) + 1$ buffers. 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 its destination.

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 its destination. Once the message is on system, it is routed according to the routing tables: let us refer to $nb(m, b)$ as the next buffer b' of the message m stored in b, $b \in \{IN_p(q) \vee OUT_p(q)\}$, $q \in N_p$. We have the following properties:

1. $nb(m, IN_p(q)) = OUT_p(q')$ such as q' is the next process by which m has to transit to reach its destination.
2. $nb(m, OUT_p(q)) = IN_q(p)$

Thus, if the message m is in the Output buffer $OUT_p(q)$ such as p is not the destination then it will be automatically copied to the Input buffer of q. If the the message m is in the Input buffer of p ($IN_p(q)$) then if p is not the destination

it consults the routing tables to determine which is the next process by which the message has to pass in order to meet its destination.

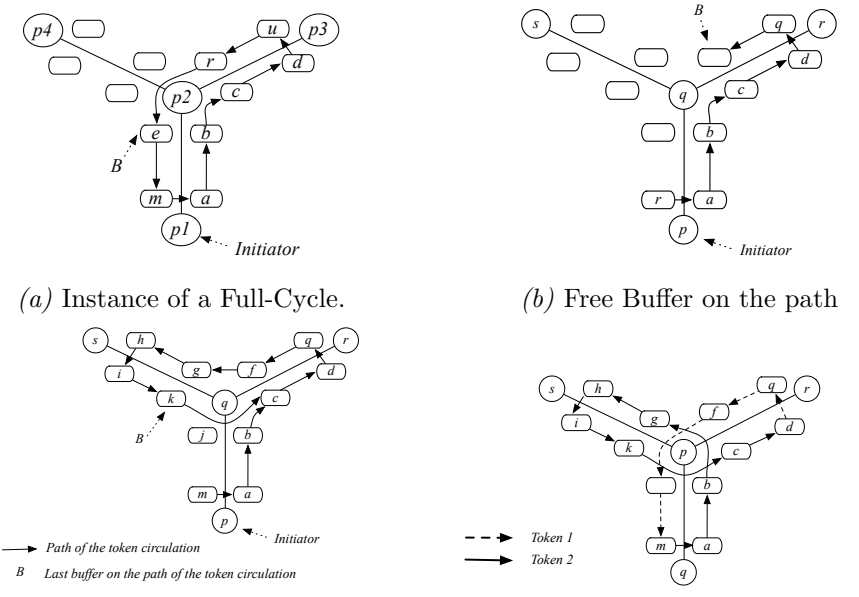
Note that when the routing tables are stabilized and when all the messages are in the right direction, the first property $nb(m, IN_p(q)) = OUT_p(q')$ is never verified for $q = q'$. However, this is not true when the routing tables are not yet stabilized and when some messages are in the wrong direction.

Let us now recall the message progression. 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. 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.

In the sequel, let us consider our buffer graph taking in account only active arcs (an arc is said to be active if it starts from a non empty buffer). Observe that in this case the sub graph introduced by the active arcs can be seen as a resource allocation graph where the buffers correspond to the resources, for instance if there is a message m in $IN_p(q)$ such as $nb(m, IN_p(q)) = OUT_{q'}(p)$ then m is using the resource (buffer) $IN_p(q)$ and it is asking for another resource which is the output buffer $OUT_p(q')$. We will refer to this sub graph as the active buffer graph.

It is known in the literature that a deadlock situation appears only in the case there exists a cycle in the resource allocation graph. Note that this is also the case in our active buffer graph. Since due to some initial configurations of the forwarding algorithm and (or) the routing tables construction, this kind of cycles can appear during a finite prefix of any execution (refer to Figure 1, (a)). Observe also that because our buffer graph is built on a tree topology, if a cycle exists then we are sure that there are at least two messages m and m' that verifies the following condition: $nb(m, IN_p(q)) = OUT_p(q) \wedge nb(m', IN_{p'}(q')) = OUT_{p'}(q')$ (see messages m and d in Figure 1, (a)). Since in this paper we consider a distributed system, it is impossible for a processor p to know whether there is a cycle in the system or not if no mechanism is used to detect them. The only thing it can do is to suspect the presence of a cycle in the case there is one message in its input buffer $IN_p(q)$ that has to be sent to $OUT_p(q)$. In order to verify that, p initiates a token circulation that follows the active buffer graph starting from the input buffer containing the message m. By doing so, the token circulation either finds a free buffer (refer to Figure 1, (b)) or detects a cycle. Note that two kinds of cycle can be detected: (i) a Full-Cycle involving the first input buffer containing m (refer to Figure 1, (a)) or (ii) a Sub-Cycle that does not involve the input buffer containing the message m (refer to Figure 1, (c)).

If the token circulation has found an empty buffer (Let refer to this buffer by B), the idea is to move the messages along the token circulation path to make the free slot initially on B move. By doing so, we are sure that $OUT_p(q)$ becomes free. Thus, p can copy the message m directly to $OUT_p(q)$ (Note that this action has the priority on all the other enabled actions). If the token circulation has detected a cycle then two sub-cases are possible according to the type of the cycle that has been detected: (i) The case of a Full-Cycle: Note that in this case p is the one that detects the cycle (p_1 in Figure 1, (a)) . The aim will be to release



(a) Instance of a Full-Cycle. *(b)* Free Buffer on the path

(c) Instance of a Sub-Cycle. *(d)* Token circulations deadlocked.

Fig. 1. Instance of token circulations

$OUT_p(q)$. (ii) The case of a Sub-Cycle: In this case the processor containing the last buffer B that is reached by the token is the one that detects the cycle (Processor p_2 in Figure 1, (c)). Observe that B is an input buffer. The aim in this case is to release the output buffer B' by which the message m in B has to be forwarded to in order to meet its destination ($OUT_{p_2}(p_3)$ in Figure 1, (c)). Note that B' is in this case part of the path of the token circulation. In both cases (i) and (ii), the processor that detects the cycle copies the message from the corresponding input buffer (either from $IN_p(q)$ or B) to its extra buffer. By doing so the processor releases its input buffer. The idea is to move messages on the token circulation's path to make the free slot that was created on the input buffer move. This ensures that the corresponding output buffer will be free in a finite time (either $OUT_p(q)$ or B'). Thus, the message in the extra buffer can be copied to the free slot on the output buffer. Thus, one cycle has been broken.

Note that many token circulations can be executed in parallel. To avoid deadlock situations between the different token circulations (refer to Figure 1, (d)), each token circulation carries the identifier of its initiator. The token circulation with an identifier id can use a buffer of another token circulation having the identifier id' if $id < id'$. Note that by doing so, one token circulation can break the path of another one when the messages move to escort the free slot. The free slot can be then lost. For instance, in Figure 2, we can observe that the free slot that was produced by $T1$ is taking away by $T2$. Note that by

moving messages on the path of $T2$, a new cycle is created again, involving p_1 and p_4, if we suppose that the same thing happens again such that the extra buffer of p_4 becomes full and that p_4 and p_1 becomes involved again in the another cycle then the system is deadlocked and we cannot do anything to solve it since we cannot erase any valid message. Thus, we have to avoid to reach such a configuration dynamically. To do so, when a token circulation finds either a free buffer or detect a cycle, it does the reverse path in order to validate its path. Thus, when the path is validated no other token circulation can use a buffer that is already in the validated path. Note that the token is now back to the initiator. To be sure that all the path of the token circulation is a correct path (it did not merge with another token circulation that was in the initial configuration), the initiator sends back the token to confirm all the path. On another hand, since the starting configuration can be an arbitrary configuration, we may have in the system a path of a token circulation (with no initiator) that forms a cycle. To detect and release such a situation, a value is added to the state of each buffer in the following manner: If the buffer B_i has the token with the value x, then when the next buffer B_{i+1} receive the token it will set it value at $x + 1$. Thus, we are sure that in the case there is a cycle there will be two consecutive buffers B and B' having respectively x and x' as a value in the path of the cycle such as $x' \neq x + 1$. Thus, this kind of situation can be detected and solved.

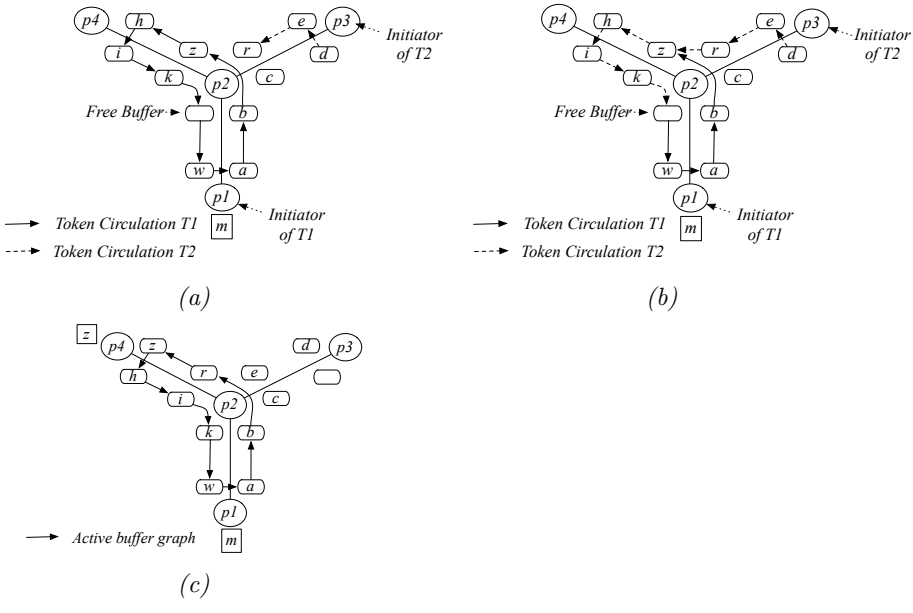


Fig. 2. Instance of a problem

3.2 Formal Description of the Solution

In this section we first define the data and variables that are used for the description of our algorithms. We then present the formal description of both the Token Circulation algorithm and the message forwarding algorithm.

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

– **Procedures**

- $Next_p(d)$: refers to the neighbour of p given by the routing tables 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.

– **Variables**

- N_p: The set of identities of the neighbors of the processor p.
- $IN_p(q)$: 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.
- $S_{pqi} = (id, previous, next, phase, x)$: refers to the state of the input buffer of the process p on the link (p,q). id refers to the identity of the process that initiates the token circulation. previous is a pointer towards the output buffer from which the buffer pqi received the token (it refers to the output buffer of q on the link (q,p)). next is also a pointer that shows the next buffer that received the token from the input buffer of p on the link (p,q). phase $\in \{S, V, F, C, E\}$ defines the state of the token circulation to determine which phase is executed respectively (Search, Validation, Confirm, Escort or non of these "Clean" State). x is an integer which will be used in order to break incorrect cycles.
- $S_{pqo} = (id, previous, next, phase, x)$: As for the input buffer, $S_{pqo} = (id, previous, next, phase, x)$ refers to the state of the output buffer of the process p connected to the link (p,q). The attributes have the same meaning as previously.
- $prev_{pqo} : q' \in N_p$ such as $S_{pq'i} = (id_{q'}, q'po, pqo, S, ?) \wedge id_{q'} = min\{id_{q''}, q'' \in N_p \wedge S_{pq''i} = (id_{q''}, q'po, pqo, S, ?)\}$.
- $Small_p$: $q \in N_p$ such as $\exists\ q' \in N_p$, $S_{pqi} = (id_q, ?, pq'o, F, x) \wedge S_{pq'o} = (id_q, X, q'pi, F, z) \wedge X \neq pqi \wedge z \neq x + 1 \wedge id_q = min\{id_{q''}, q'' \in N_p \wedge S_{pq''i} = (id_{q''}, ?, pro, F, x') \wedge S_{pro} = (id_{q''}, X', rpi, F, z') \wedge X' \neq pq''i \wedge z' \neq x' + 1$.

– **Predicates**

- $NO - Token_p$: $\forall\ q \in N_p$, $S_{pqi} = (-1, NULL, NULL, C, -1) \wedge S_{pqo} = (-1, ?, ?, ?) \wedge S_{qpo} = (-1, NULL, NULL, C, -1)$

– 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. Token circulation — Initiation and Transmission

Token initiation

R1: $Token_p(q) \wedge S_{pqo} = (-1, NULL, NULL, C, -1) \wedge S_{pqo} = S_{pqi} \rightarrow S_{pqi} := (p, NULL, pqo, S, 0)$, $S_{pqo} := (p, pqi, qpi, S, 1)$

Token transmission

- **Search phase**
  - **R2**: $\exists\, q, q' \in N_p$, $S_{qpo} = (id, ?, pqi, S, x) \wedge IN_p(q) = (m, d, c) \wedge Next_p(d) = q' \wedge S_{pq'o} \neq (id, ?, ?, ?, ?) \wedge S_{pqi} \neq (id', ?, ?, V \vee F \vee E, ?) \wedge (S_{pqi} \neq (id'', ?, ?, ?, ?) \wedge id'' <= id) \rightarrow S_{pqi} := (id, qpo, pq'o, S, x + 1)$
  - **R3**: $\exists\, q, q' \in N_p$, $prev_{pqo} = q' \wedge S_{pq'i} = (id, q'po, pqo, S, x) \wedge (S_{qpi} \neq (id, ?, ?, ?, ?) \wedge S_{pqo} \neq (id'', ?, ?, V \vee F \vee E, ?) \wedge S_{pqo} \neq (id', ?, ?, ?, ?) \wedge id' <= id \wedge OUT_p(q) \neq \epsilon \wedge OUT_p(q) \neq IN_q(p) \rightarrow S_{pqo} := (id, pq'i, qpi, S, x + 1)$

- **Validation phase**
  - **Initiation**
    - **R4**: $\exists q, q' \in N_p$, $prevpqo = q' \wedge S_{pq'i} = (id, q'po, pqo, S, x) \wedge S_{pqo} \neq (id'', ?, ?, V \vee F \vee E, ?) \wedge S_{pqo} \neq (id', ?, ?, ?, ?) \wedge id' < id \wedge S_{qpi} = (id, X, ?, S, ?) \wedge X \neq pqo \wedge OUT_p(q) \neq \epsilon \wedge OUT_p(q) \neq IN_q(p) \rightarrow S_{pqo} := (id, pq'i, qpi, V, x + 1)$
    - **R5**: $\exists\, q, q' \in N_p$, $S_{qpo} = (id, ?, pqi, S, x) \wedge IN_p(q) = (m, d, c) \wedge Next_p(d) = q' \wedge S_{pq'o} = (id, X, ?, S, z) \wedge X \neq pqi \wedge EXT_p = \epsilon \wedge S_{pqi} \neq (id', ?, ?, V \vee F \vee E, ?) \wedge (S_{pqi} \neq (id'', ?, ?, ?, ?) \wedge id'' < id) \rightarrow S_{pqi} := (id, qpo, pq'o, V, x + 1)$
    - **R6**: $\exists\, q, q' \in N_p$, $prev_{pqo} = q' \wedge S_{pq'i} = (id, q'po, pqo, S, x) \wedge [(OUT_p(q) = \epsilon \vee OUT_p(q) = IN_q(p))] \rightarrow S_{pqo} := (id, pq'i, NULL, V, x + 1)$
    - **R7**: $\exists\, q, q' \in N_p$, $S_{qpo} = (id, ?, pqi, S, x) \wedge IN_p(q) = \epsilon \rightarrow S_{pqi} := (id, qpo, NULL, V, x + 1)$
  - **Transmission**
    - **R8**: $\exists\, q, q' \in N_p$, $S_{pqo} = (id, pq'i, qpi, S, x) \wedge S_{qpi} = (id, pqo, ?, V, x + 1) \wedge x \neq 1 \wedge S_{pq'i} \neq (id, ?, pqo, F, x - 1) \rightarrow S_{pqo} := (id, ?, qpi, V, x)$
    - **R9**: $\exists\, q, q' \in N_p$, $S_{pqi} = (id, qpo, pq'o, S, x) \wedge S_{pq'o} = (id, pqi, ?, V, x + 1) \wedge S_{qpo} \neq (id, ?, pqi, F, x - 1) \rightarrow S_{pqi} := (id, qpo, pq'o, V, x)$

- **Confirm phase**
  - **Initiation**
    - **R10**: $\exists\, q \in N_p$, $S_{pqo} = (p, pqi, qpi, S, 1) \wedge S_{pqi} = (p, NULL, pqo, S, 0) \wedge S_{qpi} = (p, pqo, ?, V, 2) \rightarrow S_{pqo} := (p, pqi, qpi, F, 1)$, $S_{pqi} := (p, NULL, pqo, F, 0)$
  - **Transmission**
    - **R11**: $\exists\, q, q' \in N_p$, $S_{qpo} = (id, ?, pqi, F, x) \wedge S_{pqi} = (id, qpo, pq'o, V, x + 1) \rightarrow S_{pqi} := (id, qpo, pq'o, F, x + 1)$
    - **R12**: $\exists\, q, q' \in N_p$, $prev_{pqo} = q' \wedge S_{pq'i} = (id, ?, pqo, F, x) \wedge S_{pqo} = (id, pq'i, qpi, V, x + 1) \rightarrow S_{pqo} := (id, pq'i, qpi, F, x + 1)$

- **Escort phase**
  - **Initiation**
    - **R13**: $\exists\, q \in N_p$, $S_{pqi} = (id, idle, pqo, F, 0) \wedge S_{qpo} = (id, ?, pqi, F, x) \wedge x \geq 3 \wedge S_{pqo} = (id, pqi, qpi, F, 1) \wedge EXT_p = \epsilon \rightarrow S_{pqi} := (id, idle, pqo, E, 0)$
    - **R14**: $Small_p = q \wedge \exists\, q' \in N_p$, $S_{pqi} = (id, qpo, pq'o, F, x) \wedge S_{pq'o} = (id, X, q'pi, F, z) \wedge X \neq pqi \wedge z \neq x + 1 \wedge EXT_p = \epsilon \wedge \nexists\, q'' \in N_p$, $(S_{pq''i} = (id', NULL, Z, F, 0) \wedge S_Z = (id', pq''i, ?, F, 1)) \rightarrow S_{pqi} := (id, qpo, pq'o, E, x)$
    - **R15**: $\exists\, q \in N_p$, $S_{qpo} = (id, ?, pqi, F, x) \wedge S_{pqi} = (id, qpo, idle, V, x+1) \wedge IN_p(q) = \epsilon \rightarrow S_{pqi} := (id, qpo, idle, E, x + 1)$
    - **R16**: $\exists\, q, q' \in N_p$, $S_{pqi} = (id, qpo, pq'o, F, x) \wedge S_{pq'o} = (id, pqi, idle, V, x + 1) \wedge [OUT_p(q) = \epsilon \vee OUT_p(q) = IN_q(p)] \rightarrow S_{pq'o} := (id, pqi, idle, E, x + 1)$
  - **Propagation**
    - **R17**: $\exists q, q' \in N_p$, $S_{pqo} = (id, ?, qpi, F, x) \wedge S_{qpi} = (id, pqo, ?, E, x + 1 \vee 0) \rightarrow S_{pqo} := (id, ?, qpi, E, x)$
    - **R18**: $\exists q, q' \in N_p$, $S_{pqi} = (id, qpo, pq'o, F, x) \wedge S_{pq'o} = (id, pqi, q'pi, E, x + 1) \rightarrow S_{pqi} := (id, qpo, pq'o, E, x)$
    - **R19**: $\exists q \in N_p$, $S_{pqo} = (id, pqi, qpi, F, 1) \wedge S_{qpi} = (id, idle, pqo, E, 0) \wedge S_{qpi} = (id, pqo, ?, E, 2) \rightarrow S_{pqo} := (id, pqi, qpi, E, 1)$

Algorithm 2. Token Circulation — Cleaning Phase and Correction

- **-Cleaning phase**
  - **Initiation**
    - **R20**: $\exists\ q \in N_p,\ S_{pqi} = (id, NULL, pqo, E, 0) \wedge S_{pqo} = (id, pqi, qpi, E, 1) \rightarrow S_{pqi} := (-1, NULL, NULL, C, -1)$
    - **R21**: $\exists\ q, q' \in N_p,\ S_{pq'i} = (id, q'po, pqo, E, x) \wedge S_{pqo} = (id, X, qpi, E, z) \wedge X \neq pq'i \rightarrow S_{pq'i} := (-1, NULL, NULL, C, -1)$
    - **R22**: $\exists\ q \in N_p,\ S_{pqi} = (id, qpo, NULL, E, x) \wedge S_{qpo} = (id, ?, pqi, E, x-1) \rightarrow S_{pqi} := (-1, NULL, NULL, C, -1)$
    - **R23**: $\exists\ q, q' \in N_p,\ S_{pqo} = (id, pq'i, qpi, E, x) \wedge S_{pq'i} = (id, q'po, pqo, E, x-1) \rightarrow S_{pq'i} := (-1, NULL, NULL, C, -1)$
  - **Propagation**
    - **R24**: $\exists\ q \in N_p,\ S_{pqo} = (id, X, qpi, E, x) \wedge S_X \neq (id, ?, pqo, F, x-1) \wedge [(S_{qpi} = (id', ?, ?, ?, ?) \wedge id \neq id') \vee S_{qpi} = (-1, NULL, NULL, C, -1)] \rightarrow S_{pqo} := (-1, NULL, NULL, C, -1)$
    - **R25**: $\exists\ q \in N_p,\ S_{pqi} = (id, qpo, pq'o, E, x)\ S_{qpo} \neq (id, ?, pqi, F, x-1) \wedge [(S_{pq'o} = (id', ?, ?, ?, ?) \wedge id \neq id') \vee S_{pq'o} = (-1, NULL, NULL, C, -1)] \rightarrow S_{pqo} := (-1, NULL, NULL, C, -1)$

- **Correction rules**
  - **Freeze Cleaning**
    - **Initiation**
      - **R26**: $\exists q, q' \in N_p,\ S_{pqo} = (id, pq'i, qpi, S \vee V \vee F, ?) \wedge [S_{pq'i} = (-1, NULL, NULL, C, -1) \vee (S_{pq'i} = (id', ?, ?, ?, ?) \wedge id' \neq id) \vee (S_{pq'i} = (id, ?, M, ?, ?) \wedge M \neq pqo)] \rightarrow S_{pqo} := (id, pq'i, qpi, G, ?)$
      - **R27**: $\exists q, q' \in N_p,\ S_{pqi} = (id, qpo, pq'o, ?, ?) \wedge (S_{qpo} = (-1, NULL, NULL, C, -1) \vee (S_{qpo} = (id', ?, ?, ?, ?) \wedge id' \neq id)) \rightarrow S_{pqi} := (id, qpo, pq'o, G, ?)$
      - **R28**: $S_{pqi} = (p, NULL, pqo, ?, x) \wedge x > 0 \rightarrow S_{pqi} := (p, NULL, pqo, G, x)$
      - **R29**: $\exists\ q, q' \in N_p,\ S_{pqi} = (id, ?, pq'o, ?, x) \wedge S_{pqo} = (id, pq'i, qpi, ?, z) \wedge z \neq x+1 \rightarrow S_{pqi} := (id, ?, pq'o, G, x)$
      - **R30**: $\exists\ q \in N_p,\ S_{pqi} = (id, qpo, pqo, ?, x) \wedge S_{qpo} = (id, ?, pqi, ?, z) \wedge z \neq x+1 \rightarrow S_{pqi} := (id, qpo, pqo, G, x)$
      - **R31**: $\exists\ q \in N_p,\ [(S_{pqo} = (id, ?, qpi, S, x) \wedge S_{qpi} = (id, pqo, ?, F \vee E, x+1)) \vee (S_{pqo} = (id, ?, qpi, F, x) \wedge S_{qpi} = (id, pqo, ?, S, x+1)) \vee (S_{pqo} = (id, ?, qpi, V, x) \wedge S_{qpi} = (id, pqo, ?, E \vee S, x+1))] \rightarrow S_{pqo} := (id, ?, qpi, G, x)$
      - **R32**: $\exists\ q, q' \in N_p,\ [(S_{pqi} = (id, ?, pq'o, S, x) \wedge S_{pq'o} = (id, pqi, ?, F \vee E, x+1)) \vee (S_{pqi} = (id, ?, pq'o, F, x) \wedge S_{pq'o} = (id, pqi, ?, S, x+1)) \vee (S_{pqi} = (id, ?, pq'o, V, x) \wedge S_{pq'o} = (id, pqo, ?, E \vee S, x+1))] \rightarrow S_{pqi} := (id, ?, pq'o, G, x)$
    - **Propagation**
      - **R33**:$\exists\ q, q' \in N_p,\ S_{qpo} = (id, ?, pqi, G, ?) \wedge S_{pqi} = (id, qpo, pq'o, S \vee V \vee F \vee E, ?) \rightarrow S_{pqi} := (id, qpo, pq'o, G, ?)$
      - **R34**: $\exists\ q, q' \in N_p,\ prev_{pqo} = q \wedge S_{pqi} = (id, ?, pq'i, G, ?) \wedge S_{pq'o} = (id, qpo, q'pi, S \vee V \vee F \vee E, ?) \rightarrow S_{pq'o} := (id, pqi, q'pi, G, ?)$
    - **Cleaning**
      - **R35**: $\exists q, q' \in N_p,\ S_{pqi} = (id, qpo, pq'o, G, x) \wedge [S_{pq'o} = (-1, NULL, NULL, C, -1) \vee (S_{pq'o} = (id', ?, ?, ?, ?) \wedge id' \neq id) \vee (S_{pq'o} = (id, ?, qpi, G, z) \rightarrow S_{pqi} := (-1, NULL, NULL, C, -1)$
      - **R36**: $\exists q, q' \in N_p,\ S_{pqo} = (id, pq'i, qpi, G, x) \wedge [S_{qpi} = (-1, NULL, NULL, C, -1) \vee (S_{qpi} = (id', ?, ?, ?, ?) \wedge id' \neq id) \vee (S_{qpi} = (id, pqo, ?, G, z) \rightarrow S_{pqo} := (-1, NULL, NULL, C, -1)$
  - **R37**: $\exists\ q, q' \in N_p,\ S_{pqi} = (id, ?, pq'o, G, x) \wedge S_{pq'o} = (id, pqi, q'pi, G, z) \wedge z \neq x+1 \rightarrow S_{pqi} := (-1, NULL, NULL, C, -1)$
  - **R38**: $\exists\ q \in N_p,\ S_{pqi} = (id, qpo, pqo, G, x) \wedge S_{qpo} = (id, ?, pqi, G, z) \wedge z \neq x+1 \rightarrow S_{pqi} := (-1, NULL, NULL, C, -1)$
  - **R39**: $Token_p(q) \wedge S_{pqi} = (p, ?, ?, ?, ?) \rightarrow Token_p(q) := false$
  - **R40**: $\exists\ q, q' \in N_p,\ S_{pqi} = (id, qpo, pq'o, F, x) \wedge S_{pq'o} = (id, X, q'pi, S \vee V, z) \wedge z \neq x+1 \rightarrow S_{pqi} := (-1, NULL, NULL, C, -1)$
  - **R41**: $\exists\ q \in N_p,\ S_{pqi} = (id, qpo, NULL, S \vee V \vee F, x) \wedge IN_p(q) \neq \epsilon \rightarrow S_{pqi} := (-1, NULL, NULL, C, -1)$
  - **R42**: $\exists\ q, q' \in N_p,\ S_{pqo} = (id, pq'i, NULL, S \vee V \vee F, x) \wedge OUT_p(q) \neq \epsilon \rightarrow S_{pqo} := (-1, NULL, NULL, C, -1)$
  - **R43** $\exists\ q \in N_p,\ S_{pqo} = (id, ?, qpi, V \vee F, x) \wedge [(S_{qpi} = (id', ?, ?, ?, ?) \wedge id \neq id') \vee S_{qpi} = (-1, NULL, NULL, C, -1)] \rightarrow S_{pqo} := (-1, NULL, NULL, C, -1)$
  - **R44** $\exists\ q \in N_p,\ S_{pqi} = (id, qpo, pq'o, V \vee F \vee E, x) \wedge [(S_{pq'o} = (id', ?, ?, ?, ?) \wedge id \neq id') \vee S_{pq'o} = (-1, NULL, NULL, C, -1)] \rightarrow S_{pqo} := (-1, NULL, NULL, C, -1)$

Algorithm 3. Message Forwarding

- **Message generation (For every processor)**
  R'1: $Request_p \wedge Next_p(d) = q \wedge [OUT_p(q) = \epsilon \vee OUT_p(q) = IN_q(p)] \wedge NO-Token \rightarrow OUT_p(q) := (m, d, choice(c))$, $Request_p := false$.

- **Message consumption (For every processor)**
  R'2: $\exists q \in N_p,\ IN_p(q) = (m, d, c) \wedge d = p \wedge OUT_q(p) \neq IN_p(q) \rightarrow deliver_p(m)$, $IN_p(q) := OUT_q(p)$.

- **Internal transmission**
  R'3: $\exists q, q' \in N_p,\ IN_p(q) = (m, d, c) \wedge d \neq p \wedge Next_p(d) = q' \wedge q' \neq q \wedge [OUT_p(q') = \epsilon \vee OUT_p(q') = IN_{q'}(p)] \wedge OUT_q(p) \neq IN_p(q) \wedge NO-Token \rightarrow OUT_p(q') := (m, d, choice(c))$, $IN_p(q) := OUT_q(p)$.

  R'4: $\exists\ q, q' \in N_p,\ IN_p(q') = (m, d, c) \wedge OUT_{q'}(p) \neq IN_p(q') \wedge [OUT_p(q) = \epsilon \vee OUT_p(q) = IN_q(p)] \wedge S_{pqo} = (id, pq'i, qpi, E, x+1) \wedge S_{pq'i} = (id, q'po, pqo, F, x) \rightarrow OUT_p(q) := (m, d, choice(c))$, $IN_p(q') := OUT_{q'}(p)$

- **Message transmission from q to p**
  R'5: $\exists q \in N_p,\ IN_p(q) = \epsilon \wedge OUT_q(p) = (m, d, c) \wedge q \neq d \wedge NO-Token \rightarrow IN_p(q) := OUT_q(p)$.

  R'6: $\exists q \in N_p,\ IN_p(q) = \epsilon \wedge OUT_q(p) = (m, d, c) \wedge q \neq d \wedge S_{pqi} = (id, qpo, ?, E, x+1) \wedge S_{qpo} = (id, ?, pqi, E, x) \rightarrow IN_p(q) := OUTq(p)$

- **Erasing a message after its transmission**
  R'7: $\exists q \in N_p,\ OUT_p(q) = IN_q(p) \wedge (\forall q' \in N_p \setminus \{q\},\ IN_p(q') = \epsilon \vee (IN_p(q') = (m, d, c) \wedge Next_p(d) \neq q)) \wedge NO-Token \rightarrow OUT_p(q) := \epsilon$

- **Erasing a message after its transmission (For the leaf processors)**
  R'8: $N_p = \{q\} \wedge OUT_p(q) = IN_q(p) \wedge (IN_p(q) = \epsilon \vee (IN_p(q) = (m, d, c) \wedge Next_p(d) \neq q)) \wedge NO-Token \rightarrow OUT_p(q) := \epsilon$

- **Road change**
  R'9: $\exists\ q \in N_p,\ IN_p(q) = (m, d, c) \wedge Next_p(d) = q \wedge OUT_q(p) \neq IN_p(q) \wedge (OUT_p(q) = \epsilon \vee OUT_p(q) = IN_q(p)) \rightarrow OUT_p(q) := IN_p(q)$, $IN_p(q) := OUT_q(p)$

  R'10: $\exists\ q \in N_p, IN_p(q) = (m, d, c) \wedge Next_p(d) = q \wedge OUT_p(q) \neq \epsilon \wedge OUT_p(q) \neq IN_q(p) \wedge EXT_p = \epsilon \wedge \nexists\ q' \in N_p/\{q\},\ S_{pq'i} = (id, ?, ?, ?, 0) \rightarrow Token_p(q) := true$

  R'11: $\exists\ q \in N_p,\ S_{pqi} = (id, idle, pqo, F, 0) \wedge S_{qpo} = (id, ?, pqi, F, x) \wedge x \geq 3 \wedge S_{pqo} = (id, pqi, qpi, F, 1) \wedge EXT_p = \epsilon \rightarrow EXT_p := IN_p(q)$, $IN_p(q) := OUTq(p)$

  R'12: $\exists\ q \in N_p,\ Small_p = q \wedge S_{qpo} = (id, ?, pqi, F, x-1) \wedge \nexists\ q'' \in N_p, (S_{pq''i} = (id', NULL, Z, F, 0) \wedge S_Z = (id', pq''i, ?, F, 1)) \rightarrow EXT_p := IN_p(q)$, $IN_p(q) := OUTq(p)$

  R'13: $\exists\ q \in N_p\ S_{pqi} = (id, NULL, pqo, E, 0) \wedge S_{pqo} = (id, pqi, qpi, F, 1) \wedge S_{qpi} = (id, pqo, ?, E, 2) \wedge EXT_p \neq \epsilon \wedge (OUT_p(q) = \epsilon \vee OUT_p(q) = IN_q(p)) \rightarrow OUT_p(q) := EXT_p$, $EXT_p := \epsilon$

  R'14: $\exists\ q, q' \in N_p\ S_{pq'i} = (id, q'po, pqo, E, x) \wedge S_{pqo} = (id, X, qpi, F, z) \wedge X \neq pq'i \wedge z \neq x+1 \wedge S_{qpi} = (id, pqo, ?, E, z+1) \wedge EXT_p \neq \epsilon \wedge (OUT_p(q) = \epsilon \vee OUT_p(q) = IN_q(p)) \rightarrow OUT_p(q) := EXT_p$, $EXT_p := \epsilon$

- **Correction Rules**
  R'15: $EXT_p \neq \epsilon \wedge (NO-Token \wedge (\forall\ q \in N_p,\ S_{pqi} \neq (id, qpo, ?, E)) \wedge (\exists\ q \in N_p,\ S_{pqi} = (id, NULL, pqo, E, 0) \wedge S_{pqo} = (id, pqi, qpi, E, 1) \wedge OUT_p(q) \neq \epsilon \wedge OUT_p(q) \neq IN_q(p)) \rightarrow EXT_p := \epsilon$

  R'16: $EXT_p \neq \epsilon \wedge (NO-Token \wedge (\forall\ q \in N_p,\ S_{pqi} \neq (id, qpo, ?, E)) \wedge (\exists\ q, q' \in N_p, S_{pq'i} = (id, ?, pqo, E, x) \wedge S_{pqo} = (id, X, qpi, E, z) \wedge X \neq pq'i \wedge z \neq x+1 \wedge OUT_p(q) \neq \epsilon \wedge OUT_p(q) \neq IN_q(p)) \rightarrow EXT_p := \epsilon$

  R'17: $Token_p(q) = true \wedge IN_p(q) = \epsilon \vee IN_p(q) = (m, d, c) \wedge Next_p(d) \neq q \rightarrow Token_p(q) = false$

3.3 Proof of Correctness

In this section, due to the lack of space[1], we outline correctness proofs of our solution only.

The idea of the proofs is as follows: we first show that no valid message is deleted from the system unless it is delivered to its destination. We then show that each buffer is infinitely often free, thus, neither deadlocks nor starvation appear in the system. We finally show that every valid message is delivered to its destination once and only once in a finite time.

Let us define first some notions that will be used later.

Definition 1. *Let B_1 and B_2 be two buffers and let p, q and q' be processors in the network such as one of those properties holds: (i) $B_1 = IN_p(q) \wedge B_2 = OUT_p(q')$ (ii) $B_1 = OUT_p(q) \wedge B_2 = IN_q(p)$. $B2$ is the successor of $B1$ denoted by $B_1 \mapsto B_2$ if and only if $S_{B_1} = (id, ?, B_2, ?, x) \wedge S_{B_2} = (id, B_1, ?, ?, x + 1)$*

Definition 2. *A sequence of k buffers $B_1 \mapsto B_2 \mapsto ... \mapsto B_k$ starting from B_1 is called an abnormal sequence if the following property holds:*
$S_{B_1} = (id, ?, ?, ?, ?) \wedge (B_1 = IN_p(q) \vee B_1 = OUT_p(q)) \wedge id \neq p$

A buffer B is said to be cleared if $S_B = (-1, NULL, NULL, C, -1)$. In the same manner, a sequence is said to be cleared, if all the buffers part of it becomes cleared in a finite time. Let us state the following lemma:

Lemma 1. *If the configuration contains an abnormal sequence of buffers $S_1 = B_1 \mapsto B_2 \mapsto ... \mapsto B_k$, then S_1 will be cleared in a finite time.*

The proof of Lemma 1 is based on the fact that the processor p that has $B1$ as a buffer will be able to detect this abnormal sequence (p is not the initiator of the token circulation and B_1 sent the token without receiving it from any other buffer). p will initiate a Freeze cleaning phase by executing either $R24$ or $R25$ or $R26$. Thus, the state of B_1 will be referring to the freeze cleaning phase. This special phase will be propagated on the path of the abnormal sequence. When it reaches the last buffer B_k, B_k clears its state. Then B_{k-1} will clear it state too and so on. Thus, at the end, we are sure that all the buffers in sequence have cleared their state. Note that if in the sequence $S_{B_k} = (id, B_{k-1}, B_1, ?, z)$ and $S_{B_1} = (id, B_k, B_2, ?, x)$ then we are sure that $z \neq x + 1$. In this case too, the processor having B_1 as a buffer will be able to detect such a situation and initiates the freeze cleaning phase as previously.

Note that when the sequence of a token circulation T is completely validated (all the buffers B_i part of it have the state $S_{B_i} = (id, ?, ?, V, ?)$) , no other token circulation can break it (use a buffer that is already part of T). Note also that the validation phase starts from the last buffer B_k such as either B_k is free or it contains a message that has to be forwarded to one buffer B_s that is already part of T. (i) if the state of B_s is corrupted (its state was set already in the initial configuration, B_s is part of an abnormal sequence), then when the confirm phase is initiated, and when the state of B_k is updated to the confirm phase ($S_{B_k} = (id, ?, ?, F, ?)$), we are sure that the state of B_s is not in the confirm phase (Recall that the confirm phase starts from the initiator and it passes by

[1] The complete proofs can be found in `http://arxiv.org/pdf/1107.6014`

all the buffers part of T. Since B_s is not part of it, it will never update its state to the Confirm phase). Thus, B_k clears its state, then B_{k-1} does the same and so on. Thus, in this case we are sure that T will be cleared. (ii) if the state of B_s is not corrupted, when the initiator initiates the confirm phase, we are sure that all the buffers part of it will updates their state to the confirm phase. Note that since the abnormal sequences eventually clear their state (refer to Lemma 1), The token circulations that validate all their sequence will be able to confirm it completely in a finite time.

In a case of a cycle, when the sequence of a token circulation T is completely confirmed, the message that is in the input buffer B of the processor p that detected the cycle (note that B is part of T) is copied to EXT_p. We show the following result where B is $IN_p(q)$:

Lemma 2. *If a valid message m is copied to EXT_p from $IN_p(q)$ in order to be copied later in $OUT_p(q')$, then when $S_{pq'o} = (id, ?, ?, E, ?)$, EXT_p is free.*

Proof Outline. The idea of the proof is as follows: when the message is copied to EXT_p from $IN_p(q)$, the processor p initiates the last phase of the token circulation (the Escort phase). Note that $IN_p(q)$ becomes a free buffer. We prove that there is a synchrony between the Token circulation and the forwarding algorithm such as the escort phase's token progresses on the system according to the token circulation sequence at the same time as the free slot. Thus, when the action that allows p to update the state of $OUT_p(q')$ to $S_{pq'o} = (id, ?, ?, E, ?)$ is enabled. $OUT_p(q')$ is free thus, either $R'13$ or $R'14$ is enabled as well. Thus, when p executes both actions (recall that in the case there is an action that is enabled in each algorithm (Algorithm 1 and Algorithm 2), they are both executed), Thus, EXT_p becomes free and $S_{pq'o} = (id, ?, ?, E, ?)$ and the lemma holds. □

We can now detect in some cases if the message in the extra buffer is invalid (it was in the initial configuration). Note that the algorithm deletes a message only in such cases (when we are sure that the message in the extra buffer is invalid), refer to Rules $R'15$ and $R'16$. Thus, we have the following theorem:

Theorem 1. *No valid message is deleted from the system unless it is delivered to its destination.*

We now show in Lemma 3 that the extra buffer of any processor p cannot be infinitely continuously busy (Recall that the extra buffer is used to solve the problem of deadlocks).

Lemma 3. *If the extra buffer of the processor p (EXT_p) contains a message, then this buffer becomes free after a finite time.*

Since every token circulation T that confirmed its sequence finishes its execution (recall that no other token circulation can break its sequence). We are sure that all the buffers part of T will be cleared in a finite time. On another hand, the abnormal sequences clear their state in a finite time and the extra buffer of each processor cannot be infinitely continuously busy thus, we can deduce the next lemma:

Lemma 4. *If there is a processor that wants to initiates a token circulation, it will be able to do it in a finite time.*

Since many token circulations can be executed in parallel on the system, we have to prove that at least one of them will be able to validate its path. Thus, the Lemmas 5 follows:

Lemma 5. *If there are some Token Circulations that are initiated then at least one of them will validate all its path.*

We can then deduce that at least one message will undergo a route change. Note that once the routing tables are stabilized, every new message is generated in a suitable buffer. So, it is clear 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 all the messages will be in a suitable buffer in a finite time.*

We have to show now that any processor can generate a message in a finite time and that no deadlock situation appears in the system. Lemma 6 ensures that all the messages are in suitable buffers in a finite time. Since the Token circulations are only used for route changes, then:

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

Note that the fair pointer mechanism cannot be disturbed anymore by the token circulations. Since our buffer graph is a DAG, the following lemma holds:

Lemma 8. *All the messages progress in the system.*

In the same manner, since the fair pointer mechanism cannot be disturbed. 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.*

We can deduce the following theorem:

Theorem 2. *Neither deadlock nor Starvation situation appears in the system.*

The color management (Function Choice(c)) ensures the next lemma :

Lemma 10. *The forwarding protocol never duplicates a valid message even if the routing tables are not yet stabilized.*

From Theorem 1 , no message is deleted unless it meets its destination. From Theorem 2, no deadlock situation appear and any message can be generated in a finite time. From Lemma 10, no valid message is duplicated. Hence, the following theorem holds:

Theorem 3. *The proposed solution (Algorithms* 1, 2 *and* 3*) is a snap-stabilizing message forwarding algorithm (satisfying SP) under a weakly fair daemon.*

4 Conclusion

In this paper, we presented the first snap-stabilizing message forwarding protocol on trees that uses a number of buffers per node being independent of any global parameter. Our protocol uses only 4 buffers per link and an extra one per node. This is a preliminary version to get a solution that tolerates topology changes provided that the topology remains a tree.

References

1. Cournier, A., Dubois, S., Lamani, A., Petit, F., Villain, V.: Snap-Stabilizing Linear Message Forwarding. In: Dolev, S., Cobb, J., Fischer, M., Yung, M. (eds.) SSS 2010. LNCS, vol. 6366, pp. 546–559. Springer, Heidelberg (2010)
2. Dolev, S.: Self-stabilization. MIT Press (2000)
3. Bui, A., Datta, A.K., Petit, F., Villain, V.: Snap-stabilization and PIF in tree networks. Distributed Computing 20, 3–19 (2007)
4. Duato, J.: A necessary and sufficient condition for deadlock-free routing in cut-through and store-and-forward networks. IEEE Trans. Parallel Distrib. Syst. 7, 841–854 (1996)
5. Merlin, P.M., Schweitzer, P.J.: Deadlock avoidance in store-and-forward networks. In: Jerusalem Conference on Information Technology, pp. 577–581 (1978)
6. Toueg, S.: Deadlock- and livelock-free packet switching networks. In: STOC, pp. 94–99 (1980)
7. Toueg, S., Ullman, J.D.: Deadlock-free packet switching networks. SIAM J. Comput. 10, 594–611 (1981)
8. Awerbuch, B., Patt-Shamir, B., Varghese, G.: Self-stabilizing end-to-end communication. Journal of High Speed Networks 5, 365–381 (1996)
9. 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 (1995)
10. 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)
11. 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)
12. Dijkstra, E.W.: Self-stabilizing systems in spite of distributed control. Commum. ACM 17(11), 643–644 (1974)
13. 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)
14. Merlin, P.M., Schweitzer, P.J.: Deadlock avoidance in store-and-forward networks. In: Jerusalem Conference on Information Technology, pp. 577–581 (1978)