Theoretical Computer Science 496 (2013) 89–112

Contents lists available at SciVerse ScienceDirect

Theoretical Computer Science

journal homepage: www.elsevier.com/locate/tcs

The snap-stabilizing message forwarding algorithm on tree topologies[☆]

Alain Cournier[a], Swan Dubois[b], Anissa Lamani[a,*], Franck Petit[b,**], Vincent Villain[a,**]

[a] MIS, Université de Picardie Jules Verne, Amiens, France

[b] LiP6/CNRS/INRIA-REGAL, Université Pierre et Marie Curie, Paris 6, France

A R T I C L E   I N F O

Keywords:
Message forwarding
Snap-stabilization
Self-stabilization
Fault tolerance

A B S T R A C T

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 process, where n is the number of processes in the network) making the solution not scalable or 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 Cournier et al. (2010) [1]. However the solution works only on a chain network. 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 Cournier et al. (2010) [1] but works on tree topologies.

1. Introduction

The ability of a distributed system to tolerate faults is a crucial issue in most distributed systems. Self-stabilization [2] is a lightweight type of fault-tolerance that ensures starting from any arbitrary state, a legitimate state is eventually reached. Snap-stabilization [3] offers a stronger property than self-stabilization since it guarantees that any computation started after faults cease immediately satisfies the expected specification. In other words, a snap-stabilizing algorithm is also a self-stabilizing algorithm that stabilizes in 0 steps and is optimal in terms of the worst-case stabilization time.

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 includes 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 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 [4–8]. In [9,10,8] self-stabilizing solutions were proposed. Both solutions deal with network dynamic, *i.e.*, systems in which links can be added or removed.

[☆] A preliminary version of this manuscript appeared in ICDCN 2012. 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.

[*] Corresponding author. Tel.: +33 144278767.

[**] Corresponding authors.

E-mail addresses: alain.cournier@u-picardie.fr (A. Cournier), swan.dubois@lip6.fr (S. Dubois), anissa.lamani@u-picardie.fr (A. Lamani), franck.petit@lip6.fr (F. Petit), vincent.villain@u-picardie.fr (V. Villain).

0304-3975/$ – see front matter © 2013 Elsevier B.V. All rights reserved.
http://dx.doi.org/10.1016/j.tcs.2013.04.012

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 [11,12,1]. In [11], the problem was solved using n buffers per node (where n denotes the number of processes in the network). The number of buffers was reduced in [12] 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 will explain in Section 3.1, why the solution proposed in [1] cannot be used on tree 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 process plus one internal buffer, thus, $2\delta + 1$ buffers by process, where δ is the degree of the process in the system.

Road Map. The rest of the paper is organized as follows: our model is presented in Section 2. In Section 3, we provide our snap-stabilizing solution for the message forwarding problem, followed by its correctness proofs. 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 (processes) and E is the set of bidirectional communication links. Each process has a unique *id*. Two processes p and q are said to be neighbors if and only if there is a communication link (p, q) between the two processes. Note that, every process 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 processes.

Computational model. In this paper we consider the classical local shared memory model introduced by Dijkstra [13] known as the state model. In this model, communications between neighbors are modeled by direct reading of variables instead of exchange of messages. The program of every process consists in a set of shared variables (henceforth referred to as variable) and a finite number of actions. Each process can write in its own variables and read its own variables and those of its neighbors. Each action is constituted as follows:

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

The guard of an action is a Boolean expression involving the variables of p and its neighbors. 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 process is defined by the value of its variables. The state of a system is the product of the states of all processes. The local state refers to the state of a process and the global state (configuration) to the state of the system.

Let us refer by C to the set of all the 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. Process 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 $\to$, on C. An execution of a protocol P is a maximal sequence of configurations $e = y_0 y_1 \ldots y_i y_{i+1} \ldots$ such that, $\forall\, i \geq 0$, $y_i \to y_{i+1}$ (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 process evaluates its guard; (ii) One or more enabled processes execute their enabled actions. When the two phases are done, the next step begins. This execution model is known as the *distributed daemon* [14]. We assume that the daemon is *weakly fair*, meaning that if a process 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 process 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 [15] 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 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 to identify the neighbor to which p must forward a given message m in order for m to reach its destination d. In the following we assume that there is no message in the system whose destination is not in the system.

Note that the problem of having messages with unreachable destinations can be treated separately with two different approaches depending on assumptions made on the system. Assume first that every process knows the identity of all the other processes. This is the case for instance when each process knows n, the actual number of processes in the system and its ID belongs the range $0 < id <= n - 1$. Then, every message with a destination which is not in $[0, n - 1]$ can be deleted. This approach works on static systems only. Assume now that no process knows the set of identities in the system. In this case, when the destination of a given message m does not appear in the routing table, then a snap-stabilizing Propagation of Information with Feedback (PIF) wave [3] can be initialized to check whether the destination of m exists or not. Although several PIF waves can be initialized, we can add only one buffer per process by providing the priority to the PIF wave with the smallest identity. If the destination does not exist, then the message m is deleted. This second approach works in both static and dynamic systems. However, in the latter case, additional assumptions on the speed of the topology changes must be taken in account carefully.

Observe that the solution in [1] cannot be applied directly on systems that have a tree topology. Indeed, in [1], since the network topology is a chain, the authors proposed a buffer graph such that in the case where there is a cycle, the leaf processes are sure to be involved in such a cycle. Thus, only the leaf processes were endowed with an extra buffer and were able to initialize a cycle resolution. When the topology of the system is a tree, this solution cannot be applied. Indeed, some messages will be forced to be forwarded in the wrong direction during a cycle resolution. Thus, we can no more guarantee that all the messages will be in the right direction and delivered to their destination.

Before detailing our solution, let us define the buffer graph used in our solution:

Let $\delta(p)$ be the degree of process p in the tree structure and let N_p be the set of processes that are neighbor of p. Each process 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 process 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 process for the message in order to reach its destination.

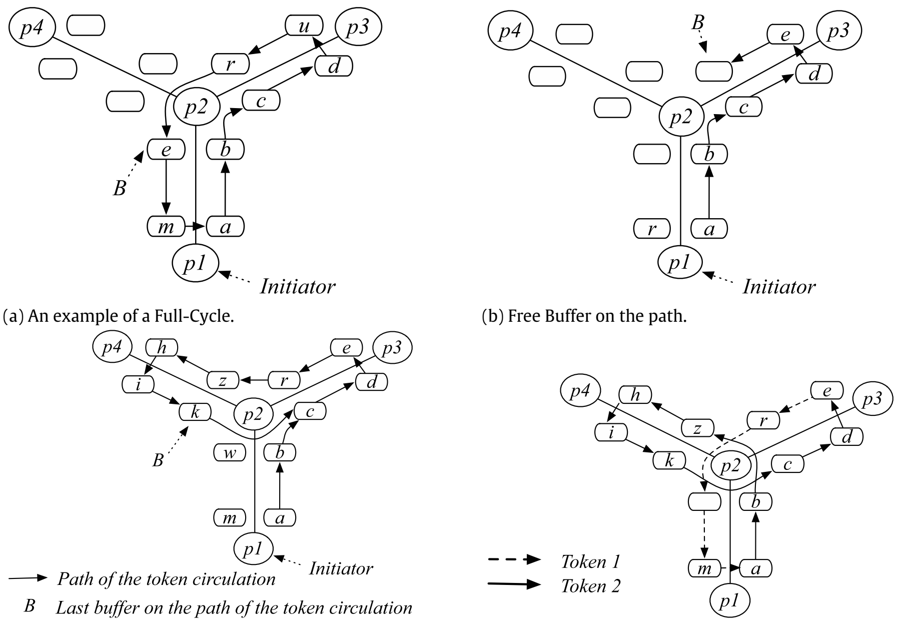
The overall idea of the algorithm is the following: when a process wants to generate a message, it consults the routing tables to determine the next neighbor by which the message will transit in order to reach its destination. Once the message is in the system, it is routed according to the routing tables: 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 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 (Suppose that there is a message m in $OUT_p(q)$, when m is sent to q, $IN_q(p) = m$. Observe that $OUT_p(q) = IN_q(p)$. Since the message of $OUT_p(q)$ have been sent, $OUT_p(q)$ is considered free). In the opposite case, a buffer is said to be busy. The transmission of messages produces the filling and the


p4
p3
p2
p1
u
r
d
c
e
b
m
a
B
Initiator
h
i
z
k
w
Path of the token circulation
B Last buffer on the path of the token circulation
Token 1
Token 2

(a) An example of a Full-Cycle.
(b) Free Buffer on the path.
(c) An example of Sub-Cycle.
(d) Token circulations deadlocked.

Fig. 1. Different configurations related to token circulations.

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 following, 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 subgraph 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 $nb(m, IN_p(q)) = OUT_p(q')$ then m is using the resource (buffer) $IN_p(q)$ and it is asking for another resource which is the output buffer $OUT_p(q')$. In the following we will refer to this subgraph 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 Fig. 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: $nb(m, IN_p(q)) = OUT_p(q) \wedge nb(m', IN_{p'}(q')) = OUT_{p'}(q')$ (Messages m and d in Fig. 1(a)). Since in this chapter, we consider a distributed system, it is impossible for a process 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 m in its input buffer ($IN_p(q)$) that has to be sent to output buffer of the same link ($OUT_p(q)$). In order to verify that, p initiates a token circulation that follows the active buffer graph starting from its output buffer $OUT_p(q)$. By doing so, the token circulation either finds a free buffer (refer to Fig. 1(b)) or detects a cycle. Two kinds of cycle can be detected:

(i) a *Full-Cycle* that involves the output buffer of the initiator ($OUT_{p1}(p2)$ in Fig. 1(a))
(ii) a *Lasso that* does not involve the output of the initiator ($OUT_{p1}(p2)$ in Fig. 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 cycle that has been detected: (i) The case of a Full-Cycle: p is the one that detects the cycle (p_1 in Fig. 1(a)). The aim is to release $OUT_p(q)$. (ii). The case of a Lasso: the process containing the last buffer B that can be reached by the token is the one that detects the cycle (process p_2 in Fig. 1, (c)). Observe that B is a non empty input buffer. Our aim, in this case, is to release the output buffer B' by which the message in B has to transit in order to meet its destination ($OUT_{p_2}(p_3)$ in Fig. 1(c)). Note that B' is part of the path of the token circulation.

In both cases (i) (full-cycle) and (ii) (lasso), the process 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 process releases its input buffer. The idea is to move messages on the token circulation path to make the free slot that was created on the input buffer move. This ensures that the corresponding output buffer will be empty 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. One cycle resolution is then said to be performed.

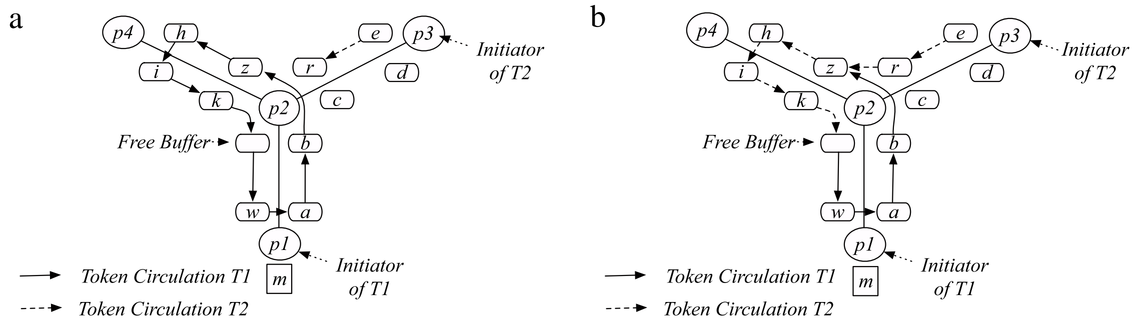


Fig. 2. Multi-token management.

Many token circulations can be executed in parallel. To avoid deadlock situations between the different token circulations (refer to Fig. 1(d)), each token circulation carries the identifier of its initiator. The token circulation with an identifier *id* can take a buffer of another token circulation having the identifier id' if $id < id'$. 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 then be lost. For instance, in Fig. 2, one can observe that the free slot that was produced by $T1$ is taken away by $T2$. 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 as the extra buffer of p_4 becomes non empty 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 (the messages in the extra buffers cannot be erased so no empty slot can be created) . Thus, the solution is 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. 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 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 a buffer B_i has the token with the value x, then when the next buffer B_{i+1} receives the token it sets its value to $x + 1$. Thus, we are sure that in the case where there is a cycle, there will be two consecutive buffers B and B' in the path of the token circulation, having respectively x and x' as a value such that $x \neq x' + 1$. This kind of situation can be then detected.

Instance of an execution. In this paragraph, we refer to Figs. 3 and 4 to describe a typical execution of our scheme. Assume that the next destination of the message m that is in $IN_{p1}(p2)$ is $p2$ (refer to Fig. 3, Case (a)). Clearly, if the routing tables are correct and stabilized, no such behavior can happen in the system. $p1$ in this case, can suspect the presence of a cycle, to be sure of that, $p1$ initiates a token circulation that follows the destination of the messages that met the token. In Fig. 3, Case (b), the next destination of the message a in $OUT_{p1}(p2)$ is $IN_{p2}(p1)$, thus the token circulation will be sent to from $OUT_{p1}(p2)$ to $IN_{p2}(p1)$.

Similarly as $p1$, $p2$ knows by consulting the routing tables, that the next destination of the message b in $IN_{p2}(p1)$ is $OUT_{p2}(p3)$, thus, the token will be sent from $IN_{p2}(p1)$ to $OUT_{p2}(p3)$ and so on. The token progresses in the system until it reaches $IN_{p2}(p4)$. When $p2$ consults the routing tables, it knows that the next buffer of the message k is $OUT_{p2}(p3)$. However, $p2$ notices also that the token has already passed by this buffer. Thus, $p2$ sends back the token in order to notify the other processes that a cycle has been detected and in the same time, to validate the path of the first token circulation (Fig. 3, Case (c)). The second circulation of the token follows exactly the path taken by the first one (Fig. 3, Case (d) and Fig. 4, Case (e)). When the token arrives to the initiator ($p1$). $p1$ sends back once again the token to confirm that the path of the token circulation has been validated and all the other processes on the path has been notified. The third circulation of the token (Phase Confirm) follows the path defined by the first token (Fig. 4, Case (f)). When $p2$ receives the token for the second time from $p4$, it knows that all the path of the token circulation has been validated and confirmed. $p2$ now can copy the message k in its internal buffer to create a free slot in $IN_{p2}(p4)$ (Fig. 4, Case (g)) and in the same time, $p2$ initiates another token circulation to accompany the free slot during its move on the path of the confirmed token circulation (refer to Fig. 4, Cases (i) and (j)). By doing so, the free slot arrives to $OUT_{p2}(p3)$ that becomes empty. Thus, $p2$ can copy the message k from its internal buffer to $OUT_{p2}(p3)$ (Fig. 4, Case (k)). $p2$ cleans the state of $IN_{p2}(p_4)$ (Fig. 4, Case (l)). By doing so, the states of buffers on the processes are cleaned progressively on the path of the token circulation.

To simplify the explanation and the description of our solution, we will suppose that the actions are enabled and executed on the buffers of the graph instead of the processes. By doing so, our graph becomes dynamic *i.e.*, the connections between buffers are defined according to the messages that are in the buffers and the state of the routing tables at time t. For instance if there is a message in Buffer A such that the next destination of this message is Buffer B then there is a connection between Buffer A and Buffer B. The connections are also defined by the token circulation in a cycle resolution. For instance when

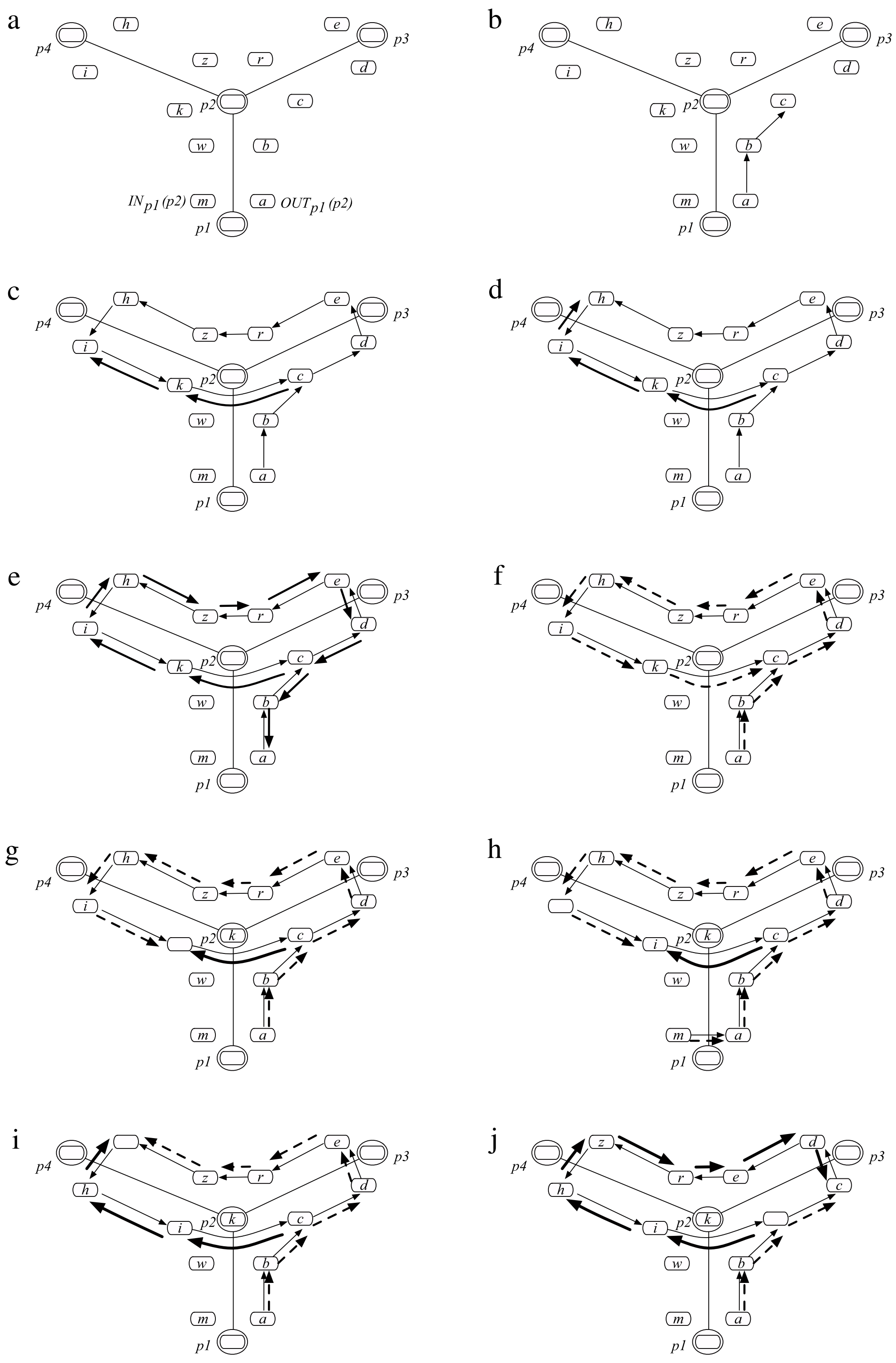


Fig. 3. An example showing an execution of our scheme, Part 1.

the path of the token circulation is defined, the messages can only be transmitted on the buffers that are part of this path (regardless of the routing tables).

In the following, we assume that the correction rules have the priority over the rules of our protocol *i.e.*, in the case where there is a correction rule that is enabled on a given buffer at the same time as a rule from our protocol, then the correction rule is the only one executed.

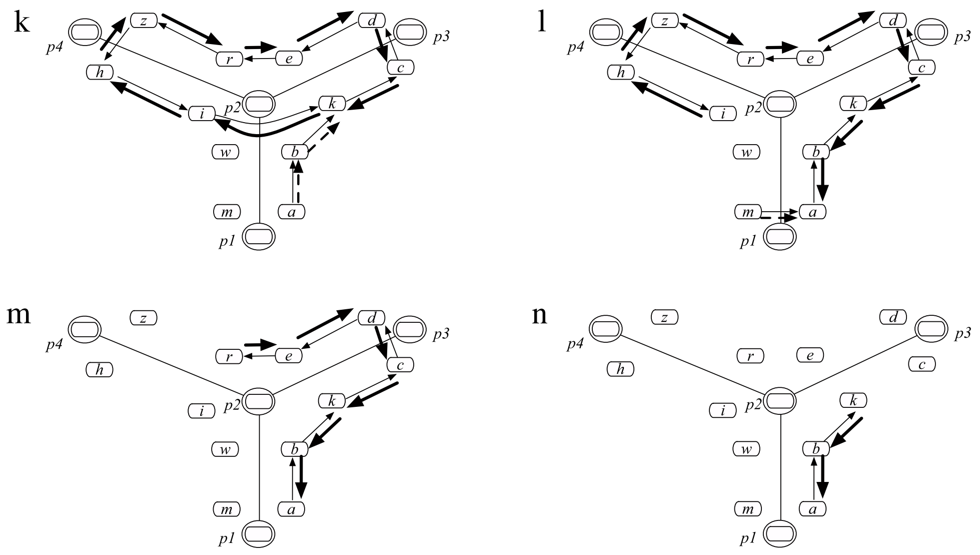


Fig. 4. An example showing an execution of our scheme, Part 2.

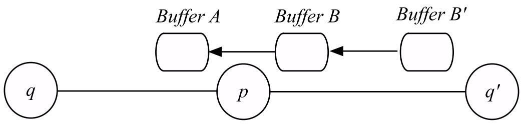


Fig. 5. Vision at distance 2.

On the other hand, we will allow the buffers, in our solution, to see the state of the buffers that are at distance 2 *i.e.*, the buffers that are neighbors of the neighboring buffers. This clearly bends our model defined previously. However, since the buffer graph is defined on the processes of the graph, and since in reality, the algorithm is executed by the processes instead of the buffers, the vision at distance 2 on buffers corresponds to the case when the two neighboring buffers are part of the same process. For instance, in Fig. 5, Buffer A is able to read the state of both buffers B and B′. If we consider the buffers of the graph instead of the processes, Buffer A is able to see at distance 2. However, when considering the graph of the processes, process p has only a vision at distance 1. Thus, the vision at distance 2 when considering the buffer graph corresponds to a vision at distance 1 on the process graph. In the same manner, in some cases, some rules of the proposed algorithm allow the execution of two actions on respectively two different buffers. This also bends our model, however, this can only happen when the two buffers on which an action is executed are part of the same process (Buffer A and Buffer B on the process p in Fig. 5). If there are actions that are enabled on many buffers of a given process p, then p executes them in the same step.

3.2. Formal description

We present in the following, respectively the formal description of the message forwarding algorithm and the token circulation algorithm.

3.2.1. Message forwarding algorithm

Let us first present the variables and predicates used in the description of our message forwarding algorithm, we then provide the formal description of our Solution in Algorithm 1.

Variables

- EXT_p: The Extra buffer of the process 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).
- Buffer(A): denotes the content of Buffer A.
- id-A: denotes the identity of Buffer A.
- Destination(m): refers to the destination of the message m.
- Process(A): refers to the process owning the buffer A.
- State-Buffer(A)=(id-Token(A), parent(A), child(A), phase(A), level(A)): refers to the state of the buffer A. id-Token(A) refers to the identity of the process that initiates the token circulation. The parameter parent(A) is a pointer towards the

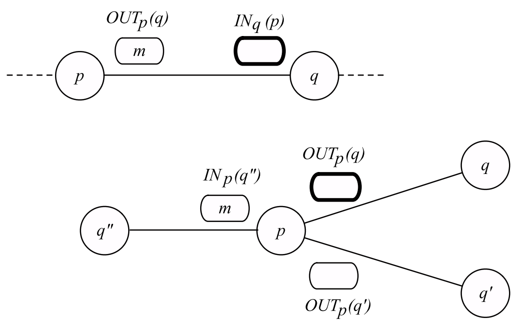


Fig. 6. Instance of a configuration.

buffer from which A has received the token for the first time. The parameter child(A) is a pointer that refers to the buffer to which A has sent the token for the first time. The parameter phase(A) $\in \{S, V, F, E, C, G\}$ defines the phase of the token circulation (Search, Validation, Confirm, Escort, Clean, Freeze). Finally, level(A) is an integer that refers to the level of the buffer in the path of the token circulation. It is set to $x + 1$ if the token was received for the first time from Buffer B which is at level x. This will be used in order to avoid cycles in the path of the token circulation.

- Token-Request(A,B): Boolean that allows the communication between the forwarding algorithm and the token circulation algorithm. It is set to true by the message forwarding algorithm when Buffer A needs to initiate a token circulation towards Buffer B (Buffer A is in this case an input Buffer).
- Message-to-Generate(m): Boolean, allows the communication with the higher layer, it is set to true by the application when a message has to be sent and to false by the forwarding protocol once the message is generated.

Actions

- Consume(m,A): delivers the message m to the higher layer.

Macros

- Next-Buffer(A,m): refers to the next buffer of the message m, after Buffer A, in order to reach its destination. In order to explain this macro, let us first define the following Predicate:
  Same-Link(A,B): process(A)=process(B)=$p \wedge \exists\, q$, Buffer(A)=$IN_p(q) \wedge$ Buffer(B)= $OUT_p(q)$. This predicate indicates that Buffer(A) and Buffer(B) are part of the same process p on the link (p, q). Buffer(A) is an input buffer whereas Buffer(B) is an output buffer.
  Next-Buffer(A,m) is determined by both the routing tables and the buffer-graph. For instance (refer to Fig. 6), in the case where the destination of the message m is different from p then, if A is an output buffer ($OUT_p(q)$ in Fig. 6) then the next buffer of m is the input buffer connected to A ($IN_q(p)$ in Fig. 6). In the case where A is an input buffer ($IN_p(q")$ in Fig. 6) then two cases are possible as follows:
  - Same-Link(A,B) $\wedge$ Token-State(B) $\wedge$ level(B)=0. In this case, regardless of the routing tables, Next-Buffer(A,m)=B (the token circulation has the priority over the routing tables
  - Same-Link(A,B) $\wedge$ (Token-State(B)$\Rightarrow$ level(B)$\neq 0$). Then, the next buffer is the output buffer of the process p connected to q ($OUT_p(q)$) where q is the next process by which the message has to transit in order to reach its destination (q is given by the routing table).
  Predicates
  - Edible(A): Buffer(A)= $m\wedge$ Destination(m)=process(A). This predicate indicates that the destination of Message m, that is in Buffer A, is process(A).
  - Free(A): Buffer(A)= $\epsilon\vee$ (Buffer(Buffer(A)= m $\wedge$ Next-Buffer(A,m)=B $\wedge$ Buffer(B)= m). This predicate indicates whether buffer A is free or not. Buffer A is said to be free if the predicate is verified, otherwise it is said to be occupied. From the description of Free(A), we can notice that Buffer A is free if it does not contain any message (Buffer(A)= ϵ), or when the message m in Buffer A is exactly the same message as in Next-Buffer(A,m) (the message m has been sent to Next-Buffer(A,m)).
  - No-Duplication(m,A): Buffer(A)=$m \wedge \forall B$, Next-Buffer(B,m')=A $\Rightarrow m' \neq m$. This indicates that there is no other copy of the message m in Buffer B such that Next-Buffer(B,m')=A. This predicate will be used to ensure that there is at most two copies of a valid message in the buffer graph (refer to Fig. 7).
  - Message-to-send(B,m,A): Buffer(B)=$m \wedge$ Next-Buffer(B,m)=A. This predicate indicates that the message m, in Buffer B, needs to be sent to Buffer A.
  - Clean-State(A): State-Buffer(A)=(-1,$\perp$,$\perp$,C,-1). This predicate indicates that Buffer A is not part of any token circulation (its state is clean).

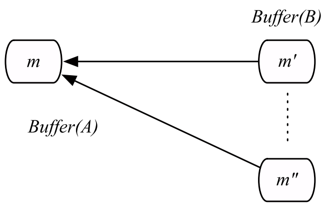


Fig. 7. No-Duplication(m,A)=true.

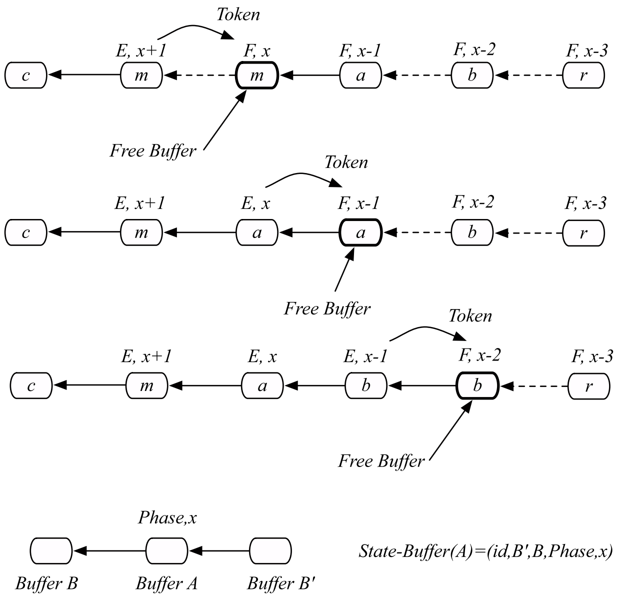


Fig. 8. Token path.

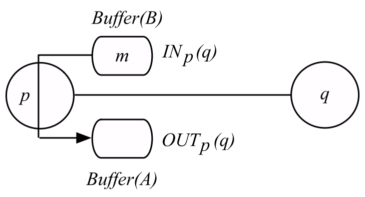


Fig. 9. Uturn of the message m.

– Incorrect-Clean-State(A): [id-Token=-1 $\vee$ level(A)=-1 $\vee$ phase(A)=C $\vee$ parent(A)=child(A)] $\wedge$ $\neg$ Clean-State(A). This predicate indicates that Buffer A is not part of any token circulation. However, Clean-State(A) is not verified.
– Token-State(A): State-Buffer(A)$\neq$ $(-1, \bot, \bot, \text{C}, -1)$ $\wedge$ $\neg$ Incorrect-Clean-State(A). This predicate indicates that A is part of a token circulation.
– No-Token(A): Clean-State(A) $\wedge$ ($\forall B$, Token-State(B) $\Rightarrow$ child(B) $\neq$ A) $\wedge$ (Same-Link(A,B$'$) $\Rightarrow$ Level(B$'$) $\neq$ 0). This predicate indicates that Buffer A is not part of any token circulation and there exists no other buffer B$'$ that has Buffer A as a child. In addition, in the case where A is an input buffer of a given process p of the link (p, q) ($IN_p(q)$), then $OUT_p(q)$ (Buffer B$'$) did not initiate any token circulation (Token-State(B$'$) $\Rightarrow$ level(B$'$)$\neq$ 0).
– Token-Path(B,A): Token-State(A) $\wedge$ State(A)=(id,B$'$,B,F,x) $\wedge$ Token-State(B$'$) $\wedge$ State(B$'$)=(id,?,A,F,$x-1$) $\wedge$ Token-State(B) $\wedge$ State(B)=(id,A,?,E,$x+1$). Regardless of the routing tables, the message in Buffer B needs to be sent to Buffer A (this is defined by the token circulation). This happens when a free slot is being dragged in a cycle resolution). The transmission is done at the same time as the evolution of the Escort phase of the token circulation (refer to Fig. 8).
– Uturn(m,B,A): Buffer(B)=m $\wedge$ Next-Buffer(m)=A $\wedge$ Same-Link(A,B). This predicate indicates that there is a message m in Buffer(B) that needs to be forwarded to Buffer(A) such that Buffer(B) and Buffer(A) are two buffers of the same link (refer to Fig. 9).
– Forced-Uturn(m,B,A): Same-Link(B,A) $\wedge$ Token-State(A) $\wedge$ Clean-State(B) $\wedge$ State-Buffer(A)=(id,A,child(A),F,0) $\wedge$ State-Buffer(child(A))=(id,A,?,E,1). This predicate indicates that a U-Turn can be performed since the token circulation has found a free buffer.

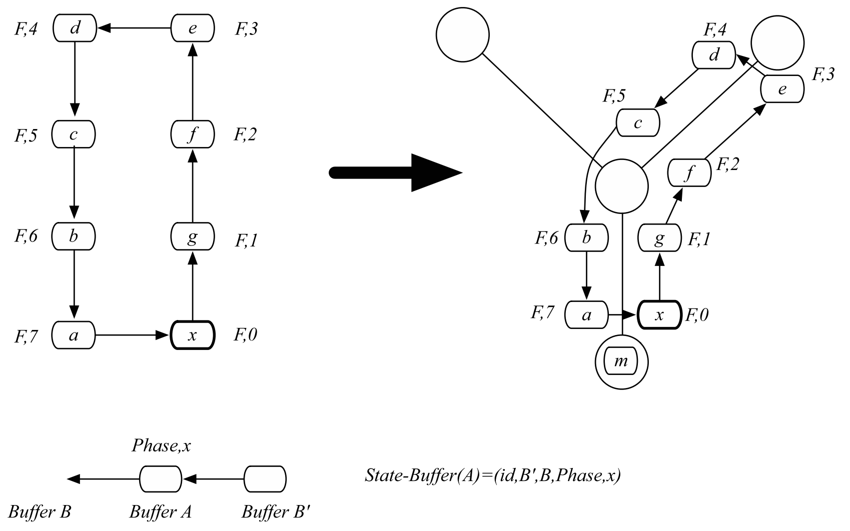


Fig. 10. Confirmation of a Full-Cycle.

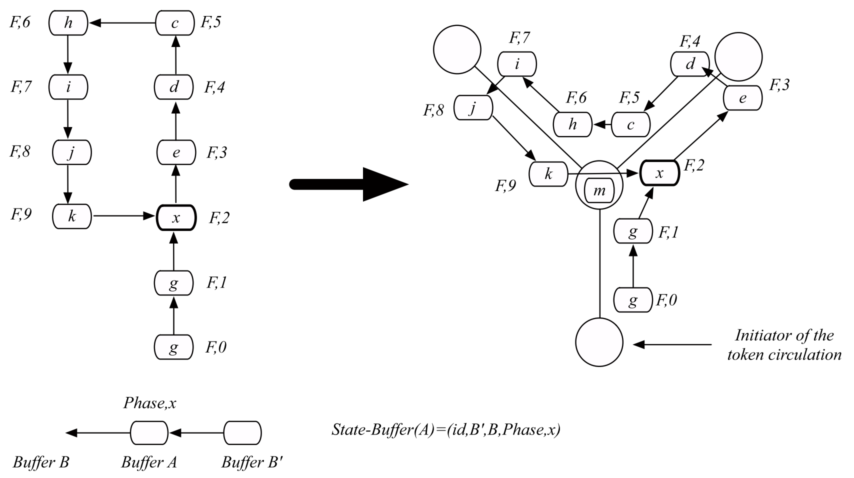


Fig. 11. Confirmation of a Lasso.

- Synchro[Action1(A), Action2(B)]. This indicates that Action1 and Action2 are executed at the same time on respectively Buffer A and Buffer B (notice that this happens only when both buffers A and B are part of the same process *i.e.*, process(A)=process(B)).
- Full-Cycle(A): Token-State(A) $\wedge$ Token-State(child(A)) $\wedge$ $level(A) \geq 3$ $\wedge$ level(child(A))=0 $\wedge$ id-Token(A)=id-Token(child(A)) $\wedge$ parent(child(A))=child(A). This predicates indicates that Buffer A is the last buffer of a token circulation that has detected a full-cycle.
- Lasso(A): Token-State(A) $\wedge$ Token-State(child(A)) $\wedge$ id-Token(A)=id-Token(child(A)) $\wedge$ parent(child(A))$\neq$ A $\wedge$ level(child(A))$\neq$ level(A)+1. This predicate indicates that Buffer A is the last buffer of a token circulation that detected a lasso.
- Confirm-Cycle(A): [Full-Cycle(A) $\vee$ Lasso(A)] $\wedge$ phase(A)=F $\wedge$ phase(child(A))=F. This predicate indicates that all the buffers of the token circulation has updated their phase to F (Figs. 10 and 11).
- Filiation(A,B): Token-State(A) $\wedge$ Token-State(B) $\wedge$ child(A)=B $\wedge$ parent(B)=A $\wedge$ id-Token(A)=id-Token(B). This predicate indicates that Buffer A is the parent of Buffer B and that buffer B is the child of Buffer A for the same token circulation.
- Normal-Filiation(A,B): Filiation(A,B) $\wedge$ ($A \neq B \Rightarrow$ level(B)=level(A)+1). This predicate indicates that Buffer A is the parent of Buffer B, Buffer B is the child of A and the levels of both Buffer A and B are correct.
- Last-Buffer(A): Token-State(A) $\wedge$ [child(A)=A $\vee$ Full-Cycle(A) $\vee$ Lasso(A)]. This indicates that Buffer A is the last buffer of a token circulation.
- First-Buffer(A): Token-State(A) $\wedge$ parent(A)=A $\wedge$ level(A)=0 $\wedge$ id-Token(A)=id-A. This predicate indicates that Buffer A is the first buffer of a token circulation (more precisely, it is the first buffer of a normal token circulation defined in Definition 2).

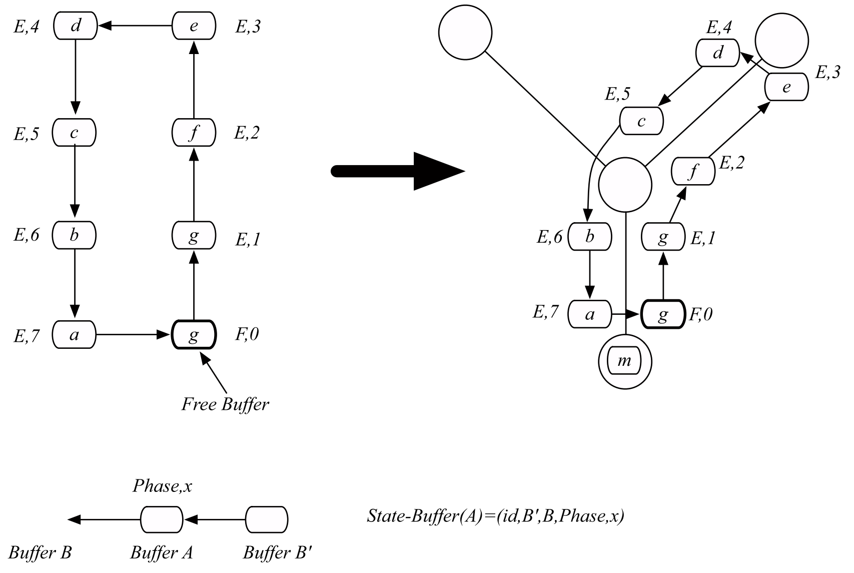


Fig. 12. End-Token in the case of a Full-Cycle.

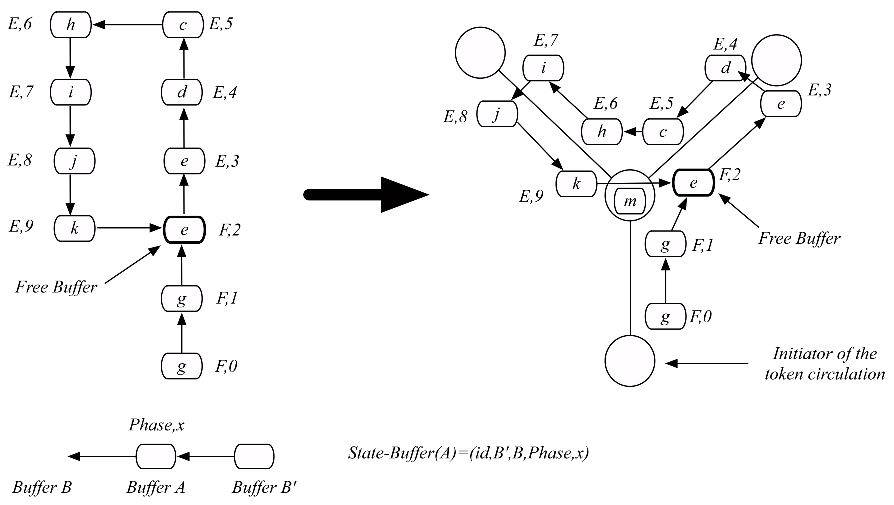


Fig. 13. End-Token in the case of a Lasso.

– End-Token(A): [∃ B, [Full-Cycle(B) ∨ Lasso(B)] ∧ child(B)=A ∧ phase(B)=E ∧ phase(A)=F ∧ phase(child(A))=E] ∨ [First-Buffer(A) ∧ Normal-Filiation(A,child(A)) ∧ phase(A)=F ∧ phase(child(A))=E] (Figs. 12 and 13).
This indicates the end of the Escort phase of the token circulation.

We define a round robin pointer on the extra buffer of each process. The extra buffer is a memory shared by a subset of buffers. All such buffers have the ability to read and write in the extra buffer. A given buffer A is allowed to write in the extra buffer ($EXT_{process(A)}$) if and only if the round robin pointer is on A as shown in Fig. 14. The round robin pointer points only towards an input buffer (for instance A) that is ready to use the extra buffer (Confirm-Cycle(A) is satisfied). If there exists no such buffers, then the round robin pointer is set to ⊥.

In the case where a given input buffer A is involved in a cycle, its message *m* is copied in the extra buffer to release a free slot. Let us refer to the next buffer of *m* by B (note that Buffer B is an output buffer). The free slot is dragged on the path of the cycle (defined by the token circulation) until it reaches Buffer B. Buffer B then, copies the message *m* and deletes it from $EXT_{process(A)}$.

Remark 1. A transmission between buffers, during a normal behavior, is performed by executing Rule R(F)3. However, in a case of a transmission from an input buffer ($IN_p(q)$) to an output buffer ($OUT_p(q')$) of a process p then, when the message is copied in ($OUT_p(q')$), p copies at the same time the message in $OUT_q(p)$ in $IN_p(q)$ (refer to Fig. 15). This is done to make sure that there exist at most two copies of a valid message in the buffer graph.

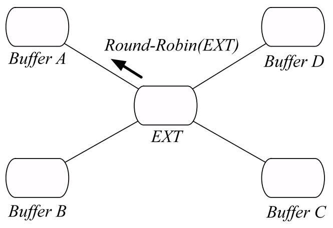


Fig. 14. Round-Robin pointer on the extra buffer EXT.

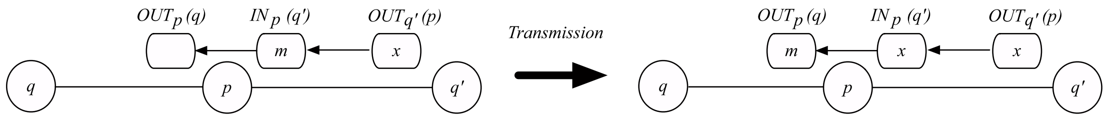


Fig. 15. Internal transmission.

Algorithm 1 Message Forwarding Protocol

During normal behavior

- **Message generation (output buffer)**
  - **R(F)1**: Message-to-Generate(m) $\wedge$ Buffer(m)=A $\wedge$ Free(A) $\wedge$ NO-Token(A) $\rightarrow$ Buffer(A):=m, Message-to-Generate(m):=false.
- **Message consumption (input buffer)**
  - **R(F)2**: Edible(A) $\wedge$ No-Duplication(m,A) $\rightarrow$ Consume(m,A), Buffer(A):=ϵ.
- **Transmission of the message** ***m*** **(input/output buffer)**
  - **R(F)3**: Message-to-send(B,m,A) $\wedge$ Free(A) $\wedge$ No-Duplication(m,B) $\wedge$ $\neg$ Uturn(m) $\wedge$ No-Token(A) $\wedge$ No-Token(B) $\rightarrow$ Buffer(A):=m.
- **Erasing the message** ***m*** **after its transmission (input/output buffer)**
  - **R(F)4**: Buffer(A)= m $\wedge$ Next-Buffer(A,m)=B $\wedge$ Buffer(B)=Buffer(A) $\wedge$ No-Duplication(m,A) $\rightarrow$ Buffer(A):=Choice-Buffer(A).

During abnormal behavior

- **Transmission of the message** ***m*** **(input/output buffer)**
  - **R(F)5**: Buffer(parent(A))=m $\wedge$ Token-Path(B,A) $\wedge$ Free(A) $\rightarrow$ Buffer(A):=m.
- **U-turn (input to output buffer)**
  - **R(F)6**: Uturn(m,B,A) $\wedge$ Clean-State(A) $\wedge$ Clean-State(B) $\wedge$ Free(A) $\wedge$ No-Duplication(B,m) $\rightarrow$ Synchro[Buffer(A):=m; Buffer(B):=ϵ].
  - **R(F)7**: Forced-Uturn(m,B,A) $\wedge$ Free(A) $\rightarrow$ Synchro[Buffer(A):=m; Buffer(B):=ϵ].
- **Token-Circulation-Request (input buffer)**
  - **R(F)8**: Uturn(m,B,A) $\wedge$ $\neg$ Free(A) $\wedge$ No-Duplication(B,m) $\wedge$ Free($EXT_{process(A)}$) $\wedge$ No-Token(B) $\rightarrow$ Token(A,Next-Buffer(A,m)):=true.
- **Free-Slot-Creation (input buffer)**
  - **R(F)9**: Confirm-Cycle(A) $\wedge$ process(A)=p $\wedge$ Free(EXT_p) $\wedge$ Round-Robin(EXT_p)=A $\rightarrow$ $EXT_p := m$, Buffer(A):=Buffer(parent(A)).
- **Delayed-U-turn (output buffer)**
  - **R(F)10**: End-Token(A) $\wedge$ Free(A) $\wedge$ process(A)=p $\wedge$ $\neg$ Free(EXT_p) $\wedge$ child(Round-Robin(EXT_p))=A $\rightarrow$ Buffer(A):=EXT_p, $EXT_p := \epsilon$.
- **Invalid message suppression without cycles (output buffer)**
  - **R(F)11**: $\forall$ A, process(A)=p, $\neg$ Free(EXT_p) $\wedge$ [Token(Round-Robin(EXT_p))=$\perp$ $\vee$ phase(Round-Robin(EXT_p))$\neq$ E)] $\rightarrow$ $EXT_P := \epsilon$.
- **Invalid message suppression in the case of a cycle (output buffer)**
  - **R(F)12**: $\neg$ Free(EXT_p) $\wedge$ End-Token(A) $\wedge$ $\neg$ Free(A) $\rightarrow$ $EXT_p := \epsilon$.
- **Cancellation of unnecessary token circulation requests (input buffer)**
  - **R(F)13**: Token-Request(A,B) $\wedge$ [Free(A) $\vee$ $\neg$ Uturn(m)] $\rightarrow$ Token-Request(A,B):=false.

Remark 2. In the description of our algorithm, Rules R(F)3, R(F)6 and R(F)8 allow Buffer A to read the state of Buffer B′ that is at distance 2 *i.e.*, Buffer B′ is a neighbor of Buffer B that is neighbor of A (refer to Fig. 16). As said previously, in reality this can happen when both Buffers A and B are part of the same process *p*. This is assumed when the output buffer of *p* is considered to avoid the duplications of messages. For instance in Fig. 16, Buffer A is allowed to receive the message *m* that is in Buffer B if and only if the message in Buffer B′ is different from *m*.

Observe that in the case where an input buffer is considered (Buffer A is an input buffer, refer to Fig. 17), then A copies the message in B regardless of the message being in Buffer B′. Since *m* is valid then, when *m* was copied in Buffer B, it has been removed from Buffer B′ according to Remark 1. Thus, in the case where there is a copy of the message *m* in the three buffers A, B and B′, that means that *m* in an invalid message (it was in the system in the initial configuration).

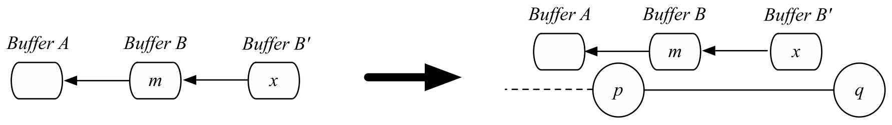


Fig. 16. Vision 2.

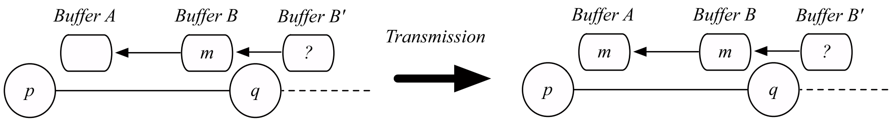


Fig. 17. Vision 1.

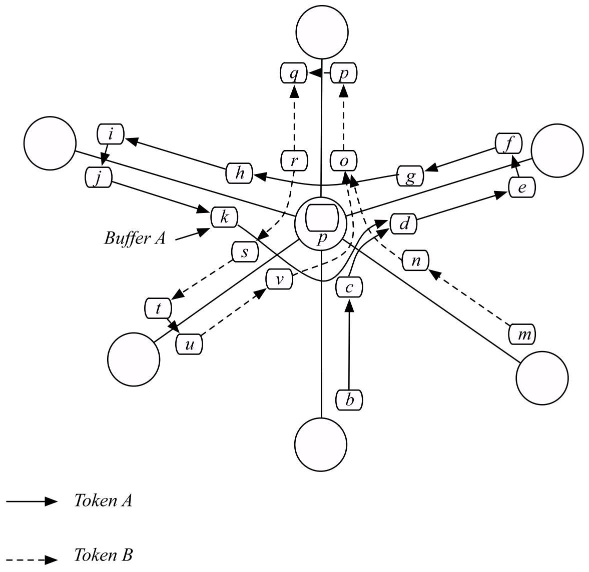


Fig. 18. Instance of a configuration.

3.2.2. *Token circulation algorithm*

Recall that the proposed token circulation comprises two round trips for the token in addition of the cleaning phase. The first round trip corresponds to respectively Search and Validation phases, while the second one corresponds to Confirm and Escort phases. In the following the 5 phases of the token circulation:

- Search Phase. The aim of this phase is to determine a cycle or to find a free buffer. The path of this token circulation is maintained in each buffer *i.e.*, each buffer that has received the token updates its state to save the identity of the token it has received and keep pointers to the buffer from which its has received the token and to which it has sent it.
- Validation Phase. It aims to validate all the path determined by the first phase (Search) of the token circulation. The token is sent back by the last buffer that has received the token and follows the reverse path of the Search phase. Upon receiving the token, each buffer updates its phase to the validation phase.
- Confirm Phase. The aim of this phase is to confirm all the path of the token circulation. When the token of the validation phase is received by the buffer that initiated the token circulation, this latter sends the token one again to confirm the path determined by the first phase. This is done to be sure that in the case where a cycle is detected then the cycle is valid.
- Escort Phase. This phase aims to escort the free slot that has been found or that has been created (in a cycle resolution) to make sure that no message generation is performed on the free slot (recall that this free slot will be used to allow a buffer to perform a u-turn).
- Cleaning Phase. The aim of this phase is to clean the path of the token circulation. This phase is initiated by the last buffer of the token circulation. By receiving the token, each buffer cleans its state.

The round-robin pointer on each extra buffer of the system aims, as explained earlier, to determine the next buffer to use the extra buffer if needed. For instance, in Fig. 18, Both Buffer A and Buffer B need to use the extra buffer to re-root their respective message and initialize the Escort phase. Buffer A will be the one to do so if Round-Robin(EXT_p)=A. Thus, Buffer A copies its message k in EXT_p and initializes the Escort phase. On another hand, the next buffer of m can also be determined thanks to the round-robin pointer (Child(Round-Robin(EXT_p))). Thus, when such a buffer becomes free k is copied there. The value of the round-robin pointer is updated only when the extra buffer is released.

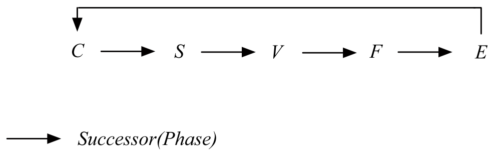


Fig. 19. Successor(Phase(A)).

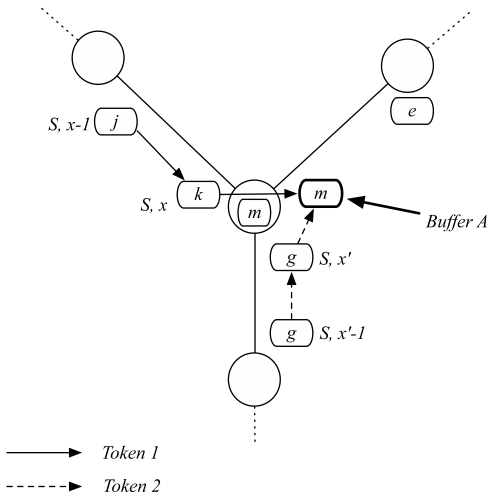


Fig. 20. parent(A) election.

Algorithm 2 Token Circulation Algorithm

- **Token circulation initialization (output buffer)**
  - **R(T)1**: Token-Request(B,A) $\wedge$ Initialization-Allowed(B,A) $\rightarrow$ Token-Initialization(B,A).
- **Search Phase (input/output buffer)**
  - **R(T)2**: S-Forward(id,B,A) $\rightarrow$ Set-Token(id,B,A).
- **Propagation (input/output buffer)**
  - **R(T)3**: Forward(A) $\vee$ Backtrack(A) $\vee$ Change-Direction(A) $\rightarrow$ Update-Phase(A).
- **Cleaning-Phase (input/output buffer)**
  - **R(T)4**: Clean-child(A,E) $\vee$ Another-Token(A,E) $\vee$ Cycle-Escort(A,E) $\rightarrow$ Set-clean-Buffer(A).

In the following, in addition of the data that will be defined in the sequel, some variables and predicates defined in Section 3.2 will be used to describe our Token Circulation algorithm.

To define the macros used in the token circulation algorithm let us first define the following predicate:

- Pseudo-Free(A): Free(A) $\vee$ Edible(A). This predicate indicates that Buffer A either is free or it contains a message whose destination is process(A). A is said to be pseudo free.

Macro:

$$-Search(A) = \begin{cases} A & \text{if Pseudo-Free(A)} \\ B & \text{if Message-to-Send(A,m,B)} \\ \text{Child(A)} & \text{Otherwise} \end{cases}$$

— Successor(Phase(A)): Recall that there are 5 possible phases for the token circulation as follows: C (Clean), S (Search), V (Validation), F (Confirm) and E (Escort). The succession of the phases is given by Fig. 19.

- Smallest-Token-to(A)=B if: Token-State(B) $\wedge$ State-Buffer(B)=(id,?,A,S,x) $\wedge$ ($\forall$ B′, Token-State(B′) $\wedge$ State-Buffer(B′)= (id’,?,A,S,x’) $\Rightarrow id < id'$). This macro determines the parent of Buffer A. This buffer is chosen by considering the identity of the token that has been sent to Buffer A. The parent is the buffer that has sent a token to Buffer A with the smallest identity (in Fig. 20, Buffer A will have to choose between Token1 and Token2, this is done by using the identity of both tokens).

Predicates (Part1)

- Initialization-Allowed(B,A): [Clean-State(A) $\wedge$ Clean-State(B)] $\vee$ (Token-State(A) $\Rightarrow$ (id-Token(A)>id-A $\wedge$ phase(A)=S)). It indicates that Buffer A wants to initialize a token circulation by sending a token to Buffer B (Token-Request(A,B)= true). Both buffers A and B are not part of any token circulation (Clean-State(A) $\wedge$ State-Buffer(B)=State-Buffer(A)).

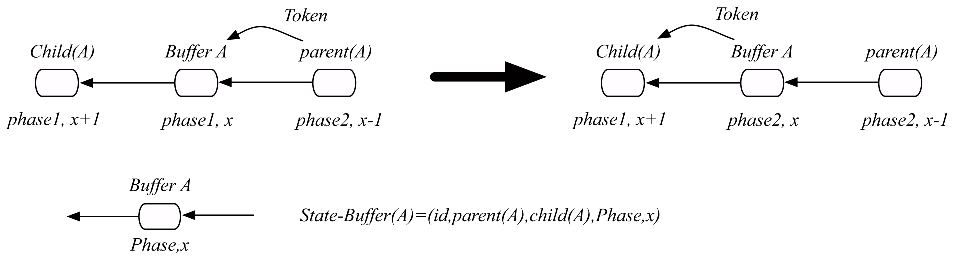


Fig. 21. The token moving Forward.

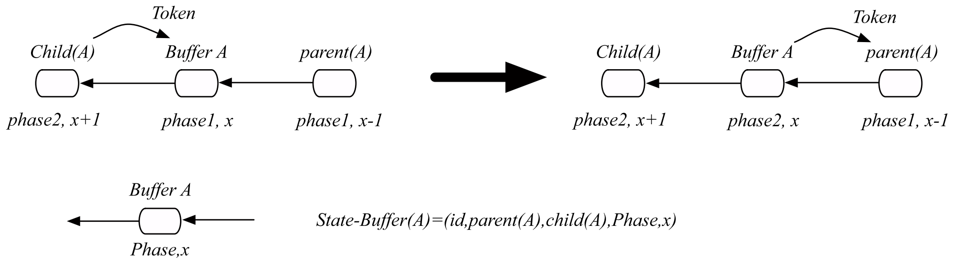


Fig. 22. The token moving Backward.

- Propagation-Allowed(A,id): (Clean-State(A) $\vee$ (Token-State(A) $\wedge$ id-Token(A) > id $\wedge$ phase(A)=S)) $\wedge$ (parent(Search(A))= A $\Rightarrow$ id-Token(Search(A))$\neq$ id). It indicates when the token is allowed to be propagated to Buffer A. This can be done in one of the three following cases: (i) Buffer A is not part of any token circulation (Clean-State(A)). (ii) Buffer A is part of another token circulation such that the identity of this token is greater than the new one and its phase is different from the elements in the set {V,F,E}. (iii) In the case where parent(Search(A))=A, id-Token(Search(A)) must be different from id to avoid the merging with abnormal token circulation.
- S-Forward(id,B,A): Token-State(B) $\wedge$ child(B)=A $\wedge$ Smallest-Token-to(A)=B $\wedge$ Propagation-Allowed(A,id). This predicates is used during the first phase of the token circulation (search). It indicates that Buffer A needs to update it state to be part of the token circulation of the token sent by Buffer B.
- Forward(A): Normal-Filiation(parent(A),A) $\wedge$ phase(parent(A))= successor(phase(A)) $\wedge$ [Last-Buffer(A) $\vee$ (Normal-Filiation(A,child(A)) $\wedge$ phase(child(A))=phase(A))]. Recall that the path of the token circulation is already defined. All the buffers part of this token circulation have set their parent and child pointer. This predicates indicates that the token is moving forwards. If Buffer A is not the last buffer of the token circulation, then the token is sent to Child(A) by Buffer(A). Buffer A sends the token after checking the phases of both its parent (phase(parent(A))=successor(phase(A))) and its child (phase(child(A))=phase(A)) (refer to Fig. 21). If Buffer A is the last buffer (Last-Buffer(A)), then, the token cannot be sent forward anymore. Buffer A updates its state and eventually sends the token back to initialize the next phase.
- Backtrack(A): Normal-Filiation(A,child(A)) $\wedge$ phase(child(A))= successor(phase(A)) $\wedge$ [First-Buffer(A) $\vee$ (Normal-Filiation(parent(A),A) $\wedge$ phase(A)=phase(parent(A)))]. As for Forward(A), the path of the token circulation is already defined. This predicates indicates that the token is moving backward. If Buffer A is not the first buffer of the token circulation then, Buffer A sends the token to parent(A). Before doing so, Buffer A checks the phases of both its parent and its child (refer to Fig. 22). In the case where Buffer A is the first buffer, then the token cannot be sent backward anymore, Buffer A updates its phase after checking the phase of its child so that it can initialize, in the next step, the next phase.
- Change-Last-Buffer(A): Last-Buffer(A) $\wedge$ phase(A)=phase(parent(A)) $\wedge$ phase(A)=phase(child(A)). This indicates that Buffer A is the last buffer of the token circulation id. It allows a buffer to detect the end of either the Validation phase or the Confirm phase and to send the token phase to initialize the next phase.
- V-Initialization(A): Change-Last-Buffer(A) $\wedge$ phase(A)=S. This predicate indicates that Buffer A is ready to initialize the Validation phase in the backward direction.
- F-Initialization(A): First-Buffer(A) $\wedge$ phase(A)=V $\wedge$ phase(child(A)) =phase(A). This predicate indicates that all the path of the token circulation has been validated *i.e.,* the validation phase reached the buffer that has initiated the token circulation. Thus, the token has to be sent back to initialize the Confirm phase.
- E-Initialization(A): Round-Robin($EXT_{process(A)}$)= A $\wedge$ phase(A)=F $\wedge$ Change-Last-Buffer(A). This predicate indicates that Buffer A is ready to initialize the Escort phase in the backward direction.
- Change-Direction(A): Token-State(A) $\wedge$ [V-Initialization(A) $\vee$ F-Initialization(A) $\vee$ E-Initialization(A)]. It indicates that the token must be sent back in the opposite direction. This happens when Buffer A is either the first buffer of the token circulation (the one at level 0) or the last one.
- Clean-child(A, phase): Token-State(A) $\wedge$ phase(A)=phase $\wedge$ (child(A)=A $\vee$ Clean-State(child(A))). Buffer A is the last buffer of the token circulation. The phase of this buffer is equal to the parameter phase. Thus, it has to clean its state.

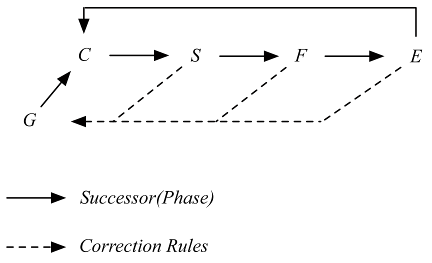


Fig. 23. Transitions between phases.

- Another-Token(A, phase): Token-State(A) ∧ phase(A)=phase ∧ Token-State(child(A)) ∧ id-Token(A)≠ id-Token (child(A)). The last buffer of the token circulation has a child buffer involved in another token circulation.
- Cycle-Cleaning(A, phase): Token-State(A) ∧ phase(A)=phase ∧ Token-State(child(A)) ∧ phase(child(A))=phase ∧ parent(child(A))≠ A ∧ id-Token(A)=id-Token(child(A). The last buffer of the token circulation (Buffer A) has a child buffer that is already in the same phase as A (phase). Note that, in this case, parent(child(A))≠ A. This is used to detect when to initialize the cleaning phase. This happens when phase is respectively equal to E or G.

Actions:

- Token-Initialization(B,A): Synchro[Token-Request(B,A):=false, State-Buffer(A):=(id-A,A,Search(A), S,0)]. Buffer A initializes the requested token circulation by sending the token for the first time. Buffer B sets Token-request(A,B) to false.
- Set-Token(id,B,A): State-Buffer(A):=(id,B,Search(A),S,level(B)+1). Buffer A sends the token that it has received from Buffer B to the buffer determined by Search(A).
- Set-clean-Buffer(A): State-Buffer(A):=(-1,⊥, ⊥, C, -1). The state of Buffer A is cleaned.
- Update-Phase(A): Phase(A):=$Successor(Phase(A))$. Buffer A updates its phase to successor phase(A).

Correction Rules

Previously, we introduced five possible values for the phase of the token circulation. In the following, we introduce a new value (G for Freeze) that will be used to clean a sequence of buffers that are part of an incorrect token circulation (the state of at least one buffer part of this path is not correct *i.e.,* cannot be reached dynamically by executing our protocol). The transitions between the phases is given in Fig. 23. The correction Rules are given in Algorithm 3.

Algorithm 3 Correction Rules of the Token circulation

- **Freeze Initialization (input/output buffer)**
  - **R(T)5**: Initiator-Incoherence(A) ∨ Abnormal-Parent(A)] ∧ phase(A)≠ G → Freeze-Buffer(A).
- **Freeze Propagation (input/output buffer**)
  - **R(T)6**: G-Forward(A) → Freeze-Buffer(A).
- **Cleaning Freeze (input/output buffer**)
  - **R(T)7**: Clean-child(A,G) ∨ Another-Token(A,G) ∨ Cycle-Freeze(A,G) → Set-clean-Buffer(A).
- **Additional correction rules (input/output buffer)**
  - **R(T)8**: Abnormal-Cycle(A) ∨ Error-Free-Buffer(A) ∨ Abnormal-Token-End(A) ∨ Abnormal-child(A) → Set-clean-Buffer(A).
  - **R(T)9**: Same-Link(A,B) ∧ Token-Request(A,B) ∧ Token-State(A) → Token-Request(A,B):=false.
  - **R(T)10**: Incorrect-Clean-State(A) → Set-Clean-Buffer(A).

Predicates (Part2)

- IN-Phase(A,B): (phase(A)= S ⇒ phase(B)∈{S,V}) ∧ (phase(A)=V ⇒ phase(B)=V) ∧ (phase(A)=F ⇒ phase(B)∈{V,F,E}) ∧ (phase(A)=E ⇒ phase(B)=E) ∧ (phase(A) ≠ G ⇒ phase(B) ≠ G). This indicates that the phase of Buffer A is correct with respect to the phase of Buffer B. Observe that phase(A)=C is not considered in this predicate. This is due to the fact that IN-Phase(A,B) is only used when Buffer A is part of a token circulation (Token-State(A) is true).
- Abnormal-Phase(A,B): Normal-Filiation(A,B) ∧ ¬ IN-Phase(A,B). This indicates that the phases between A and its child are incoherent.
- Abnormal-Level(A): Filiation(A,B) ∧ ¬ Normal-Filiation(A,B). This predicate indicates that the levels of Buffer A and its child is incoherent.

- Initiator-Incoherence(A): Token-State(A) $\wedge$ parent(A)=A $\wedge$ (level(A)=0 $\Rightarrow$ id-Token(A)$\neq$ id-A). This predicate indicates that the level of the first buffer (initiator) of a given circulation is incorrect.
- Abnormal-child(A): Abnormal-Phase(A,child(A)) $\vee$ Abnormal-Level(A,child(A)). This predicate indicates that there is an incoherence between Buffer A and Buffer child(A).
- Abnormal-Parent-identity(A): Token-State(A) $\wedge$ [$\neg$ Smaller-id(parent(A),A) $\vee$ Bigger-id(parent(A),A) $\vee$ Clean-State (parent(A))]. This indicates that Buffer A is not the child of its parent, where:
  - Bigger-id(B,A): id-Token(B) $>$ id-Token(A). This indicates that the identity of the token of Buffer B is bigger than the one of Buffer A.
  - Smaller-id(B,A): id-Token(B)$<$ id-token(A) $\Rightarrow$ (phase(A)=S $\wedge$ phase(B)=S). This indicates that in the case where id-Token(B)$<$ id-token(A), both Buffer A and Buffer B are in the Search phase.
- Abnormal-parent(A): Abnormal-Parent-identity(A) $\vee$ Abnormal-Level(parent(A),A) $\vee$ Abnormal-Phase(parent(A),A). It indicates that the state of parent(A) is not coherent with the state of Buffer A.
- G-Forward(A): Token-State(A) $\wedge$ phase(A)$\neq$ G $\wedge$ Token-State(parent(A)) $\wedge$ Filiation(parent(A),A) $\wedge$ phase(parent(A))=G. This indicates that the parent of Buffer A is in the Freeze phase.
- Abnormal-Cycle(A): Token-State(A) $\wedge$ [lasso(A) $\vee$ Full-Cycle(A)] $\wedge$ phase(A)=phase1 $\wedge$ phase(child(A))=phase2 $\wedge$ phase(parent(A))$\neq$ G $\wedge$ [(phase1=S $\wedge$ phase2 $\in$ {V,F,E}) $\vee$ (phase1=V $\wedge$ phase2=E) $\vee$ (phase1=F $\wedge$ phase2 $\in$ {S,V,F,E}) $\wedge$ (phase1=E $\wedge$ phase2$\in${S,V})]. This indicates that the state of child(A) is incorrect (there is an inconstancy while considering the phases). Before cleaning its state, Buffer(A) checks whether its parent is in the freeze phase (G).
- Error-Free-Buffer(A): $\neg$ Free-Buffer(A) $\wedge$ child(A)=A $\wedge$ phase(A)$\in$ {S,V,F} $\wedge$ phase(parent(A))$\neq$ G. The state of Buffer(A) indicates that the token circulation has found a free buffer however Buffer(A) $\neq \epsilon$.
- Abnormal-Token-End(A): State-Token(A) $\wedge$ phase(A) $\in$ {V,F,E} $\wedge$ phase(parent(A))$\neq$ G $\wedge$ [(id-Token(A)$\neq$ id-Token(child(A)) $\vee$ Clean-State(child(A))]. This predicate indicates that child(A) is not part of the same token circulation as A.

Actions

- Freeze-Buffer(A): phase(A):=G. The phase of buffer A is updated to G.

Observe that some correction rules (R(T)8 to R(T)10) do not need the Freeze mechanism. When an incoherency is locally detected, the state of the concerned buffer is cleared (set to $(-1, \perp, \perp, C, -1)$ (refer to Rule R(T)8).

3.3. Proof of correctness

We prove in this section the correctness of our solution. The idea of the proofs is the following: 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.

In the following, a message m that has to perform a u-turn and has generated a token circulation T is said to be associated to T (T is also said to be associated to m). More formally: Message m is said to be associated to $T = A_1 \mapsto A_2 \mapsto \cdots \mapsto A_k$ if $\exists$ Buffer A such that Buffer(A)=m and Same-Link(A,A_1). Buffer A is said to be cleared if State-Buffer(A) is set to $(-1, \perp, \perp, C, -1)$. In the same manner, a token circulation T is said to be cleared, if all the buffers of T clears their state in a finite time. A token circulation is said to be valid, if it is has been initialized after the faults. We distinguish two kinds of valid token circulations: (i) A complete token circulation *i.e.*, all the buffers of T has been visited by T. (ii) An incomplete token circulation *i.e.*, T finds a buffer A that is already in the same token circulation as T but was not visited by T (A was part of a token circulation in the initial configuration). A message m is said to be in a suitable buffer if m is in the right buffer to reach its destination (the right path towards the destination). Buffer A is said to hook Buffer B, if during the search phase, Buffer B becomes the child of Buffer A. In other words, Buffer A hooks Buffer B when A executes S-Forward(A) such that Search(A)=B.

Before detailing the proofs, let us now define some notions that will be used later.

Definition 1 (***B-Successor***)**.** Let A_1 and A_2 be two buffers of the system. A_2 is called the B-successor of A_1, denoted $A_1 \mapsto A_2$, if the following predicate is satisfied:

$$\text{Normal-Filiation}(A_1,A_2) \wedge \text{IN-Phase}(A_1,A_2).$$

Definition 2 (***Normal Token Circulation***)**.** A maximal sequence of k buffers $A_1 \mapsto A_2 \mapsto \cdots \mapsto A_k$ starting from A_1 is called a normal Token circulation if First-Buffer(A_1) is satisfied.

Definition 3 (***Abnormal Token Circulation***)**.** A maximal sequence of k buffers $A_1 \mapsto A_2 \mapsto \cdots \mapsto A_k$ starting from A_1 is called an abnormal Token circulation if the following predicate is satisfied:

$$\text{Initiator-Incoherence}(A_1) \vee \text{Abnormal-Parent}(A_1).$$

Remark 3. Observe that, for a normal token circulation there exist at most two segments, respectively $S_1 = A_1 \mapsto A_2 \mapsto \cdots \mapsto A_i$ and $S_2 = A_{i+1} \mapsto A_{i+2} \mapsto \cdots \mapsto A_k$ of different phases ($\forall\ 1 \leq j < i$ phase(A_j)=phase(A_{j+1}) and $\forall\ (i+1) \leq j' < k$ phase($A_{j'}$)=phase($A_{j'+1}$)). When the token circulation is abnormal, there exist at most three segments respectively S_1, S_2, and S_3 of different phases such that $\forall\ A_j$ part of S_1, phase(A_j)=G (all the buffers part of S_1 have their phase set to G).

Definition 4 (***G-Validated Token Circulation***). A maximal sequence of k buffers $T = A_1 \mapsto A_2 \mapsto \cdots \mapsto A_k$, starting from A_1 is said to be G-Validated, if there exist a prefix possibly empty $P = A_1 \mapsto A_2 \mapsto \cdots \mapsto A_i$ and a suffix possibly empty $X = A_{i+1} \mapsto A_{i+2} \mapsto \cdots \mapsto A_k$ such that phase(A_j)=G for $1 \leq j \leq i$, and phase($A_{j'}$)∈{V,F,E} for $(i+1) \leq j' \leq k$. Let us refer to the number of Buffers of P by $|P|$.

Lemma 1. *If an abnormal token circulation T is G-Validated, then T will be cleared in a finite time.*

Proof. Let us refer to such a token circulation by $T = A_1 \mapsto A_2 \mapsto \cdots \mapsto A_k$. Since T has been G-Validated, no other token circulation can use a buffer of T (refer to Rule R(T)2 (Algorithm 2), Predicates S-Forward, and Propagation-allowed(A,id). Let P and X be respectively the prefix and the suffix of T as defined Definition 4. Observe that since T is G-Validated, phase(A_k)∈ $\{E, F, G\}$. Thus, $|T|$ cannot increase. However, $|T|$ can decreases since on the last buffer of T, R(T)4 or R(T)8 can be enabled. Two cases are possible:

1. $|P| \geq 1$. Let $i = |P|$. In this case, Rule R(T)6 becomes enabled on A_{i+1}. Once A_{i+1} updates its phase to G, $|P|$ increases whereas $|X|$ decreases. If $|X| > 0$ then by induction, eventually $|X| = 0$. Let $A_{j'}$ ($j' \leq k$) be the last buffer of T. When $|X| = 0$, R(T)7 becomes enabled on $A_{j'}$. Once the rule is executed, $A_{j'}$ clears its state. By induction, all the buffers of T eventually executes R(T)7 to clear their state. Thus, T is eventually cleared.
2. $|P| = 0$. Since T is an abnormal token circulation, Rule R(T)5 is enabled on Buffer A_1. If during one round, the token circulation T is reduced to A_1 before A_1 executes R(T)5, then A_1 clears its state by executing Rule R(T)8. Otherwise, since that Rule R(T)5 keeps being enabled and the daemon is weakly fair, A_1 executes R(T)5 in a finite time and we retrieve Case 1.

   Since in both cases, T is eventually cleared, the lemma holds. □

In the following, we first show that a valid token circulation never merge with an abnormal token circulation they become part of the same token circulation).

Lemma 2. *If there is an abnormal token circulation* $T = A_1 \mapsto A_2 \mapsto \cdots \mapsto A_k$ *that contains a segment* $Y = A_i \mapsto A_{i+1} \mapsto \cdots \mapsto A_j$ *($i \geq 1$ and $j \leq k$) in the Search phase then the Buffers of T can never sets their phase to F.*

Proof. If Y is not a prefix, then by definition of an abnormal token circulation, Y can only be preceded by a segment whose buffers have their phase equal to G. Since the Confirm phase can only be initiated by the first buffer of a normal token circulation, Rule R(T)3 will never be enabled on A_1. So, T will never contain a buffer in the F phase. Thus, the lemma holds. □

Lemma 3. *If a normal token circulation T is G-validated, then T is eventually cleared.*

Proof. Observe that since T is a normal token circulation, no buffer in P has its phase in G ($|P| = 0$). Let us refer to such a token circulation by $T = A_1 \mapsto A_2 \mapsto \cdots \mapsto A_k$. Since the token circulation is G-validated, then at least all the buffers of T have set their phase to V. The cases below are possible:

1. The escort phase is being executed. In this case there exist a prefix, possibly empty, $P = A_1 \mapsto A_2 \mapsto \cdots \mapsto A_i$ and a possibly empty suffix $X = A_{i+1} \mapsto A_{i+2} \mapsto \cdots \mapsto A_k$ such that phase(A_j)=F for $1 \leq j \leq i$, and phase($A_{j'}$)= E for $(i+1) \leq j' \leq k$. Two sub-cases are possible:
   (a) $|P| \neq 0$. Three sub-cases are possible as follows: (i) $|X| = 0$ and T is an incomplete token circulation. In this case, A_k has hooked a buffer B of an abnormal token circulation T' when both A_k and B were in the Search phase. Note that B=child(A). By hypothesis, phase(A_k)=F. From Lemma 2, phase(B) will never be equal to F as long as B is part of T'. R(T)8 is the only rule that can be executed on A_k (Abnormal-Child(A_k)) is satisfied). Once the rule is executed, A_k clears its state. By induction, all the buffers of T eventually clear their state. (ii) $|X| = 0$ and T is a complete token circulation. In this case R(T)8 is enabled on A_k only in the case where Round-Robin($EXT_{process(A_k)}$)=A_k. Recall that Round-Robin($EXT_{process(A_k)}$) points only towards buffers that are ready to initialize the Escort phase. If Round-Robin($EXT_{process(A_k)}$) points on Buffer B such that B $\neq A_k$, then according to Lemmas 1, 3 and 4, B is cleared in a finite time. Thus, Round-Robin($EXT_{process(A_k)}$) eventually updates its value and points towards A_k. R(T)3 is then enabled on A_k. When the rule is executed, A_k updates its phase to E. We retrieve Case (iii). (iii) $|X| > 0$. Rule R(T)3 is enabled on A_i. Since the rule keeps being enabled and since we consider a weakly fair daemon, Buffer A_i eventually executes R(T)3. Its phase is updated to E. Thus, $|P|$ decreases. Rule R(T)3 becomes enabled on A_{i-1}. By induction, eventually $|P| = 0$ (all the buffers of T update their phase to E). We retrieve Case 1b. Remark that in the case where T is not a cycle, the successive last buffers of T which have their phase in E can execute R(T)4. So $|T|$ can decrease.
   (b) $|P| = 0$. Let $|T| = k'$ ($k' \leq k$), in this case Rule R(T)4 is enabled on $A_{k'}$. Once the rule is executed, $A_{k'}$ clears its state. Thus, $|T|$ decreases. By induction, eventually $|T| = 0$ (all the buffers of T cleared their state).

2. The Confirm phase is being executed. In this case there exist a prefix, possibly empty, $P = A_1 \mapsto A_2 \mapsto \cdots \mapsto A_i$ and a non empty suffix $X = A_{i+1} \mapsto A_{i+2} \mapsto \cdots \mapsto A_k$ such that phase(A_j)=F for $1 \leq j \leq i$, and phase($A_{j'}$)=V for $(i+1) \leq j' \leq k$. In this case, Rule R(T)3 becomes enabled on A_{i+1} (A_1 if P is empty *i.e.,* $i = 0$)). Once the rule is executed, A_{i+1} updates its phase to F. Hence, $|X|$ decreases whereas $|P|$ increases. By induction, eventually $|X| = 0$ (all the buffers of T updated their phase to F). Two sub-cases are then possible:
   (a) T is a complete valid token circulation. In this case, T has either found a free buffer or detected a cycle. In both cases, R(T)3 becomes enabled on Buffer A_k. We retrieve Case 1.
   (b) T is an incomplete token circulation. R(T)8 is then enabled on A_k. Once the rule is executed, A_k clears its state. Thus, $|T|$ decreases. By induction, eventually $|T| = 0$ (all the buffers of T are eventually cleared).

From the cases above we can deduce that T is eventually cleared and the lemma holds. □

Lemma 4. *Let X be a suffix of buffers of a given token circulation T. If X is G-validated, then X is cleared in a finite time.*

Proof. If T is G-validated, the result is given by Lemmas 1 and 3. Thus, in the following we assume that T is not G-validated *i.e., T* has a non empty prefix of buffers that are in the Search phase (S). If there exists in the path of a given token circulation T, a suffix of buffers $X = A_{i+1} \mapsto A_{i+2} \mapsto \cdots \mapsto A_k$ that is G-validated, and a prefix of buffers $P = A_1 \mapsto A_2 \mapsto \cdots \mapsto A_i$ that are still in the search phase then in this case, $\forall j$, $(i+1) \leq j \leq k \Rightarrow$ phase(A_j)=V (recall that Predicate IN-Phase(A_i, A_j) is true for any couple of successive buffers). Note that Rule $R(T)3$ is enabled on A_i. If in one round, there is no other token circulation that uses Buffer A_i, Buffer A_i executes R(T)3 and thus updates its state to V. Note that $|P|$ has decreased. By induction, if there is no other token circulation that uses the last buffer of P, then $|P|$ is eventually reduced to 1 (P contains only Buffer A_1). Observe that in the case where there exists another token with a smaller identity that hooks a buffer of the prefix P ($|P| \geq 1$), then the suffix of T (that we will still call T) that still contains X becomes an abnormal token circulation. Note that T is either G-validated if its new prefix (we keep the same notation P) satisfies $|P| = 0$. Otherwise, $|P|$ has been reduced. By induction $|P|$ is either equal to 0 or to 1. If $|P| = 0$, then T is G-validated. When $|P| = 1$, two cases are possible:

- T is an abnormal token circulation. R(T)5 is enabled on A_1. A_1 eventually executes R(T)5 and updates its phase to G. T becomes then G-validated.
- T is a valid token circulation. R(T)3 is in this case, enabled on A_1. A_1 executes R(T)3 and updates its phase to V. T becomes then G-validated.

From the cases above, we can deduce that the suffix X eventually becomes part of a token circulation that is G-validated. According to Lemmas 1 and 3, X is eventually cleared and the lemma holds. □

Lemma 5. *If there exist several token circulations that are executed in the system, then the token circulation T, that has the smallest identity, is cleared in a finite time.*

Proof. Let us refer to the token circulation that has the smallest identity by $T = A_1, A_2, \ldots, A_k$. Note that if T is G-validated then according to Lemmas 1 and 3, T is cleared in a finite time. In the case where T contains a suffix that is G-validated, then From Lemma 4, we know that each suffix X that is G-validated is cleared in a finite time. In fact, all the part of the token circulation that keeps being attached to X is cleared in a finite time. Since we consider in this proof the token circulation T that has the smallest identity, no other token circulation can hook a buffer of T. Thus, Lemma 4 allow us to affirm that T is cleared in a finite time. Hence, it remains to prove that T is cleared when T does not have a suffix that is G-validated.

So, let us consider now, the token circulations that contains (i) a prefix P such that all the buffers in P are in the G phase and a suffix X such that all the buffers of X are in the S phase (Note that $|P| \geq 0$). Recall that no other token circulation can hook a buffer of T. Note also that while T has not been completely cleared, T keeps its first buffer A_1. Two cases are possible as follow:

1. T is abnormal. Note that T cannot progress and include new buffers forever since the number of buffers in the system is finite. Two cases are possible:
   (a) $|P| = 0$. R(T)5 keeps being enabled on A_1. If during one round, the token circulation T is not reduced to only A_1, then A_1 eventually executes R(T)5. Once the rule is executed, A_1 updates its phase to G. Note that T consists now of a prefix P that contains Buffer A_1 such that phase(A_1)=G and a suffix X such that $\forall\ 1 < i \leq k$, phase(A_i)$\neq$ G. We retrieve Case (1b).
   (b) $|P| > 0$. Let $P = A_1 \mapsto A_2 \mapsto \cdots \mapsto A_i$ be the prefix of T such that $\forall\ 1 \leq j \leq i$, phase(A_j)=G and let $X = A_{i+1}, A_{i+2}, \ldots, A_k$ be the suffix of T such that $\forall\ (i+1) \leq j \leq k$, phase($A_j$)=S. Rule R(T)6 becomes enabled on A_{i+1}. When A_{i+1} executes the rule, it sets its phase to G. Thus $|P|$ increases whereas $|X|$ decreases. By induction, eventually $|X| = 0$. T becomes then G-validated. Thus, according to Lemma 1, T is eventually cleared.

(b) T is normal. Note that $|P| = 0$. During the progression of T, if T wants to hook a buffer B that has a phase ∈{V,F,E} (B is already part of a token circulation that has G-validated a sub-part of its phase), then according to Lemmas 1, 3 and 4, B is cleared in a finite time. Thus, T will eventually continue its progression. Hence, T stops its progression when it finds a free buffer, or detects a cycle (note that T can be incomplete *i.e.,* there is on the path of T another token circulation $T2$ with the same identity as T — $T2$ was in the system in the initial state —). Now, R(T)3 is enabled on the last buffer of T. When the rule is executed, the last buffer of T updates its phase to V. T consists now of a prefix P such that $\forall A \in P$ phase(A)=S, and a suffix X such that $\forall A \in X$ phase(A)=V. On the last buffer B of P, R(T)3 is always enabled (recall that no other token circulation can break T). Since, we assume a weakly fair daemon, R(T)3 is eventually executed. Thus, B updates its phase to V. Note that $|P|$ decreases while $|X|$ increases. By induction, eventually $|P| = 0$. Thus, T becomes G-validated. According to Lemma 3, T is eventually cleared.

From the cases above, we can deduce that T is eventually cleared and the lemma holds. □

In the following, we show that although a token circulation may abort (see the proof of Lemma 5, Case 2), at least one message that has to undergo a route change will eventually initialize a token circulation that will become completed (refer to Lemma 10).

Lemma 6. *In the case where Token-Request(A,B)=true. Then, Token-Request(A,B) is set to false in a finite time.*

Proof. In the case Token-Request(A,B)=true such that Rule R(T)1 is enabled, then, Token-Request(A,B) is set to false by the token circulation algorithm when this Rule R(T)1 is executed. Otherwise, the two cases below are possible:

- The predicate Token-State(A) is true. In this case Token-Request(A,B) is set to false by the token circulation algorithm by executing Rule R(T)9.
- The predicate Free(A) ∨ ¬ Uturn(m) is true. In this case, Rule R(F)12 is enabled on Buffer A. Once executed, Token-Request(A,B) is set to false.

From the cases above, we can deduce that in the case Token-Request(A,B)=true, it will be set to false in a finite time. □

Lemma 7. *The extra buffer of any process p (EXT_p) is infinitely often free.*

Proof. Recall that the extra buffer is used to solve the problem of deadlocks (cycles). Suppose that the extra buffer contains a message m. The cases below are possible:

1. Round-Robin(EXT_p)=⊥ or phase(Round-Robin(EXT_p))≠ E. In this case the message m is deleted by executing $R(F)11$.
2. ∃ A such that Round-Robin(EXT_p)=A and both Token-State(A) and Token-State(child(A)) are true. Let us refer to the token circulation that includes Buffer A by T. The three following sub-cases are possible:
   (a) T is abnormal. In this case, all its buffers of T are eventually cleared (refer to Lemma 1). We retrieve Case 1.
   (b) T is a normal token circulation. In this case, either

      - (i) End-Token(child(A)) is satisfied. The message in EXT_p is either deleted (if child(A) is not free, refer to Rule R(F)12) or copied in child(A) and deleted from EXT_p (if child(A) is free, refer to Rule R(F)10).

      - Or (ii) End-Token(child(A)) is not satisfied. In this case, phase(A)=E ∧ phase(child(A))=F and phase(child (child(A)))=F. Hence there is a suffix X, such that the buffer of X are in Phase E. According to the proof of Lemma 3, eventually phase(child(child(A)))=E. End-Token(child(A)) becomes satisfied, we retrieve Case (i).

From the cases above, we can deduce that the extra buffer of any process p (EXT_p) is infinitely often free and the lemma holds. □

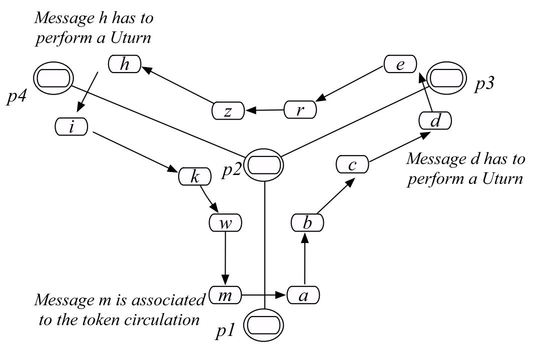
Lemma 8. *Any token circulation that has the smallest identity among the ones that can be concurrently executed in the system can be eventually initialized.*

Proof. From Lemma 6, we know that Token-Request(A,B) is set to false in a finite time. From Lemma 7, we know that EXT_p is infinitely often free. From Lemmas 1, 3 and 4, we know that if there is a token circulation T (or a suffix of T: X) such that T is G-validated (X is G-validated), then T (X) is cleared in a finite time. On another hand, Lemma 5 proves that the token circulation with the smallest identity is eventually cleared. Thus, any token circulation that has the smallest identity among the ones that are executed in the system can be initialized in a finite time. □

Lemma 9. *Let m be a message and let T be a valid token circulation associated to m. If T is eventually completed and G-validated, then at least one message eventually undergoes a route change.*

Proof. Let us refer to the normal complete token circulation associated to m and that is G-Validated by $T = A_1 \mapsto A_2 \mapsto \cdots \mapsto A_k$. Since T is a normal complete token circulation that is G-Validated and associated to a message, T has either found a pseudo free buffer or detected a cycle (a lasso or a full-cycle). According to the proof of Lemma 3, all the buffers of T eventually have their phase set to F. Note that, at that time, the token is held by A_k. Let us first focus on the cases where T has found a pseudo free buffer (Pseudo-Free(A_k) is satisfied). Note that in the case where Edible(A_k) is satisfied, when A_k executes R(T)2, it also executes R(F)2 since both rules are enabled at the same time, so A_k becomes free. Thus, R(T)3 becomes enabled on A_k (Change-Direction(A_k) is satisfied) and R(F)5 too (since A_k is free). When A_k executes R(T)3, it also executes R(F)5 during the same step. Rules R(T)3 and R(F)5 become enabled on A_{k-1}, Buffer A_{k-1} updates its phase to E and at the same time the message of A_{k-2} is copied in A_{k-1}. By induction, R(T)3 and R(F)7 eventually become enabled on A_1 such that A_1 is free. Thus, Message m is copied in A_1. Note that Message m has performed a route-change in this case.

Let us now consider the case where a cycle is detected (full-cycle or lasso). Observe that since the buffer graph is defined on a tree topology, in the case where there is a cycle, there is at least one message that has to perform a u-turn (refer to Fig. 24). R(T)3 is then enabled on A_k only if Round-Robin($EXT_{process(A_k)}$)=A_k. As shown in the proof of Lemma 3, Round-Robin($EXT_{process(A_k)}$) eventually updates its value and points towards A_k. A_k is then enabled to initialize the escort phase. As


Message h has to perform a Uturn
Message d has to perform a Uturn
Message m is associated to the token circulation
p1
p2
p3
p4
h
e
z
r
i
d
k
c
w
b
m
a
(a) The case of a Full-Cycle.

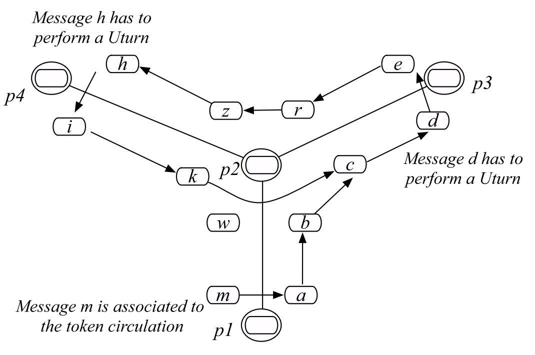

Message h has to perform a Uturn
Message d has to perform a Uturn
Message m is associated to the token circulation
p1
p2
p3
p4
h
e
z
r
i
d
k
c
w
b
m
a
(b) The case of a Lasso.

Fig. 24. The case of a cycle.

shown in the previous case (the case where T has found a pseudo free buffer), the escort phase will eventually reach all the buffers of the cycle part of T, along with the free slot. So any message on the cycle part of T will progress to its next buffer. Thus, at least the messages at the entry of a u-turn perform a u-turn (see messages d and h in Fig. 24). Note that in the case of a full-cycle, Message m associated to T also undergoes a route change.

We can deduce from the cases above that at least one message has undergone a route change and the lemma holds. □

Lemma 10. *When the routing tables are stabilized, if there are some messages that have to perform a route change, then at least one of them will eventually do.*

Proof. The proof is by contradiction. Suppose that the routing tables are stabilized and that all the messages that are in a non suitable buffer cannot be forwarded anymore. Let R be the set of such messages. First, suppose that there is no message $m_1 \in R$ that is in an input buffer *i.e.*, all the messages that are in a non suitable buffer, are in an output buffer. In this case, the buffer graph is a DAG. In addition, eventually no token circulation can be initialized (according to Lemma 6, Token-Request(A,B) is set to false in a finite time. Since there is no message of R that is in an input buffer, R(F)8 is never enabled). On the other hand, we know that if there is no more token circulations that are initialized, then all the token circulations are cleared in a finite time (Corollary to Lemmas 1 and 3–5).

Let us now, consider one message $m_1 \in R$. Since m_1 is in an output buffer (let A_1 be this buffer), the next destination of m_1 is the input buffer on the same link (let A_2 be this buffer). Observe that Same-Link(A_1,A_2) is not verified. Since m_1 cannot be forwarded, A_2 is never free. Thus, there is a message m_2 is A_2 that can never be forwarded. Let A_3 be the next buffer of m_2. Observe that A_3 is in this case an output buffer. Since m_2 cannot be forwarded anymore, A_3 is never free (recall that there is no token circulation that is executed in the system, thus, the fair pointer mechanism defined on the output buffers cannot be disturbed anymore. The fairness of such pointer mechanism guarantees that if A_3 is free, then m_2 will be able to be copied in A_3). Thus, A_3 contains a message m_3 that cannot be forwarded as well anymore. By induction, there is a sequence of buffer $A_1, A_2, \ldots, A_k$ such that: $\forall\ 1 \le i \le k$, Buffer(A_i)=m_i, Next-Buffer(A_i, m_i)=A_{i+1}, and m_i cannot be forwarded anymore. Since the buffer graph is a DAG (recall that all messages of R are in an output buffer) then A_k is an input buffer of a leave process. Note that $m_k \notin R$. Thus, A_k is the destination of m_k. R(F)2 is enabled on A_k. Since the rule keeps being enabled on A_k and the daemon is weakly fair. m_k is eventually consumed. Hence, A_k is free. Contradiction.

Consequently, there are some messages of R that are in input buffers. These messages will allow the initialization of token circulations. Let us consider the normal token circulation that is associate to a given message m (Note that $m \in R$) that has the smallest identity (let us refer to such a token by T). From Lemma 5, we can deduce that eventually, there will be in the system no other token circulation with a smaller (or equal) identity than (as) T (all the abnormal token circulations that have a smaller or equal identity as T, are cleared in a finite time). Message m keeps requesting a token circulation unless it performs a route change. From Lemma 8, the buffer of m is always able to generate a token circulation. Thus, such a buffer will be able eventually, to generate a token circulation when all the token circulations that have a smaller (equal) identity than (as) T are cleared. In this case, T cannot regress before being completed, however, it can be stopped by another token circulation that is G-validated. According to Lemmas 1, 3 and 4 such token circulations are cleared in a finite time. Thus, T continues its progression and eventually completes its path. Let $T = A_1 \mapsto A_2 \mapsto \cdots \mapsto A_k$ be the path of T once completed. R(T)3 becomes enabled on A_k. Once the rule is executed, A_k update its phase to V. Rule R(T)3 becomes enabled on A_{k-1}. By induction, T becomes eventually G-Validated (recall that no other token circulation can hook a buffer of T). According to Lemma 9, m undergoes a route change. Contradiction.

We can deduce that when the routing tables are stabilized, if there are some messages that have to perform a route change, then at least one of them will eventually do. Thus, the lemma holds. □

Lemma 11. *When the routing tables are stabilized all the messages will be in suitable buffers in a finite time.*

Proof. When the routing tables are stabilized, some messages may be on the wrong direction, however, the number of such messages never increases since both the generation and the progression of the new messages are always performed in a suitable buffer. On another hand, according to Lemma 10, there is at least one message in a non suitable buffer that undergoes a route change. Hence, the number of such messages decreases. By induction, eventually all the messages will be in a suitable buffer and the lemma holds. □

Lemma 12. *Let m be a valid message in the extra buffer of a given process p (EXT_p=m) then, m cannot be deleted before being copied in the free slot created by the appropriate token circulation.*

Proof. Note that when a given token circulation finds a free buffer, the extra-buffer is not used. Thus, in the following, this case is not considered.

Since m is normal, when it is copied from Buffer A into the extra buffer, Buffer A is part of a complete token circulation T. Let us refer to the path of the token circulation by $T = A_1 \mapsto A_2 \mapsto \cdots \mapsto A_k$. Observe that Buffer A_k is the one that contains Message m. In the following we consider only the case of a full-cycle (the same reasoning holds when a lasso is detected). In order to prove our lemma, we first show that there is a synchrony between the forwarding algorithm and the token circulation algorithm. In the case of a full-cycle, T is a valid token circulation that has been initialized because of m. Since m was copied in EXT_p and since T is valid, T is a complete token circulation that has G-validated all its path ($\forall\, 1 \le i \le k$, phase(A_i)=F). R(F)9 and R(T)3 become enabled on Buffer A_k. Recall that both rules are executed, in this case, at the same time. Thus, m is copied in the extra buffer of process(A_k) and phase(A_k) is set to E (Buffer A copies also the message that is in Buffer parent(A) at the same time, refer to Rule R(F)9). A_{k-1} becomes a free buffer. R(T)3 and R(F)5 become enabled on A_{k-1}. When both rules are executed phase(A_{k-1})=E and A_{k-2} is free (refer to Fig. 8). Similarly, both R(T)3 and R(F)5 become enabled on A_{k-2}. When both rules are executed, A_{k-2} updates its phase to E and A_{k-3} becomes free. We can show by induction that eventually R(T)3 becomes enabled on A_1 such that A_1 is free. In this case, Both Rules R(F)10 and R(T)3 becomes enabled on A_1. Once the rules are executed. The message m is copied in A_1 and deleted from EXT_p. Thus, the lemma holds. □

We can now detect in some cases if the message that is the extra buffer is not valid (either the token circulation does not bring a free slot in the right buffer (the next buffer of the message in the extra buffer) or, there is a message m in the extra buffer of a given process such that there is no token circulation that is executed). Observe that the algorithm deletes a message only when we are sure that the message in the extra buffer is invalid, refer to Rules $(MF)R11$ and $(MF)R12$. Thus, we have the following theorem:

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

Proof. Let m be a message, then:

- Rule R(F)2 allow the message m to be consumed an thus sent to the higher level. Note that this rule is executed only when m has reached its destination.
- By construction of Rules R(F)1, R(F)3, and R(F)5, m is not deleted since the message is only copied in a new buffer A. In addition, these rules are not enabled unless Buffer A is free.
- By construction of Rules R(F)8, and R(F)13, m is deleted since the rules only update some variables.
- By construction of Rules R(F)6, R(F)7, R(F)9, and R(F)10, m is not deleted since m is just copied in the next buffer to reach its destination (this buffer is either defined by the routing table and the buffer graph or, defined by the path of the token circulation. Note that these rules are enabled only if Free(A) is true. On another hand, on a given buffer, there is at most one message that can be copied simultaneously (the round-robin pointer defined on the output buffers guarantees that at a given time, only one message can be copied. The input buffers can only copy the messages that are on the output buffer connected to them). Hence, m is not deleted.
- By construction of Rule R(F)11 and R(F)12. The message m is deleted however, m is not valid (refer to Lemma 12).
- By construction of Rule R(F)4, the message m is deleted from Buffer A. However, before erasing the message m, we are sure that there is a copy of the message in Next-Buffer(A,m). Thus, once the message m deleted, there is still a copy in the system.

We can deduce from all the cases above that no valid message is deleted unless it is delivered to its destination, hence the lemma holds. □

Lemma 13. *When the routing tables are stabilized and all the messages are in suitable buffer, no Token circulation is initiated.*

Proof. According to Lemma 6, for any buffers A and B, Token-Request(A,B) is set to false in a finite time. The only rule that sets Token-Request(A,B) to true is Rule R(F)8. However, Rule R(F)8 is never enabled since according to Lemma 11, no message is in a non suitable buffer. Thus, the lemma holds. □

The fair pointer mechanism cannot be disturbed anymore by the token circulations. Note that our buffer graph is a DAG when the routing tables are stabilized and when all the messages are in a suitable buffer, thus:

Lemma 14. *All the buffers of the system are infinitely often free.*

Proof. The proof is by contradiction. Assume that there is a Buffer A_1 that is never free. This means that there is a message m_1 that cannot be consumed or forwarded to any other buffer. Let Buffer A_2 be the next buffer by which m_1 has to transit in order to reach its destination. Since m_1 cannot be forwarded to A_2, A_2 contains also a message (let this message be m_2). Note that the fair pointer mechanism cannot be disturbed anymore by some token circulation, therefore, if A_2 becomes free, m_1 will be eventually copied in A_2. Hence, m_2 cannot be forwarded nor consumed as well. The same holds for the next buffer of m_2 (A_3) and so on. Thus, there exists a sequence of buffer $A_1, A_2, \ldots, A_k$, that are never free such that, the destination of Message m_i that is in Buffer A_i is Buffer A_{i+1}. Since our buffer graph is a DAG, A_k is the input buffer of a leaf process. On another hand, since the routing tables are stabilized and since all the messages are in a suitable buffer, no message performs a u-turn. The destination of Message m_k is process(A_k). Hence, R(F)2 becomes enabled and keeps being enabled on A_k. Since we consider a weakly fair daemon, R(F)2 is eventually executed. m_k is consumed and deleted from A_k. Contradiction.

We can conclude that all the buffers of the system are infinitely often free and the lemmas holds. □

Lemma 15. *Any message can be generated in a finite time under a weakly fair daemon.*

Proof. Suppose that a process p wants to generate a message on Buffer A. Two cases are possible:

1. Free(A). In this case, the process executes either Rule R(F)2 or R(F)3 in a finite time. The result of this execution depends on the value of the pointer. Two cases are possible:
   - the pointer refers to Rule R(F)2. Once the rule is executed, m is generated in Buffer A and the lemma holds.
   - the pointer refers to Rule R(F)3. Once the rule is executed, Buffer A becomes occupied ($\neg$ Free(A)) and we retrieve case 2. Note that the fairness of the pointer guarantees that this case cannot appear infinitely.
2. $\neg$ Free(A). Since according to Lemma 14, all the buffers are infinitely often free we are sure that Buffer(A) becomes free in a finite time and we retrieve 1.

We can deduce that every process can generate a message in a finite time. □

We can now state the following theorem:

Theorem 2. *Neither deadlock nor starvation situations appear in the system.*

Proof. According to Lemma 14, there exists no message that stays locked in one buffer. On another hand, according to Lemma 15, every process is able to generate a message in a finite time. Hence, the Theorem holds. □

Lemma 16. *The forwarding protocol never duplicates a valid message even if the routing algorithm runs simultaneously.*

Proof. Let consider the message m. The cases below are possible:

- m is in the extra buffer. m is then either deleted (refer to Rules R(F)11 and R(F)12) or copied in a buffer (refer to Rules R(F)9 and R(F)10). Considering the two latter actions (Rules R(F)9 and R(F)10)) m is copied in the new buffer and deleted from the extra buffer in the same step.
- m in Buffer A (Buffer A is not an extra buffer). The following cases are then possible:
  - m is consumed (Rule R(F)2 is executed). Message m is deleted since a new value overwrites it. Note that the consumption of Message m is enabled on Buffer A only if the copy of this message in the previous buffer (No-duplication(m,A)) has been deleted.
  - m is copied in the extra buffer (Rule R(F)9 is executed). The message m is copied in the extra buffer of process(A) and deleted from buffer A.
  - m is transmitted to its next buffer B (Assume that m is in Buffer A). In this case m is copied in B, however this is done only when No-duplication(m,A) is satisfied. Thus, we can have simultaneously at most two copies of the message (in A and B). Observe that (as mentioned in Remark 2), if m is in an input buffer then m is copied in the output buffer an deleted from the input buffer. In the case where m is in an output buffer then m is neither consumed nor transmitted unless the copy in the output buffer is deleted.

From the cases above we can deduce that no message is duplicated in the system. □

Theorem 3. *Every valid message is delivered once and only once to its destination.*

Proof. From Theorem 2 and Lemma 11, we can deduce that each valid message is delivered to its destination at least once. Lemma 16 ensures that the message is delivered at most once. Thus, the lemma holds. □

Theorem 4. *The proposed algorithm is a snap-stabilizing message forwarding algorithm under a weakly fair daemon.*

Proof. From Theorem 2, neither deadlocks nor starvation appear in the system. From Theorem 3, every valid message is delivered once and only once to its destination. Hence, the theorem holds. □

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] A. Cournier, S. Dubois, A. Lamani, F. Petit, V. Villain, Snap-stabilizing linear message forwarding, in: Proceedings of 12th International Symposium on Stabilization, Safety, and Security of Distributed Systems, SSS, Vol. 6366, New York, NY, USA, September 20–22, 2010, pp. 546–559.
[2] S. Dolev, Self-stabilization, MIT Press, 2000.
[3] A. Bui, A. Datta, F. Petit, V. Villain, Snap-stabilization and PIF in tree networks, Distributed Computing 20 (1) (2007) 3–19.
[4] J. Duato, A necessary and sufficient condition for deadlock-free routing in cut-through and store-and-forward networks, IEEE Trans. Parallel Distrib. Syst. 7 (8) (1996) 841–854.
[5] P.M. Merlin, P.J. Schweitzer, Deadlock avoidance in store-and-forward networks, in: Jerusalem Conference on Information Technology, 1978, pp. 577–581.
[6] S. Toueg, Deadlock- and livelock-free packet switching networks, in: STOC, 1980, pp. 94–99.
[7] S. Toueg, J.D. Ullman, Deadlock-free packet switching networks, SIAM J. Comput. 10 (3) (1981) 594–611.
[8] S. Dolev, J. Welch, Crash resilient communication in dynamic networks, IEEE Trans. Comput. 46 (1) (1997) 14–26.
[9] B. Awerbuch, B. Patt-Shamir, G. Varghese, Self-stabilizing end-to-end communication, J. High Speed Netw. 5 (4) (1996) 365–381.
[10] E. Kushilevitz, R. Ostrovsky, A. Rosén, Log-space polynomial end-to-end communication, in: STOC ’95: Proceedings of the Twenty-seventh Annual ACM Symposium on Theory of Computing, ACM, 1995, pp. 559–568.
[11] A. Cournier, S. Dubois, V. Villain, 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, pp. 1–11.
[12] A. Cournier, S. Dubois, V. Villain, How to improve snap-stabilizing point-to-point communication space complexity? in: Stabilization, Safety, and Security of Distributed Systems, 11th International Symposium, SSS 2009, in: Lecture Notes in Computer Science, vol. 5873, 2009, pp. 195–208.
[13] Edsger W. Dijkstra, Self-stabilizing Systems in Spite of Distributed Control, Commu. ACM 17 (11) (1974) 643–644.
[14] J. Burns, M. Gouda, R. Miller, On relaxing interleaving assumptions, in: Proceedings of the MCC Workshop on Self-Stabilizing Systems, MCC Technical Report No. STP-379-89, 1989.
[15] P.M. Merlin, P.J. Schweitzer, Deadlock avoidance in store-and-forward networks, in: Jerusalem Conference on Information Technology, 1978, pp. 577–581.