Easily rendering token-ring algorithms of distributed and parallel applications fault tolerant

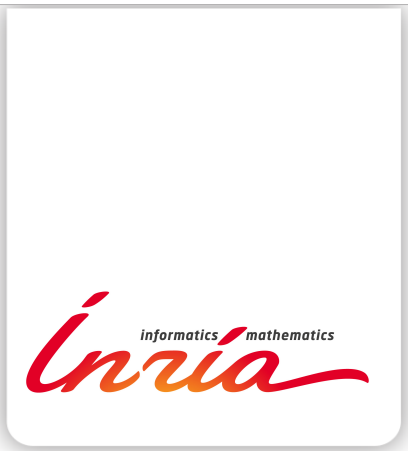
Luciana Arantes, Julien Sopena

Easily rendering token-ring algorithms of distributed and parallel applications fault tolerant

Luciana Arantes , Julien Sopena

RESEARCH REPORT
N° 8359
Septembre 2013
Project-Teams Regal

ISSN 0249-6399 ISRN INRIA/RR--8359--FR+ENG



Easily rendering token-ring algorithms of distributed and parallel applications fault tolerant

Luciana Arantes *, Julien Sopena*

Project-Teams Regal

Research Report n° 8359 — Septembre 2013 — 23 pages

Abstract: We propose in this paper a new algorithm that, when called by existing token ring-based algorithms of parallel and distributed applications, easily renders the token tolerant to losses in presence of node crashes. At most k consecutive node crashes are tolerated in the ring. Our algorithm scales very well since a node monitors the liveness of at most k other nodes and neither a global election algorithm nor broadcast primitives are used to regenerate a new token. It is thus very effective in terms of latency cost. Finally, a study of the probability of having at most k consecutive node crashes in the presence of f failures and a discussion of how to extend our algorithm to other logical topologies are also presented.

Key-words: distributed algorithm, fault tolerance, token-based, scalability

* LIP6 - Université de Paris 6 - INRIA Rocquencourt 4, Place Jussieu 75252 Paris Cedex 05, France.

RESEARCH CENTRE
PARIS – ROCQUENCOURT

Domaine de Voluceau, - Rocquencourt
B.P. 105 - 78153 Le Chesnay Cedex

Résumé : Pas de résumé

Mots-clés : Pas de motclef

1 Introduction

Numerous distributed and parallel applications use ring-based algorithms. The latter are based on the principle of the uniqueness of a token which travels along a logical ring that comprises all the nodes of the system. Examples of such algorithms are mutual exclusion [1], termination detection [2] [3], leader election [4], [5], [6], moving sequencer or privilege-based total order broadcast algorithms [7], group membership [8], etc.

In a distributed token-based algorithm, at any moment, there is at most one node that holds the token (*safety property*) and the token holder eventually grants it to its successor in the logical ring, i.e., the token circulates among all nodes and eventually every node will hold it (*liveness property*). However, in case of failure, the detection of the token loss and the regeneration of a new token can be quite costly and not very effective if scalability is taken into account. Therefore, for performance sake, fault tolerant token-based algorithms should not involve all nodes of the system for electing a new token holder in case of failure of the previous one; it should avoid monitoring the liveness of all of nodes, using reliable broadcast communication primitives, and reconstructing the logical ring, since they are costly operations. A third point is that, if the token holds some information such as in [2] [7], the latter should be easily restored when the new token is regenerated.

Therefore, aiming at overcoming scalability, performance, and loss of token information issues discussed above, we propose in this paper a new approach that renders the token fault tolerant to node crashes and which can be easily "plugged" in existing token ring distributed algorithm applications.

By keeping temporal copies of the token and without using reliable broadcast, our algorithm tolerates at most k consecutive faults in relation to the ring order which defines the sequence of token holders and satisfies the points previously raised: it is scalable since a node monitors at most k other nodes and it starts (respectively, stops) monitoring the token when the latter is in (respectively, out of) its neighborhood; there is no need to regenerate the token by using a global election algorithm nor to reconstruct the ring; finally, the information the token retains can be easier restored if the token holder fails.

Our token regeneration strategy is very effective in terms of latency when compared to existing approaches which usually monitor all sites, involve all sites for the detection of the loss of the token, and require a global election and/or a group membership protocol to chose a new token holder. In our approach, the token is regenerated instantaneously: when the failure of the token holder is detected, one of the site knows that it has the right to regenerate the token without needing to communicate with other sites whatsoever. Notice that our algorithm do not use reliable broadcast, which is very costly in terms of latency. At the same time, we should emphasize that it is not algorithmically trivial to provide such an "instantaneous regeneration". To this end, our algorithm keeps a set invariants, which are ensured, at any given time. Interestingly that the latter are related both to the current state of the nodes and the messages in transit.

We also point out that our algorithm can tolerate more than k faults, provided that the latter are not consecutive. Furthermore, we present in this paper a theoretical study of the value of k (see section 4) that show that the assumption of k consecutive failure is quite realistic. For instance, in a system with 10000 it is just necessary to have $k = 20$ in order to tolerate, with a probability closer to 100%, 50% of the nodes that crash.

A last interesting remark is that, combined with any protocol which maintains a virtual token circulation on top of a graph (e.g. a depth-first token circulation protocol), our approach can be applied on any arbitrary graph which corresponds to a given topology. Moreover, when failures (at most k consecutive nodes in the corresponding logical ring) take place, the graph does not need to be reconstructed.

The paper is organized as follows. Section 2 defines the computation model and presents the fault-tolerant token "plug-in" algorithm. Some examples of well-known distributed algorithms that could call the API functions offered by our algorithm in order to render the token tolerant to losses as well as a use case are described in section 3. Section 4 presents a theoretical study of the value of k in relation to the number of nodes and total number of failures while section 5 discusses how our approach could be extended to a general topology and its constraints. Section 6 comments on related work. Finally, section 7 concludes the paper.

2 Reliable Token-based Algorithm

We consider a distributed system consisting of a finite set Π of $N > 1$ sites, namely $\Pi = \{S_0, S_1, \ldots, S_{N-1}\}$ which communicate only by message passing. Communication channels are reliable, but messages might be delivered out of order. There is one process per node. Hence, the words node, site and process are interchangeable.

The N nodes are logically organized in a ring and each node knows the identity of its respective $k+1$ predecessors and $k+1$ successors with which it can communicate. To avoid complicated notations, we note that the successor of node S_i is S_{i+1} and not $S_{(i+1)\%N}$.

Initially, one node holds the token which circulates in a given direction. We denote S_T the current token holder which always grants the token to its direct successor within a bounded delay.

In order to ensure the uniqueness of the token, false suspicion of node failure can never take place [9]. i.e., the failure detection mechanism of our algorithm should eventually detect failures but never makes a mistake about them. Therefore, our algorithm executes either on top of synchronous systems or asynchronous ones enriched with a perfect failure detector.

Nodes can fail by crashing, and this crash is permanent. A *correct* process is a process that does not crash during a run, otherwise, it is *faulty*. Let k ($k < N-1$), which is known to every process, be the maximum number of consecutive faulty processes in the ring.

Our algorithm offers three API functions to the application: *SafeSendToken*, *SafeReceiveToken*, and *UpdateToken*. The application must then replace its original function to send the token to its successor in the ring and the one to receive the token from its predecessor by the functions *SafeSendToken* and *SafeReceiveToken* respectively. The function *UpdateToken* can be used by the application whenever, in case of failure, it is necessary to update the information kept by the token.

Basically, the idea of our approach is to avoid the loss of the token due to crashes by maintaining temporal backup copies. Whenever the node that holds the token grants it to its direct successor by sending a $\langle TOKEN \rangle$ message, the former also transparently sends a copy of such a message to k successors of the latter in the ring. Upon reception of a $\langle TOKEN \rangle$ message, a node starts monitoring the liveness of a subset of nodes that received the same copy of the message. A node has the exclusive right to use the token when either the received $\langle TOKEN \rangle$ message informs that it is the next node that can do it, or when the monitoring mechanism of this node informs that all nodes whose liveness it controls have crashed.

One could argue that our solution implies k additional messages per $\langle TOKEN \rangle$ emission. It does indeed, yet its message complexity remains equal to that of the original algorithm. In addition, our monitoring mechanism for failure detection presents a much lower cost in terms of messages when compared to other solutions where a site usually monitors all the others.

Note that it may happen that at a given moment there is no token in the system because the new one has not been regenerated yet. In our algorithm, a copy of the token is not considered a token as long as it does not replace the original one.

2.1 Variables and Messages

Node S_i keeps the local variables $count_{S_i}$ and $token_{S_i}$. The former is used to avoid that a site considers an old $\langle TOKEN \rangle$ message as a valid one while the latter controls if S_i holds the token, a copy of it, or neither of them. The values *REAL*, *BACKUP*, or *NONE* can be assigned to $token_{S_i}$: (1) $token_{S_i}$ is equal to *REAL* whenever S_i has the right to use the token (token holder). At a given time t, only one process has its token variable equal to *REAL* which ensures the uniqueness of the token; (2) the value *BACKUP* is assigned to $token_{S_i}$ if it is one of the k direct successor nodes of S_T and it maintains a valid copy of the token. Notice that if S_i does not crash, $token_{S_i}$ will eventually change from *BACKUP* to *REAL*; (3) the $token_{S_i}$ variable has value *NONE* when S_i holds neither a *REAL* nor a *BACKUP* token.

The following two sets are also handled by S_i:

- $\mathcal{D}_{S_i}$ (Detection set): This set includes both the processes that S_i must monitor, the liveness and S_i itself. It is composed of $\{S_T \dots S_i\}$, i.e., the set of processes between S_T and S_i in the ring increasing order, including both sites. It thus has at most $k+1$ sites.

- $\mathcal{F}_{S_i}$ (Faulty set): The set of processes that S_i has detected to be faulty.

If the value of $token_{S_i}$ is either equal to *REAL* or *BACKUP*, then $\mathcal{D}_{S_i} \neq \emptyset$. Remark that the $k+1$ detection sets are constructed in such a way that they are different from each other but have nested intersections. S_T is present in all detection sets. The advantage of such a construction is the lower cost in terms of messages when compared to an all-to-all monitoring of $k+1$ nodes. Furthermore, the election of the node that will hold the new token in case of node failures does not require any extra message.

Figure 1 shows $N = 12$ sites organized in a ring with k=3. In Figure 1(a), process S_4 is the token holder and processes S_5, S_6, S_7 keep copies of the token while in Figure 1(b), we can observe the detection set $\mathcal{D}$ of processes $S_4, \dots, S_7$.

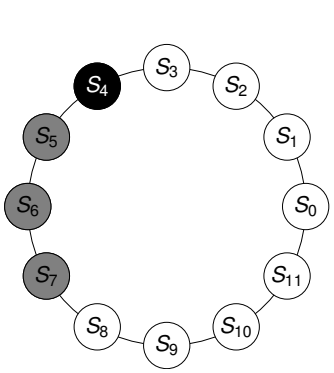
A $\langle TOKEN \rangle$ message contains the identification of the successor of the message sender as well as the current value of the *count* variable of this node. Moreover, the $\langle TOKEN \rangle$ message might contain application data.

2.2 Algorithm Description

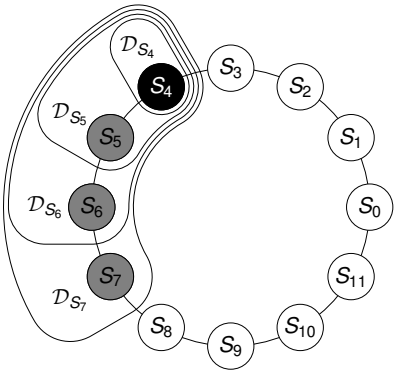
Algorithm 2 describes our fault tolerant token algorithm for a site S_i on ring topologies.

For the sake of simplicity, the liveness checking of processes that belong to $\mathcal{D}_{S_i}$ and the detection of node crashes are not shown in the pseudo code of the algorithms. Whenever the set $\mathcal{D}_{S_i}$ is updated, the function $updateDetection(\mathcal{D}_{S_i})$ stops monitoring the liveness of the processes in the previous $\mathcal{D}_{S_i}$ and starts monitoring the liveness of the processes in the new $\mathcal{D}_{S_i}$. On the other hand, when the crash of S_j is detected, a *Suspected* event is generated to S_i. Note that there are no false suspicions. A second remark is that S_i includes itself in $\mathcal{D}_{S_i}$ not for liveness checking reasons but just to easily conclude that it holds the *REAL* token when all the predecessor nodes that it monitors have crashed. A third point is that, in terms of implementation of the monitoring mechanism, it would not be necessary that S_i monitors all the sites of its $\mathcal{D}_{S_i}$ at the same time. It could just monitor the closest predecessor in its $\mathcal{D}_{S_i}$ that it has not detected to be faulty. If the latter crashes, it then starts monitoring the predecessor of this site that also belongs to its $\mathcal{D}$.

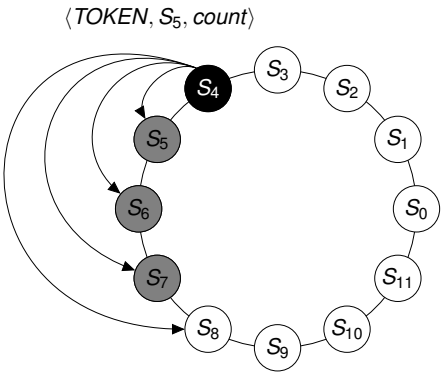
In the initialization phase (lines 1-1), we assume that S_0 holds the token. Hence, each process $S_i \in \{S_1, \dots S_k\}$ has a copy of the token (token = *BACKUP*) and initializes its respective $\mathcal{D}$ to $\{S_0, \dots S_i\}$. It then starts checking the liveness of the processes other than itself, (line 1) which



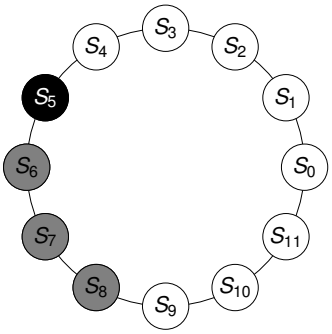
(a) The token holder and the k "backups"



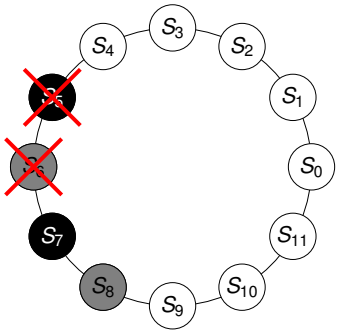
(b) Detection sets



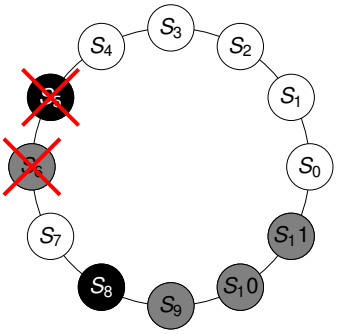
(c) $k+1$ unicast sending of token and backup copies



(d) The new token holder and the k "backups"



(e) Crash detection



(f) The holder and the k "backups" after a release from S_7

Figure 1: Ring-based reliable token algorithm with $k = 3$

```
1  /* D: Detection set                                   */
2  /* F: Faulty set                                      */

3  Initialisation ()
4      count ← 0
5      F ← { }
6      case i = 0
7          token ← REAL
8          D ← {S_0}
9      case 0 < i ≤ k
10         token ← BACKUP
11         D ← {S_0, .., S_i}
12     case k < i
13         token ← NONE
14         D ← { }
15     UpdateDetection(D)

16 SafeSendToken (⟨TOKEN⟩) to S_{i+1}
17     count ← count + 1
18     Send ⟨TOKEN,S_{i+1},count⟩ to {S_{i+1}, .., S_{i+k+1}}
19     token ← NONE
20     D ← { }
21     UpdateDetection(D)

22 SafeReceiveToken (⟨TOKEN,S_T,count_recv⟩) from S_j
23     if count < count_recv then
24         count ← count_recv
25         D ← {S_T, .., S_i}
26         if S_i = S_T then
27             token ← REAL
28             DeliverToken(⟨TOKEN⟩)
29         else if D/F = {S_i} then
30             UseBackup()
31         else
32             token ← BACKUP
33         updateDetection(D)

34 ReceiveSuspected (S_j)
35     F ← F ∪ S_j
36     if D/F = {S_i} then
37         UseBackup()

38 UseBackup ()
39     count ← count + ( #(D) - 1 )
40     token ← REAL
41     D ← {S_i}
42     UpdateToken(⟨TOKEN⟩)
43     DeliverToken(⟨TOKEN⟩)
```


Figure 2: Ring-based fault tolerant token algorithm

belong to its $\mathcal{D}$ set. All the other processes of the system do not keep any valid copy of the token at all ($token = NONE$).

The function $SafeSendToken$ allows S_i to send the token to its successor S_{i+1} and a copy of it to $\{S_{i+2} \dots S_{i+k+1}\}$, signaling that the next owner of the $REAL$ token is the site S_{i+1} (See Figures 1(c) and 1(d)). Notice that the function uses unicast sending primitive and not necessarily needs to respect the order of $\{S_{i+1} \dots S_{i+k+1}\}$. Since S_i does not keep a copy of the token anymore ($token = NONE$) it stops monitoring other processes (line 1). Note that we assume that S_i can call the function $SafeSendToken$ only if it holds the $REAL$ token and once it does, it grants the token.

Upon the reception of a $TOKEN$ message, which is not an old message, from S_j ($SafeReceiveToken\langle TOKEN, S_T, count_{recv}\rangle$), S_i updates its variable $count_{S_i}$ with the value received in the $TOKEN$ message (line 1). It then sets its $\mathcal{D}_{S_i}$ with nodes between S_T and itself including both sites (line 1). Then, (1) if S_i is the next holder of the token, it sets its $token_{S_i}$ variable to $REAL$ and delivers the token to the application (lines 1-1); (2) if S_i detects that all the nodes it current monitors have crashed, it becomes the site that holds the $REAL$ token by calling function $UseBackup()$ (lines 1-1. See Figure 1(e)). It delivers in the same way that in (1) the token to the application. When it further releases the token, it will send it to its $k + 1$ successors, as previously explained (See Figure 1(f)); (3) otherwise S_i sets its $token_{S_i}$ variable to $BACKUP$ (line 1). In all cases, it updates the set of sites it must monitor (line 1).

When S_i is informed ($ReceiveSuspected(S_j)$, line 1) of the crash of S_j, one of the nodes it monitors, it updates ($\mathcal{F}_{S_i}$), the set of faulty nodes of which it is aware (line 1). If all the processes whose liveness it monitors have crashed, S_i calls the function $UseBackup()$ in order to become the new holder of the $REAL$ token.

Function $UseBackup()$ updates $count_{S_i}$ by adding the number of nodes that S_i has detected as crashed (line 1), which ensures that old pending token messages that may arrive later to S_i will be discarded. It also changes S_i's $token$ variable from $BACKUP$ to $REAL$ (line 1). Before delivering the token to the application, S_i can update the information associated to the token (line 1), if necessary (e.g., Misra's detection termination algorithm [2]). Note that the token regeneration is very effective since it basically consists of considering one of the backup copies as the $REAL$ token.

Due to lack of space the formal proof of the algorithm correctness is not presented in the article itself but can be found in the appendix[1].

3 Examples of applications

In this section we discuss how some algorithms found in the literature, which are based on the circulation and uniqueness of a token (or objects that are seen as tokens), can call the functions offered by our API in order to become tolerant to token (objects) losses in the presence of node failures. For all the examples, we consider that it is possible to implement a monitoring node failure mechanism which never makes a false suspicions.

3.1 Mutual exclusion

Our algorithm naturally adapts itself to Le Lann's mutual exclusion algorithm [1] where the exclusive access to a shared resource by a site depends on the possession of the single token, which circulates over all sites. Whenever a process receives the token, if it wants to access the

[1] If the paper is accepted the proof will be available in an INRIA Technical Report which will be referenced by the article.

shared resource, it does so; otherwise it sends the token to its successor. Upon releasing the access to the shared resource, a site also sends the token to its successor. In order to ensure the correct execution of the algorithm in case of node failures, each process just needs to call the $SafeSendToken$ and $SafeReceiveToken$ functions offered by our communication API for sending and receiving the token respectively.

In Le Lann's algorithm, the token keeps circulating on the ring even when no node requires it. Martin's algorithm [10] removes this constraint by including requests in Le Lann's algorithm: requests for the token move in one direction while the token moves in the opposite direction. The holder of the token keeps it as long as it does not receive a request. In order to render the algorithm tolerant to failures neither the token nor the requests can be lost in presence of node failures. However, each node request can be seen as a single object like the token ("request-object"). The functions $SafeSendToken$ and $SafeReceiveToken$ can thus be called to render each request tolerant to loss. Every request can be replaced by a $\langle TOKEN \rangle$ message which has the identification, S_q, of the node that issued the request. Requests are in this way uniquely identified by S_q. Nevertheless, we should point out that the initialisation function of our algorithm ensures the presence of $k+1$ copies of the token but not of the requests; in order to guarantee the circulation of the $k+1$ copies of the request issued by S_q, it would be necessary to impose that S_q will not fail before sending the $k+1$ copies of the "request-object" ($\langle TOKEN \rangle$) message. On the other hand, the loss of S_q's request is not actually a problem in this case since it concerns a request of a faulty node. Hence, the liveness of the algorithm is not violated. A final remark is that since the maximum number of pending requests is $N-1$ for a ring with N sites, our algorithm must control the circulation and uniqueness of at most $N-1$ requests (tokens). In other words, the number of "request-object" ($\langle TOKEN \rangle$) messages is bounded.

3.2 Termination Detection

Some algorithms that detect the termination of distributed applications [3] [2] consider that the N nodes on top of which the application runs are organized in a logical ring. The guiding principle of such algorithms is to verify that all the N processes are in the *passive* state after a complet round of the token. A process becomes passive when it finishes its execution. However, the reception of an application message by a passive process makes it actif again. In Misra's algorithm [2] nodes compose a unidirectional logical ring and messages are received by a node in FIFO order. The token, which circulates among all nodes, has the variable *passive_ count* that sums up the number of nodes that the token crossed which is in passive state upon reception of the token. A node that holds the token sends it to its sucessor only when becoming passive. If the latter was already passive when it receives the token, it adds one to *passive_ count*; otherwise it restarts *passive_ count* to 1.

The termination of the application is detected when all the nodes are passive after one round of the token ($passive\_count = N$ for each node). This algorithm can use the functions $SafeSendToken$ and $SafeReceiveToken$ to ensure the circulation of the token in case of failures. A faulty node can be viewed as a passive one. Nevertheless, if the holder of the $REAL$ token did not receive it from its predecessor (crash of one or more consecutive nodes), the token's *passive_ count* variable must be updated before being delivered to the termina-

tion detection algorithm. The *UpdateToken*() function would then have the following code:

```
44 UpdateToken (⟨TOKEN⟩)
45 |  if (⟨TOKEN⟩.passive_count ← ⟨TOKEN⟩.passive_count + (#(D) − 1)) > N
   |  then
46 |  |  ⟨TOKEN⟩.passive_count ← N
```

Note that we consider that the node that initiates the detection algorithm can fail only after sending the $k+1$ copies of the $\langle TOKEN \rangle$ message.

3.3 Leader Election

Several authors [4], [5], [6], etc. have proposed algorithms for the leader election problem for nodes organized in a logical ring. A node wishing to become the leader sends a claim message ($\langle TOKEN \rangle$) which will travel along the ring according to the comparison and transmission rules of the algorithm. For instance, in Chang and Roberts' algorithm [4], a claim message is transmitted from node to node in the ring until it is received by a node which is a better leader candidate. A node considers itself the new leader when it receives its own claim message.

In order to guarantee the correct execution of Chang and Roberts' algorithm despite node failures, the circulation and uniqueness of the claim messages should be ensured. However, similarly to the requests of Martins's mutual exclusion algorithm, a claim message could be seen as a single object that travels along the ring. In other words, the application can use the functions offered by our algorithm to render each claim message tolerant to failure: a $\langle TOKEN \rangle$ message which contains the identification, S_{claim}, of the leader candidate. Since for a ring with N sites, the maximum number of pending claims is N, our algorithm must control the circulation and uniqueness of at most N "claim-objects" (tokens).

Our algorithm guarantees that in the presence of at most k consecutive failures, the leader election algorithm ends (if and only if one of the candidates sent $k+1$ copies of the claim message). It is worth mentioning that the elected leader is not necessarily a non faulty process. On the other hand, we could offer an *UpdateToken* function to this end. Node S_j becomes leader upon receiving its own claim message ($\langle TOKEN \rangle$ message where $S_{claim} = S_j$). The latter was sent when the function $SafeSendToken$ was called. Consequently, the message is also received by the successors of S_j, that, therefore, monitor S_j. If S_j fails, the function *UpdateToken* is executed by S_i, the non faulty successor of S_j that has a non empty detection set. If S_i had previously registered S_j as the leader (*currenLeader*), it knows that the leader has crashed. Hence, in this case, S_j can be replaced by S_i in the corresponding $\langle TOKEN \rangle$ (claim) message in order to ensure the election of a non faulty node. Receiving its own claim, S_i becomes the new leader. The code of the *UpdateToken* function is thus:

```
47 UpdateToken (⟨TOKEN⟩)
48 |  if currentLeader = S_claim of ⟨TOKEN⟩ then
49 |  |  set S_i as S_claim in ⟨TOKEN⟩
```

3.4 Use Case: TranspeerToken

In [11], we have proposed TranspeerToken, a system for managing transactions for WEB2.0 applications, which is based on the existing Transpeer system [12]. Contrarily to the latter,

TranspeerToken is able to manage peak load of transaction requests efficiently, i.e., when most of transactions try to access the same data concurrently.

The Transpeer system accepts transactions issued from Web application clients. To this end, it keeps several transaction routing nodes, denoted TMs, that route each transaction request to the most suitable data server to process it. In order to serialize transactions that operate on possibly intersecting datasets, a common distributed directory stores the history of transactions (metadata) performed on every dataset. Although the common directory is distributed, transaction requests reference a single node per dataset which can thus suffers from contention when several concurrent clients attempt to access the same dataset. In other words, Transpeer fails to cope with peak load and can, therefore, fails to process transactions when a TM receives the majority of the request load.

We have leverage the above bottleneck problem by organizing the TMs into a logical ring and replacing the shared directory by our token-ring algorithm. Each token is associated to a metadata, carried around the TMs ring, similarly to the approach proposed for the requests of Martin's mutual exclusion algorithm (see section 3.1). Hence, TranspeerToken avoids locking the metadata stored on a node of the common distributed directory which would induce contention. Furthermore, as our token-ring algorithm tolerates the failure of $k = 5$ consecutive nodes of the ring, TranspeerToken guarantees that transactions requests are not lost in case of failure.

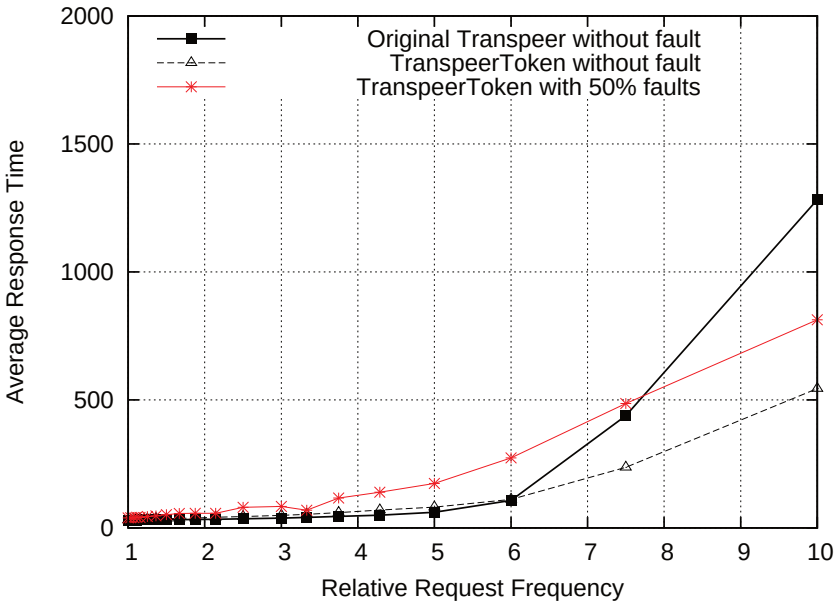


Figure 3: Fault tolerance overhead

Evaluation performance results were quite promising. For instance, Figure 3 shows that TranspeerToken tolerates better peek load than Transpeer even in presence of failures. The experiment consisted of processing 500 transactions. It started with 20 TMs; then 10 TMs fail upon receiving their 5th transaction. The peak load (transaction request frequency) varied relatively from 1 to 10 times the initial load. We have then measured the maximum response time of the transactions for both systems without failure as well as TranspeerToken with 50% of nodes that fail, such as it did no more than 5 consecutive failures. We observe that, although tolerating faults comes at a cost (approx. 40% overhead on the average response time), TranspeerToken

$$\forall k > 0,\ \forall f \leq N,\ W_k(f,N) = \begin{cases} 1 & \text{If } N = f \wedge f \leq k \\ 0 & \text{If } N = f \wedge f > k \\ n & \text{If } N - 1 = f \wedge f \leq k \\ 0 & \text{If } N - 1 = f \wedge f > k \\ \sum_{i=0}^{min(k,f)} W_k(f-i, N-i-1) & \text{Else} \end{cases}$$

Figure 4: $\boldsymbol{W_k(f,N)}$: the number of possible distributions of f failures in a ring of N sites such that there are at most k consecutive failures.

remains faster than Transpeer for high peak load (when load exceeds seven times the initial load). Note that during one run, the transaction response time keeps growing because of the peak load behavior.

More details about both systems as well as other evaluation experiments results can be found in [11].

4 Theoretical study of the value of k

Our algorithm is based on the assumption that there are at most k consecutive failures in the ring. It is thus interesting to study the relation between the value of k, the total number N of nodes of the ring and the number f of failures. In other words, a discussion of the following probability:

$P_k(f,N)$: Probability of having at most k consecutive node crashes among the N nodes that compose the ring in presence of f failures.

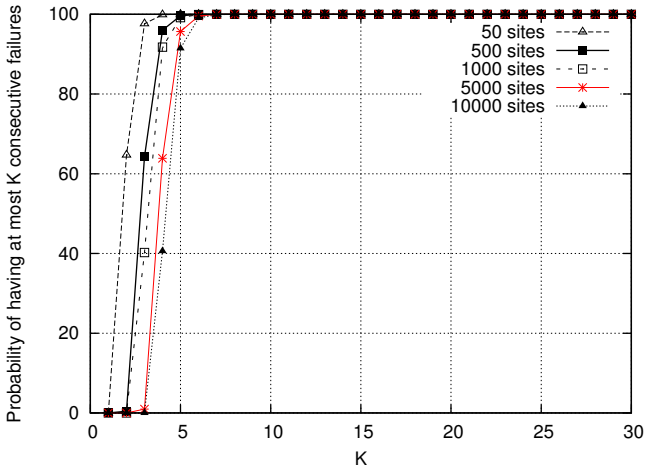
Due to lack of space, a detailed description of how this probability can be calculated is not presented in this article. Basically, its calculation consists of distributing the f failures with regard to S_0 taking into account its state, i.e., either S_0 is correct or it belongs to a sequence of i consecutive failure nodes ($i \leq k$). In both cases, the remaining failures (respectively f or $f-i$) must be distributed in the rest of the ring, composed of nodes other than the former. Notice that with this approach, such a distribution is analogous of distributing the failures in a chain (queue) of sites. Let us consider the position of the first non faulty node[2] among the first k ones of the chain, i.e., a sequence of i failures ($0 \leq i \leq k$) followed by a non faulty site. By recurrence, we can then distribute the $f - i$ failures among the $N - i - 1$ sites of the chain. It is worth emphasizing that, contrarily to a ring, this recurrence is possible in the case of a chain since the crashes at the beginning of it do not join the ones at the end of it; otherwise, a sequence of $k+1$ consecutive failures might be formed.

According to the previous enumeration, Figure 4 shows $\boldsymbol{W_k(f,N)}$: the number of possible distributions of f failures in a ring of N sites such that there are at most k consecutive failures.

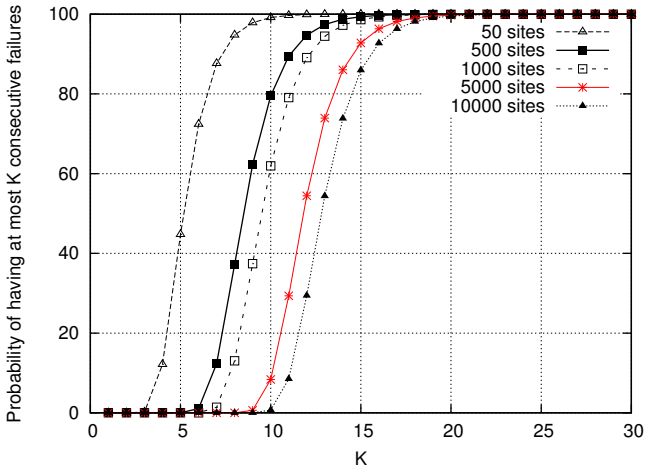
In order to study how to size k, we have programmed the above function by inferring that $P_k(f,N) = W_k(f,N)/C_N^f$ Figures 5(a) and 5(b) show, for 10% and 50% of nodes that crash respectively, the probability to stand up such failures for queues with 50, 500, 1000, 5000 and 10000 nodes.

In both figures, we can observe that the probabilities are functions that go towards a "resilience threshold". In Figure 5(a), there is a small increase of this threshold when the number of nodes

[2]the case where there is no correct node ($N = f$) is not neglected in the calculation.



(a) More than 10% of failures.



(b) More than 50% of failures.

Figure 5: Probability study of having at most k consecutive node crashes in a ring.

increases. For instance, the threshold for a system composed of 10000 nodes is $k = 8$. On the other hand, when the number of failures increases to 50% (Figure 5(b)) the increasing of the threshold is more significant when the number of nodes increases. Nevertheless, in a system with 10000 it is just necessary to have $k = 20$ in order to tolerate, with a probability closer to 100%, 50% of the nodes that crash.

In summary, such a study shows that with a small value for k, our approach can tolerate a much greater number of node crashes, specially in large systems.

We should point out that this study does not consider that the ring can be reconstructed. If it was the case, the stand up to failures would be greater. A second comment is that even if the probability of having more than k consecutive failures with the threshold is extremely small, such a probability is not null. Thus, if it happens, we could think of offering a function, rarely called, which would be able to regenerate the new token (e.g. a global election algorithm).

5 Generalizing our algorithm

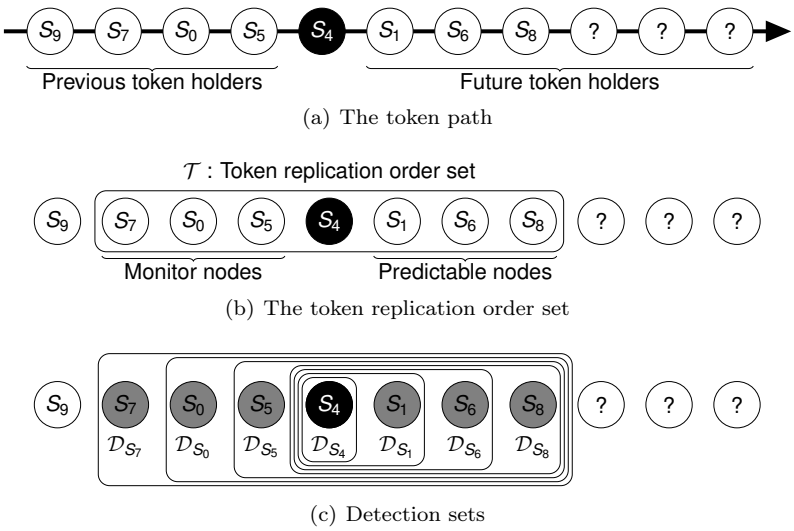
Our algorithm is based on a logical ring topology but we could think of extending its k-copies principle to any topology aiming at rendering other token-based algorithms, whose sites are not logically organized into a ring, fault tolerant. However, the concept of sending $k + 1$ copies of the token, in this case, confronts two problems:

- It is necessary to ensure that the k successors of the token holder are all distinct which it is not always true, contrarily to the ring topology. For instance, in a tree-based algorithm (e.g. Raymond's mutual exclusion algorithm [13]) the token does not always travel between k distinct successive nodes of the logical tree;

- In our ring-base approach, the token holder knows which are the k nodes that should receive the token after it and this is not always possible for some algorithms as, for instance, algorithms where the token circulates only between nodes that have requested it such as in Suzuki and Kasami's mutual exclusion algorithm [14].

The solution we propose for the above problems is that the following: token holder chooses the set of its successors by exploiting as much as possible the knowledge that it has about the future path of the token (e.g. set of nodes from which the token holder has received a request that has not been satisfied yet as in [14]). See Figure 6(a), where S_4 is the current token holder. Additionally, if the number of nodes of this set, whose nodes are all distinct, is smaller than $k+1$ (the token holder includes itself in this set), it will be completed with nodes which have held the token recently. The latter are just used to control the liveness of the other nodes of this set, even if they might become a token holder if all the nodes that they monitor crash. In Figure 6(b), we respectively denote the first and the second type of nodes *predictable* and *monitor*. The mentioned set is denoted $\mathcal{T}$ and $k = 6$. The respective detection set of each of the nodes of $\mathcal{T}$ is given in Figure 6(c). Remark that, similarly to the ring topology, the $k + 1$ detection sets are different from each other and have nested intersections. Each node of a detection set $\mathcal{D}_i$ which does not intersect with the nodes of the detection set nested by $\mathcal{D}_i$ monitors these nodes. For instance, S_8 monitors S_6, S_1, and S_4 while S_5 monitors S_4, S_1, S_6, and S_8.

Before sending the token in the $SafeSendToken$ function, S_k, the token holder, must update its $\mathcal{T}$ set by replacing the *monitor* nodes by *predictable* ones as much as possible. Nodes are replaced till either S_k is removed from $\mathcal{T}_{S_i}$ or there are no more sites in the predictable path. Remark that it might happen that S_k is not removed from $\mathcal{T}$ if there are not enough new *predictable* nodes. In this case, S_k will become a *monitor* node.



(a) The token path

(b) The token replication order set

(c) Detection sets

Figure 6: Generalized approach with $k = 6$

6 Related work

Several authors have proposed fault-tolerant extensions to token-based mutual exclusion algorithms [15] [16] [17]. However, they usually adopt non scalable solutions to regenerate the token such as an election algorithm as in Chang et al.'s work [17] or require that the site that detected the loss of the token receives a positive acknowledgement from the other sites before regenerating a new token [15] [16].

Misra [2] proposes in his termination detection article to adapt the algorithm in order to detect the loss of the token and regenerate a new one for ring topologies. The author argues that token of loss is similar to application termination if only token messages are considered. Similarly to our solution he uses the concept of backup/real tokens: there are two symmetrical tokens in the system but one is the backup. However, the algorithm works only if a token is not lost within the round of the other token's loss. Wu et al. [18] extended Misra's work to offer a mutual exclusion algorithm for MANETs that tolerates token losses.

In [19], Mueller presents an interesting mechanism to support fault tolerance for token-based synchronization protocols. A logical ring is used to detect a node failure and, if necessary, elect a new token holder. However, the detection and handling of failures are not transparent to the application as in our approach. The application must be modified to add a monitoring mechanism which then calls the author's fault-tolerant mechanism when a failure is suspected. A second difference with regard of our approach is that when a new token is regenerated, the information kept by it is not preserved or restored as in our solution.

Many total order broadcast algorithms adopt a token-based ring mechanism for ordering message [7]. The token circulates among all nodes or a subset of them. Some of these algorithms such as [20] and [8] tolerate failures but involve costly global mechanism such as a reformation phase [20] or a membership protocol [8] for redefining the logical ring and electing a new token holder.

A work close to ours is R. Ekwall et al.'s [21]: nodes are organized in a logical ring and the token is sent to the $f + 1$ successors, where f is the maximum number of failures. However, the

goal of the authors' approach differs from ours. They consider an asynchronous system on top of which they want to build a consensus algorithm based on token. A node expects to receive the token from its predecessor. On the other hand, if the former suspects that the latter has failed, it waits the token from any of the f predecessors. Contrarily to our solution, failure detection is not perfect and the uniqueness of the token is not ensured. Furthermore, the proposal of our algorithm is that it can be exploited by ring-based algorithms which need token uniqueness in presence of failures. This portability is not offered by the authors' approach whose aims are to provide a consensus and atomic broadcast algorithms. A last remark is that our algorithm tolerates k consecutive failures, i.e., it might be that the total number of failures is greater than k.

7 Conclusion

We have presented in this article an algorithm that avoids the loss of the token by maintaining temporal backup copies of the token on k consecutive nodes. Its functions can be easily called by existing token ring-based algorithms in order to make them tolerant to k consecutive failures. If the token holder fails, the token regeneration is almost instantaneous and quit inexpensive due to the mentioned backup copies and the invariants kept by the algorithm. One key feature of our solution is that it is scalable since it depends on k and not on N. Furthermore, the information that the token retains is not lost when a new one is regenerated. We have also presented a study of the probability of having at most k consecutive node crashes in the presence of f failures in a ring with N nodes which shows that the value of k to tolerate the f failures with a probability closer to 100% is quite small. Finally, we have discusses the challenges of extending our algorithm to other logical topologies and have proposed a solution that keeps the good properties of our algorithm.

References

[1] G. L. Lann, "Distributed systems - towards a formal approach," in *IFIP Congress*, 1977, pp. 155–160.

[2] J. Misra, "Detecting termination of distributed computations using markers," in *PODC '83: Proceedings of the second annual ACM symposium on Principles of distributed computing*, 1983, pp. 290–294.

[3] D. E. W, W. H. J. Feijen, and A. J. M. van Gasteren, "Derivation of a termination detection algorithm for distributed computations," in *Proc. of the NATO Advanced Study Institute on Control flow and data flow: concepts of distributed programming*, 1986, pp. 507–512.

[4] E. Chang and R. Roberts, "An improved algorithm for decentralized extrema-finding in circular configurations of processes," *Commun. ACM*, vol. 22, no. 5, pp. 281–283, 1979.

[5] R. W. Franklin, "On an improved algorithm for decentralized extrema finding in circular configurations of processors," *Communications of the ACM*, vol. 25, no. 5, pp. 336–337, may 1982.

[6] G. L. Peterson, "An o(nlog n) unidirectional algorithm for the circular extrema problem," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 4, no. 4, pp. 758–762, october 1982.

[7] X. Défago, A. Schiper, and P. Urbán, "Total order broadcast and multicast algorithms: Taxonomy and survey," *ACM Comput. Surv.*, vol. 36, no. 4, pp. 372–421, 2004.

[8] Y. Amir, L. E. Moser, P. M. Melliar-Smith, D. A. Agarwal, and P. Ciarfella, "The totem single-ring ordering and membership protocol," *ACM Trans. Comput. Syst.*, vol. 13, no. 4, pp. 311–342, 1995.

[9] C. Delporte-Gallet, H. Fauconnier, R. Guerraoui, and P. Kouznetsov, "Mutual exclusion in asynchronous systems with failure detectors," *J. Parallel Distrib. Comput.*, vol. 65, no. 4, pp. 492–505, 2005.

[10] A. J. Martin, "Distributed mutual exclusion on a ring of processes," *Science of Computer Programming*, vol. 5, no. 3, pp. 265–276, 1985.

[11] L. Millet, M. Lorrillere, L. Arantes, S. Gançarski, H. Naacke, and J. Sopena, "Facing peak loads in a p2p transaction system," in *Proceedings of the First Workshop on P2P and Dependability*, 2012.

[12] I. Sarr, H. Naacke, and S. Gançarski, "Transpeer: Adaptive Distributed Transaction Monitoring for Web2.0 applications," in *ACM Symposium on Applied Computing (SAC DADS)*, 2010.

[13] K. Raymond, "A tree-based algorithm for distributed mutual exclusion," *ACM Transactions on Computer Systems (TOCS)*, vol. 7, no. 1, pp. 61–77, 1989.

[14] I. Suzuki and T. Kasami, "A distributed mutual exclusion algorithm," *ACM Transactions on Computer Systems (TOCS)*, vol. 3, no. 4, pp. 344–349, 1985.

[15] S. Nishio, K. F. Li, and E. G. Manning, "A resilient mutual exclusion algorithm for computer networks," *IEEE Trans. on Parallel and Distributed Systems*, vol. 1, no. 3, pp. 344–355, july 1990.

[16] D. Manivannan and M. Singhal, "An efficient fault-tolerant mutual exclusion algorithm for distributed systems," in *Int. Conf. on Parallel and Distributed Computing Systems*, 1994, pp. 525–530.

[17] I. Chang, M. Singhal, and M. T. Liu, "A fault tolerant algorithm for distributed mutual exclusion," in *Proceedings of the IEEE 9th Symposium on Reliable Distributed Systems*, 1990, pp. 146–154.

[18] W. Wu, J. Cao, and M. Raynal, "A dual-token-based fault tolerant mutual exclusion algorithm for manets," in *MSN*, 2007, pp. 572–583.

[19] F. Mueller, "Fault tolerance for token-based synchronization protocols," *Workshop on Fault-Tolerant Parallel and Distributed Systems, IEEE*, april 2001.

[20] J.-M. Chang and N. F. Maxemchuck, "Reliable broadcast protocols," *ACM Trans. Comput. Syst.*, vol. 2, no. 3, pp. 351–273, 1984.

[21] R. Ekwall, A.Schiper, and P. Urbán, "Token-based atomic broadcast using unreliable failure detectors," in *SRDS*, 2004, pp. 52–65.

8 Appendix: Proof

In order to prove the correctness of our algorithm ,we must show that the *safety* and *liveness* properties always hold.

We suppose that the original algorithm correctly calls the functions offered by our algorithm and verifies the *safety* and *liveness* properties.

Hyp. 1 *A site can call the SafeSendToken function provided it keeps the REAL token.*

We consider that time is discretized by *SafeSendToken*, *SafeReceiveToken*, and *UseBackup* function executions. Time t_0 is set by the execution of the *Initialisation* function. We denote $\mathcal{C}$ such a discrete-time. Notice that processes do not have access to $\mathcal{C}$ which is only introduced for the convenience of the proof presentation.

We denote $\mathcal{P}(\Pi)$ the power set of π and $\mathcal{F}(t) : \mathcal{C} \rightarrow \mathcal{P}(\Pi)$ a function which returns the set of faulty processes at t.

In order to express some variables in regard with $\mathcal{C}$, we define a set of functions. For a given time $t \in \mathcal{C}$ and a site S, each of these functions returns the value of a variable after being executed at time t. The functions, which respectively return the value of variables *token*, *count*, and $\mathcal{D}$ for site S at t, are the following:

$Token(S,t) : \ \pi \times \mathcal{C} \rightarrow \{NONE, BACKUP, REAL\}$

$Count(S,t) : \ \pi \times \mathcal{C} \rightarrow \mathbb{N}$

$\mathcal{D}(S,t) : \ \pi \times \mathcal{C} \rightarrow \mathcal{P}(\Pi)$

Finally, we define $\mathcal{M}(S_i,t)$ as the set of pending messages $\langle TOKEN, S_T, count_{recv} \rangle$ addressed to S_i such that $count_{recv} > count(S_i,t)$ at t and $\mathcal{M}(t)$ is the set of all $\mathcal{M}(S_i,t)$. We denote $\langle Pend_{REAL} \rangle$ the pending $\langle TOKEN \rangle$ message of $\mathcal{M}(t)$ such that $S_i = S_T$ and $\langle Pend_{BACKUP} \rangle$ the other pending $\langle TOKEN \rangle$ messages of $\mathcal{M}(t)$.

For helping the proof of our algorithm, we introduce three properties. The property $\boldsymbol{PSafetyCond}$ is a sufficient condition for proving the safety property. The two other properties, $\boldsymbol{PCount}$ and $\boldsymbol{PHolderMonitored}$ respectively ensure that the token holder has the greatest *count* and that the latter is always monitored respectively. These two properties must always hold in order to ensure the safety property after a crash. $\forall t \in \mathcal{C}$, the tree properties are thus defined as follows:

$\boldsymbol{PSafetyCond(t)}$: There exists at most one non faulty site S_i that either holds the *REAL* token or is the receiver of $\langle Pend_{REAL} \rangle$, i.e., a pending $\langle TOKEN \rangle$ message which informs that S_i is the new token holder.

If $PSafetyCond(t)$ holds, we denote:

- $\boldsymbol{Holder(t)}$**:** the non faulty site that satisfies $PSafetyCond(t)$ or the last faulty node which has verified $PSafetyCond$, i.e., when a site crashes, it is considered as $Holder(t)$ till a new one is regenerated.

- $\boldsymbol{HCount(t)}$**:** corresponds to $Count(Holder(t),t)$, if $Token(Holder(t),t) = REAL$, or equal to the *count* value of $\langle Pend_{REAL} \rangle$.

PHolderMonitored(t):

$$\forall S_i \in \pi/\mathcal{F}(t), \mathcal{D}(S_i,t) \neq \emptyset \Longrightarrow$$
$$Holder(t) \in \mathcal{D}(S_i,t) \quad (1)$$

$$\forall \langle TOKEN, S_T, count_{recv}\rangle \in \mathcal{M}(S_i,t),$$
$$Holder(t) \in \{S_T, .., S_i\} \quad (2)$$

$\forall t \in \mathcal{C}$, $PHolderMonitored(t)$.(1) states that the token holder is always monitored by those nodes whose detection set is not empty and $PHolderMonitored(t)$.(2) states that S_i will start monitoring the token holder when it receives $\langle TOKEN, S_T, count_{recv}\rangle$.

PCount(t):

$$\forall S_i \in \pi/\mathcal{F}(t), \mathcal{D}(S_i,t) \neq \emptyset \Longrightarrow$$
$$HCount(t) - Count(S_i,t) =$$
$$\#(\mathcal{D}(S_i,t)) - \#(\{Holder(t), .., S_i\}) \quad (3)$$

$$\forall \langle TOKEN, S_T, count_{recv}\rangle \in \mathcal{M}(S_i,t),$$
$$HCount(t) - count_{recv} = \#(\{S_T, .., Holder(t)\}) - 1 \quad (4)$$

This property states that, $\forall t \in \mathcal{C}$, the difference between the *count* of the token holder and the *count* of either those sites that have non empty detection set or the one included in the messages of $\mathcal{M}(t)$ is greater or equal to zero. In addition, such a difference depends on the position of $Holder(t)$ compared to either S_i or S_T (in the case of pending messages).

Since, in $PCount(t)$, the difference between $HCount(t)$ and any *count* value (of both non-empty $\mathcal{D}$ sites and pending messages) is greater or equal to zero, we can define the following corollary:

PMaxCount(t) (corollary of $PCount(t)$) : At t, the token holder has the greatest *count* value among all non faulty sites and all $\langle TOKEN\rangle$ pending messages.

We present now the lemmas that prove that the three above properties always hold $\forall t \in \mathcal{C}$ which then will help us prove by induction the safety of our algorithm.

For each of these lemmas, we distinguish the calls to *SafeReceiveToken*, *SafeSendToken* and *UseBackup* which discretize the time $\mathcal{C}$.

Lemma 1 $\forall t \in \mathcal{C}, PSafetyCond(t) \wedge PCount(t) \wedge PHolderMonitored(t) \implies PSafetyCond(t+1)$.

Proof. Suppose that $PSafetyCond$, $PCount$, and $PHolderMonitored$ be true at t. At $t+1$, site S_i can call one of the three following functions:

1. ***SafeReceiveToken*$\langle$*TOKEN, S_T, $count_{recv}$*$\rangle$:** if $\langle TOKEN, S_T, count_{recv}\rangle$ is a $\langle Pend_{BACKUP}\rangle$, $Token(S_i, t+1) = BACKUP$ (line 1) and $PSafetyCond$ continues to hold at $t+1$; otherwise since the message, $\langle Pend_{REAL}\rangle$, was pending at t, $PSafetyCond(t)$ ensures that S_i was the $Holder(t)$. Upon calling the function *SafeReceiveToken*, S_i will continue to be the token holder, i.e. $Token(S_i, t+1) = REAL$. Moreover, there exists no other $\langle Pend_{REAL}\rangle$ in the system. Hence, $PSafetyCond$ is true at $t+1$ with $Holder(t+1) = S_i$, whose identity, S_T, is in the messages.

$$\forall S_x \in \pi/\mathcal{F}(t), \mathcal{D}(S_x,t) \neq \emptyset \wedge PCount(t).(3)$$

$$\Longrightarrow \begin{cases} HCount(t) - count(S_x,t) = \#(\mathcal{D}(S_x,t)) - \#(\{Holder(t),..,S_x\}) \\ HCount(t) - count(S_i,t) = \#(\mathcal{D}(S_i,t)) - \#(\{Holder(t),..,S_i\}) \end{cases}$$

$$\underset{*^1}{\Longrightarrow} \begin{cases} HCount(t) - count(S_x,t) = \#(\mathcal{D}(S_x,t)) - (\#(\{Holder(t),..,S_i\}) + \#(\{S_i,..,S_x\}) - 1) \\ HCount(t) - count(S_i,t) = \#(\mathcal{D}(S_i,t)) - \#(\{Holder(t),..,S_i\}) \end{cases}$$

$$\Longrightarrow count(S_i,t) - count(S_x,t) = \#(\mathcal{D}(S_x,t)) - \#(\{S_i,..,S_x\}) + 1 - \#(\mathcal{D}(S_i,t))$$

$$\Longrightarrow (count(S_i,t) + \#(\mathcal{D}(S_i,t)) - 1) - count(S_x,t) = \#(\mathcal{D}(S_x,t)) - \#(\{S_i,..,S_x\})$$

$$\underset{*^2}{\Longrightarrow} count(S_i,t+1) - count(S_x,t) = \#(\mathcal{D}(S_x,t)) - \#(\{S_i,..,S_x\})$$

$$\underset{*^3}{\Longrightarrow} HCount(t+1) - count(S_x,t+1) = \#(\mathcal{D}(S_x,t+1)) - \#(\{Holder(t+1),..,S_x\})$$

$$\Longrightarrow PCount(t+1).(3)$$

Notes:
$*^1$: $S_i \in \{Holder(t),..,S_x\}$
$*^2$: $count(S_i,t+1) = count(S_i,t) + \#(\mathcal{D}(S_i,t)) - 1)$
$*^3$: $Holder(t+1) = S_i$

Figure 7: Proof of the counter property upon calling *UseBackup*.

2. ***SafeSendToken*:** In this case, Hyp. 1 implies that S_i must hold a *REAL* token. Moreover, $PSafetyCond(t)$ ensures that there exists one site ($Holder(t)$) with a *REAL* token. Hence, $Holder(t) = S_i$. Since $PMaxCount$ holds at t, there is no $\langle Pend_{REAL}\rangle \in \mathcal{M}(t)$. Finally, *SafeSendToken* also guarantees that $Token(S_i, t+1) = NONE$ (line 1). Thus, $PSafetyCond$ is true at $t+1$ and S_{i+1} is the only token holder at $t+1$ ($Holder(t+1)$).

3. ***UseBackup*:** Lines 1 and 1 guarantee that S_i only regenerates a new token when all the sites it monitors, excepting itself, are faulty. Moreover, since $PHolderMonitored$ holds at t, $Holder(t)$ belongs to the detection set $\mathcal{D}(S_i,t)$. Hence, S_i is the only $Holder(t+1)$ and thus $PSafetyCond$ holds at $t+1$.

Lemma 2 $\forall t \in \mathcal{C}, PSafetyCond(t) \wedge PCount(t) \wedge PHolderMonitored(t) \implies PCount(t+1)$.

Proof. Let suppose that $PSafetyCond$, $PCount$, and $PHolderMonitored$ are true at t. At $t+1$, site S_i can call one of the three following functions:

1. ***SafeReceiveToken*$\langle TOKEN, S_T, count_{recv}\rangle$:** Since S_i does not send any message in this function, $PCount(t).(4)$ ensures $PCount(t+1).(4)$ and, by definition, this call does not change $Holder(t)$. Moreover, $HCount(t) = HCount(t+1)$ and $PCount(t+1).(3)$ is verified for all non faulty sites other than S_i. Thus, in order to prove this lemma, we just need to show that $PCount(t+1).(3)$ holds for S_i. Line 1 ensures that $count(S_i,t+1) = count_{msg}$. Moreover, line 1 implies that $\mathcal{D}(S_i,t+1) = \{S_T,..,S_i\}$, hence:
$\#(\mathcal{D}(S_i,t+1)) = \#(\{S_T,..,Holder(t+1)\}) + \#(\{Holder(t+1),..,S_i\}) - 1$.

$$
\begin{aligned}
PCount(t).(4) \implies & HCount(t) - count_{msg} = \\
& \quad \#(\{S_T, .., Holder(t)\}) - 1 \\
\implies & HCount(t+1) - count(S_i, t+1) = \\
& \quad \#(\mathcal{D}(S_i, t+1)) - \#(\{Holder(t+1), .., S_i\}) \\
\implies & PCount(t+1).(3) \text{ for } S_i
\end{aligned}
$$

2. ***SafeSendToken*:** Since S_i adds one to *count* before sending the $\langle TOKEN \rangle$ message, $HCount(t+1) = HCount(t) + 1$ (line 1). In addition, as $Holder(t+1) = S_{i+1}$ (the successor of S_i in the ring), for every site S_x, with a non empty $\mathcal{D}$ at t, $\#(\mathcal{D}(S_x, t+1)) - \#\{Holder(t+1), .., S_i)\}$ is also incremented by one. Therefore, $PCount(t+1).(3)$ is true. Analogously, we can prove that $PCount(t+1).(4)$ holds for all current pending $\langle TOKEN \rangle$ messages $\in \mathcal{M}(S_i, t)$. To complete the proof, we must show that the $\langle TOKEN,$ $S_T,$ $count_{\text{recv}} \rangle$ messages sent by S_i at t verify $PCount(t+1).(4)$. Since, at $t+1$, $S_T = Holder(t+1)$ and $count_{\text{recv}} = HCount(t+1)$, we have: $HCount(t+1) - HCount(t+1) = \#(\{Holder(t+1), .., Holder(t+1)\}) - 1$.

3. ***UseBackup*:** Let S_x be a non faulty site with a non empty $\mathcal{D}$ at t. S_i belongs to $\{Holder(t), .., S_x\}$. If such was not the case, S_x would belong to $\{Holder(t), .., S_i\}$ and, since $PHolderMonitored(t)$ ensures that $Holder(t) \in \mathcal{D}(S_i, t)$, S_x would belong to $\mathcal{D}(S_i, t)$ too. However, as S_i called the function *UseBackup*, S_x should be faulty. We come therefore to a contradiction.

   Moreover, when S_i executes *UseBackup*, it updates its *count* such that $count(S_i, t+1) = count(S_i, t) + \#(\mathcal{D}(S_i, t)) - 1$ (line 1). Hence, as show in Figure 7, we can include S_i in $\{Holder(t), .., S_x\})$ in order to be able to exploit such an increment. Thus, we prove that the fonction *UseBackup* preserve the counter property on all sites : $PCount(t).(3) \implies PCount(t+1).(3)$.

   The same reasoning can be applied for the pending messages by replacing $count(S_i, t)$ by $count_{recv}$.

Lemma 3 $\forall t \in \mathcal{C}, PSafetyCond(t) \wedge PCount(t) \wedge PHolderMonitored(t) \implies PHolderMonitored(t+1)$.

Proof. Let $PSafetyCond$, $PCount$, and $PHolderMonitored$ be true. At t. At $t+1$, site S_i can call one of the three functions:

1. ***SafeReceiveToken*$\langle TOKEN, S_T,$ $count_{recv} \rangle$:** Since $PHolderMonitored(t)$ is true, $Holder(t) \in \{S_T, .., S_i\}$. Hence, when S_i considers this set as its new detection set (line 1), S_i will start monitoring the token holder: $Holder(t) = Holder(t+1) \in \mathcal{D}(S_i, t+1)$

2. ***SafeSendToken*:** in this case, the new token holder is the successor of S_i in the ring: $Holder(t+1) = S_{i+1}$. Since a non-empty $\mathcal{D}$ of a non faulty site S_j is always composed of the set of continuous predecessors of S_j in the ring, and since $PHolderMonitored(t)$ ensures that $Holder(t) = S_i$ belongs to this set, S_{i+1} belongs to every non empty set S_j such that $S_j \neq S_i$ (notice that in line 1 S_i empties its $\mathcal{D}$). Furthermore, for all $\langle TOKEN, S_T, count_{recv} \rangle$ of $\mathcal{M}(S_i, t+1)$ such that $S_j \neq S_i$, $S_{i+1} \in \{S_T, \ldots S_j\}$. Thus, $Holder(t+1)$ is monitored at $t+1$.

3. ***UseBackup*:** To prove it, we must show that $S_i = Holder(t+1)$ (Lemma 1) belongs to every non-empty detection set. We can easily prove it by contradiction. Let us suppose that there exists one non-faulty site S_x with a non empty detection set such that $S_i \notin \mathcal{D}(S_x, t)$. Since $\{Holder(t), .., S_x\} \subset \mathcal{D}(S_x, t)$ $(PHolderMonitored(t))$, $S_i \notin \{Holder(t), .., S_x\}$ and thus
$\{Holder(t), .., S_x\} \subset \{Holder(t), .., S_i\}$. Moreover, since $\{Holder(t), .., S_i\} \subset \mathcal{D}(S_i, t)$ $(PHolderMonitored(t))$, $S_x \in \mathcal{D}(S_i, t)$ which is a contradiction because S_i calls function *UseBackup* only when all the sites it monitors, except itself, are faulty (lines 29 and 36). Based on the same reasoning, we prove that $PHolderMonitored$ holds at $t+1$ for the pending messages.

Theorem 1 *Our algorithm ensures the safety property.*

Proof. Based on the previous lemmas, we can easily prove by induction that the three properties $PSafetyCond$, $PCount$ and $PHolderMonitored$ are always verified.

At $t = 0$, the function *Initialisation* is executed and the three properties are thus verified:

- ***PSafetyCond*(0):** S_0 is the only token holder (line 1) and there exists no pending $\langle TOKEN \rangle$ message.
- ***PCount*(0):** all sites have the same *count* value (line 1).
- ***PHolderMonitored*(0) :** Each node with a non-empty detection set monitors the token holder S_0.

Suppose that all tree properties hold at time t. Lemmas 1, 2, and 3, ensure that properties $PSafetyCond$, $PCount$, and $PHolderMonitored$ are true at $t+1$ respectively.

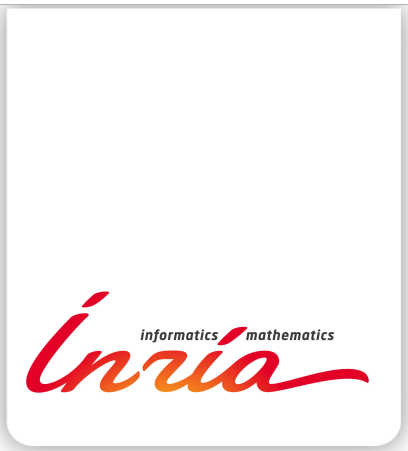
Since the $PSafetyCond$ is a sufficient condition to ensure the safety property, we can state that our algorithm ensures this property.

Theorem 2 *For at most k consecutive failures, our algorithm ensures the liveness property.*

Proof. In order to prove the liveness property, it is sufficient to prove that (1) if the token is not lost, it will be sent from a site to its successor on the ring and that (2) if the token is lost due to the failure of the current holder of the *REAL* token, the latter will be regenerated by one of the non faulty k successors of the faulty token holder.

The first one (1) is easily proved because when S_i sends the $\langle TOKEN \rangle$ message (line 1) at t, it informs in the message that the next holder of the *REAL* token is S_{i+1} $(Holder(t+1) = S_{i+1}.)$ Additionally, as the channels are reliable and S_i has the greatest *count* value at t, $(PSafetyCond(t)$ and $PHolderCount(t))$, S_{i+1} will deliver the token to the original algorithm (line 1) upon reception of the message.

To prove the seconde one (2), let us consider that S_i is the last site to hold the token which sent the $\langle TOKEN \rangle$ message at t to its k successors and that f is the number of faulty direct successors of S_i. Since $f \leq k$, site S_i has sent the message to the correct site $S_{i+f+1} \in \{S_{i+1}, .., S_{i+k+1}\}$. Moreover, for the same reasons described in (1), S_{i+f+1} will eventually receive the message and will not discard it. Line 1 will thus set $\mathcal{D}$ of S_{i+f+1} to $\{S_{i+1}, .., S_{i+f+1}\}$. By assumption, the failures of sites $\{S_{i+1}, .., S_{i+f}\}$ are (resp. will be) detected which will result in the calling of the *UseBackup* function of line 1 (resp. line 1).



RESEARCH CENTRE
PARIS – ROCQUENCOURT

Domaine de Voluceau, - Rocquencourt
B.P. 105 - 78153 Le Chesnay Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399