A O(log2n) fault-tolerant distributed mutual exclusion algorithm based on open-cube structure

Jean-Michel Hélary, Achour Mostefaoui

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

A $O(\log_2 n)$ fault-tolerant distributed mutual exclusion algorithm based on open-cube structure

Jean-Michel Hélary, Achour Mostefaoui

N° 2041

Septembre 1993

PROGRAMME 1

ISSN 0249-6399

A O($\log_2$ n) fault-tolerant distributed mutual exclusion algorithm based on open-cube structure*

J. M. Hélary, A. Mostefaoui

Programme 1

Projet Algorithmes Distribués et Protocoles

Rapport de recherche n°2041 -Septembre 1993

21 pages

Abstract: A new distributed mutual exclusion algorithm, using a token and based upon an original rooted tree structure, is presented. The rooted tree introduced, named “open-cube”, has noteworthy stability and locality properties, allowing the proposed algorithm to achieve good performances and high tolerance to node failures: the worst case message complexity per request is, in the absence of node failures, $\log_2 N+1$ where N is the number of nodes, whereas $O(\log_2 N)$ extra messages in the average are necessary to tolerate each node failure. This algorithm is a particular instance of a general scheme for token and tree-based distributed mutual exclusion algorithms, previously presented in part by the authors; consequently, its safety and liveness properties are inherited from the general one; however, the present paper is self-contained.

Key-words: distributed algorithms, mutual exclusion, token, tree structure, fault-tolerance

* Submitted to the 14th Int. Conf. on Distributed Computing Sysytems

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu, 35042 Rennes Cedex (France)
Téléphone : (33) 99 84 71 00 - Télécopie : (33) 99 38 38 32

Un algorithme réparti d'exclusion mutuelle tolérant les pannes, basé sur une structure de cube ouvert

Résumé : On présente un nouvel algorithme réparti d'exclusion mutuelle, utilisant un jeton et basé sur une structure arborescente originale. Cette structure, appelée "hypercube ouvert", possède de remarquables propriétés de stabilité et de localité, grâce auxquelles l'algorithme proposé présente de bonnes performances et une grande résistance aux défaillances de sites : la complexité maximale, en nombre de messages par requête, est $\log_2 N+1$, et $O(\log_2 N)$ messages supplémentaires, en moyenne, sont nécessaires pour traiter chaque panne de site. Cet algorithme est une instanciation d'un schéma générique d'algorithmes répartis d'exclusion mutuelle utilisant un jeton et une structure arborescente, présenté auparavant en partie par les auteurs; en conséquence, ses propriétés de sûreté et de vivacité sont héritées de l'algorithme générique; toutefois, la lecture de cet article ne suppose pas la connaissance préalable de l'algorithme générique.

Mots-clé : algorithmique répartie, exclusion mutuelle, jeton, arborescence, tolérance aux défaillances.

algorithmique répartie, exclusion mutuelle, jeton, arborescence, tolérance aux défaillances.

1 Introduction

This paper deals with mutual exclusion problem in distributed systems. A distributed system is characterized by a set of nodes, identified by 1, 2, ... , *n*. The nodes communicate only by messages exchanged through communication channels; they don't share any memory nor a global clock. Channels are supposed to be reliable (messages are neither lost nor corrupted) and communication is asynchronous (message propagation delay is finite but unpredictable). Between any pair of nodes, messages can be delivered out of order (channels can be FIFO or not). Finally, without loss of generality for our purpose, we suppose that there is exactly one process per node: so in the following we consider these two terms as synonyms.

Within such a context, several mutual exclusion algorithms have been proposed ([6,7]. One important class of solutions is based on the use of a *token*: uniqueness of the token guarantees the safety property (at any time, at most one process can be in the critical section) by subjecting the right to enter the Critical Section to the possession of this token. The liveness property (each request to enter the critical section will be satisfied after a finite time) is guaranteed, in the absence of channel or node failure, by the design of a routing structure used by the node requests and by the token, together with appropriate rules in order to avoid deadlock and starvation.

The advantages of this approach lie in the possibility to achieve good performances, in terms of number of messages needed to satisfy a request to enter the critical section (message complexity per request). The best known algorithms are tree-based: each node sends its requests to one qualified neighbor (its "father"), which makes the request progress towards the token [2,3,4,7]. The set of all *fathers* define a rooted tree, with edges directed towards the root; the token is kept by the root, which plays the role of token allocator (the root temporarily lends the token to requesting nodes, one after each other). The message complexity per request is, in the average, O(*d*), where *d* is the diameter of the tree. Depending on the tree design, it can be as low as $O(\log_2 n)$, where *n* is the number of nodes.

In Raymond's and Van de Snapsheut's algorithms, the tree structure is static, although edges are dynamically directed according to the token position: each node defines its father as the neighbor belonging to the subtree containing the token. The worst case message complexity per request is O(*d*) (in that case, *d* is statically defined). However, this solution has some disadvantages :

- the amount of work performed by each node depends only on its position in the tree (in fact, on its degree) and not on the frequency of its requests to enter the critical section,
- if a node fails, the tree has to be rebuilt.

In Naimi-Trehel's algorithm, on the contrary, the tree structure is dynamic and evolves according to the occurrence of new requests. This overcomes the first disadvantage of the preceding solution, since the less a node requests to enter the critical section, the further it is from the root, and thus the lighter becomes its workload. But this must be paid by another disadvantage: the tree can meet any possible configuration, leading to a worst case message complexity per request of O(n) (although O($\log_2 n$) in the average).

Each of these two extreme algorithms is an instance of a general scheme for the class of token-based algorithms using a rooted tree to move the requests, proposed by Hélary and al. [1], allowing to design new algorithms whose behavior can be tuned from the completely static (Raymond) to the completely dynamic one (Naimi-Trehel). In the present paper a new algorithm, based on this general scheme, is proposed; its aim is to combine the advantages of both static and dynamic cases since it allows the position of the nodes to evolve, as in Naimi-Trehel's, but within a tree whose diameter will remain bounded to O($\log_2 n$) as in Raymond's. Moreover, the particular design of the tree allows to take into account nodes failures and token loss: recovery from a node failure (including safe token regeneration if necessary) requires only O(log$n$) *extra* messages, in the average. This important feature, which, to our knowledge, has not been achieved in any of the previously known tree-based algorithms, is due to a "locality" property inherent to the structure of the underlying tree.

The rest of the paper contains four parts: the logical tree structure and its properties is addressed in Section 2; the algorithm is presented in Section 3, and Section 4 addresses correctness and performance issues; in these two sections, it is assumed that nodes do not fail, but this assumption is removed in Section 5, where it is shown how to handle such failures.

2 The open-cube tree structure

The tree structure upon which the algorithm is based can be recursively described. For simplicity, we assume that the number of nodes is a power of two, say $n=2^p$. The tree is made of two identical sub-trees, having 2^{p-1} nodes, connected by one directed edge linking their roots, as shown in Figure 1 (links are directed towards the root):

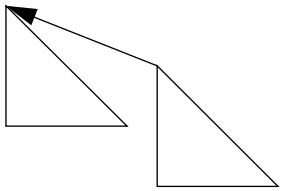


Figure 1 : recursive structure

For particular values n=2, n=4, n=8 and n=16 we obtain (Figures 2a, 2b, 2c, 2d):

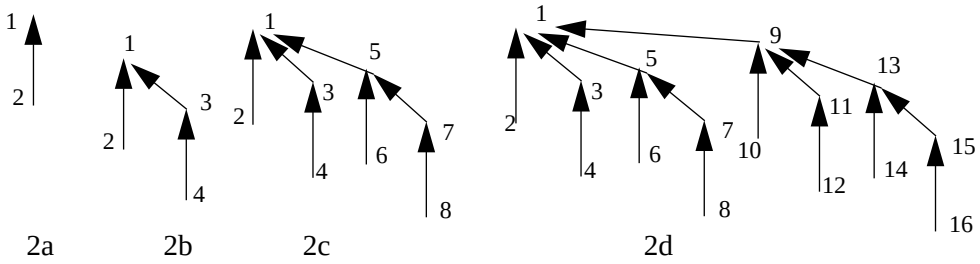


Figure 2 : examples of open-cubes

Such a tree is called “n-open-cube” since it corresponds to a n-hypercube from which some links have been removed (Figure 3).

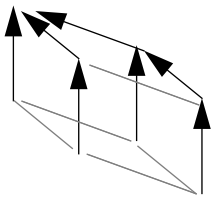


Figure 3 : the 8-open-cube and the corresponding 8-hypercube

The concept of p-group and locality

A *p-group* is a set of nodes belonging to an open-cube subtree having 2^p nodes. For example, in the 16-open-cube of figure 2d, {1,2}, {3,4}, ..., {15,16} are 1-groups, {1,2,3,4}, {5,6,7,8}, {9,10,11,12}, {13,14,15,16} are 2-groups, {1,2,...,8}, {9,10,...,16} are 3-groups and {1,2,...,16} is a 4-group. Each node is the root of one or several p-groups. For example, in the open-cube of Figure 2d, the node 1 is the root of a 0-group ({1}), a 1-group ({1,2}), and so on.

Definition 2.1. The *power* of a node *i* is the greatest integer *p* such that *i* is the root of a p-group.

For example, in the 16-open-cube of Figure 2d, node 1 is of power 4, node 2 of power 0, node 3 of power 1, node 5 of power 2, node 9 of power 3, and so on. It is easy to see that a node of power *p* has exactly *p* sons, whose powers range from 0 to *p*-1.

Definition 2.2. The *distance* of two nodes *i* and *j*, denoted *dist(i, j)*, is the smallest integer *d* such that *i* and *j* belong to the same *d*-group.

For example, in the 16-open-cube of Figure 2d, *dist*(1,2)=1, *dist*(1,*j*)=2 if *j*=3 or 4, *dist*(1,*j*)=3 for *j*=5,...,8 and *dist*(1,*j*)=4 for *j*=9, ... ,16.

Proposition 2.1 If *j* is a son of *i*, then *power(j)=dist(i, j)* - 1

Proof Obviously, a node *i* together with the subtrees rooted at the sons of *i* of power *r* (0≤*r*≤*power*(*i*)-1) form a (*r*+1)-group. Thus, if *j* is the son of *i* of power *r*, *dist*(*i*,*j*)=*r*+1

Proposition 2.2 The following implication holds :

$$\forall i \ \forall j : dist(i,j) = power(i) + 1 \Rightarrow j = father(i) \text{ or } power(j) < power(i)$$

Proof The nodes located at distance *power(i)*+1 from *i* are *father(i)* and the nodes belonging to the subtrees rooted at the brothers of *i* of power 0, ..., *power(i)*-1. All, except *father(i)*, have a power less or equal to *power(i)*-1

Corollary 2.1 *father(i)* is the only node *j* such that (i) *dist(i,j)=power(i)*+1 and (ii) *power(j)>power(i)*

Definition 2.3 The *last son* of a node of power *p*>0 is its son of power *p*-1, and an edge (*j*, *i*) is a *boundary edge* if *j* is the last son of *i*, in other words : $power(i) = power(j) + 1$ (recall that, for every edge (*j*, *i*), $power(i) \geq power(j) + 1$).

The following theorem shows which pairs father-son can be swapped over without changing the open-cube structure of the tree.

Theorem 2.1. Let (*j*, *i*) be an edge in an open-cube. The following transformation :
father(j) := *father(i)* ; *father(i)* := *j*
keeps the open-cube structure if, and only if, (*j*, *i*) is a boundary edge. In this case, it decreases the power of *i* by one, and increases the power of *j* by one.

Proof.

Figure 4 : node swapping

i) Suppose that (*j*, *i*) is a boundary edge, and let *p=power(i)*. Thus, *power(j)=p*-1. By the transformation, the open-cube remains a tree (Figure 4). Moreover, *i* loses its last son and thus, after the transformation, *i* has *p*-1 sons, whose powers range from 0 to *p*-2. Hence, after the first assignment, *power(i)=p*-1. On the other hand, before the transformation, *j* has *p*-1 sons, whose powers range from 0 to *p*-2. By the transformation, *j* gets a new son, whose power is *p*-1, and thus, after the second assignment, *power(j)=p*. Consequently, the open-cube structure is kept, with the edge (*i*, *j*) as boundary edge instead of the edge (*j*, *i*) (the node *i* becomes the last son of *j*).

ii) Conversely, suppose that (*j*, *i*) is not a boundary edge. The following counter-example shows that the transformation destroys the open-cube structure : consider the 4-open-cube (Figure 5) and the transformation performed with *i*=1 (*power(*1)=2) and *j*=2 (*power*(2)=0); after the two assignments, we have : *father*(2)=*nil*, *father*(1)=2 and, obviously, the tree is no more an open-cube.

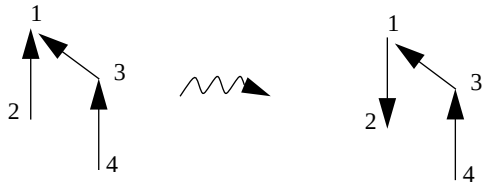


Figure 5 : node swapping

Swapping over a node with its last son will be called a b-transformation (*b* stands for *boundary*).

Corollary 2.2 When a b-transformation is performed, all the p-groups remain unchanged.

Corollary 2.3 When a b-transformation is performed, the distance between two nodes remain unchanged.

These results express the concept of locality: within a p-group, the node membership remains unchanged under the b-transformation, and the link connecting the two (p-1)-group composing the p-group is persistent (although its direction and extremities can change). For example, in the 2-group {5,6,7,8} of the 16-open-cube (Figure 2d) the link connecting the two 1-groups {5,6} and {7,8} is persistent (it can connect 5 and 7, 5 and 8, 6 and 7, or 6 and 8, in either direction).

A bound on the length of branches

Let $i_r\ i_{r-1}\ ...\ i_0$ be a branch of a N-open-cube (with $N=2^p$); i_r is the root, i_0 is a leaf. Let $p_r, p_{r-1}, ..., p_0$ be the respective powers of nodes $i_r\ i_{r-1}\ ...\ i_0$. We have the following result:

Proposition 2.3 $r \leq \log_2 N - n_1$, where n_1 is the number of nodes on the branch which are not last sons.

Proof p_r, p_{r-1}, ..., p_0 is a monotonic strictly decreasing sequence of integers, with $p_r=\log_2 N$ and $p_0=0$. But, $p_r - p_0 = \sum_{j=0}^{r-1} (p_{j+1} - p_j)$ and thus $r = p_r - p_0 - \sum_{j=0}^{r-1} (p_{j+1} - p_j - 1)$. On the other hand, $p_{j+1} - p_j - 1 \geq 1 \Leftrightarrow$ the node i_j is not the last son of the node i_{j+1}, and thus the latter inequality holds for exactly n_1 terms in the sum whence $\sum_{j=0}^{r-1} (p_{j+1} - p_j - 1) \geq n_1$

3 Description of the algorithm

In this section, we assume that nodes do not fail (recall that we have assumed earlier that channels are reliable). Node failure issue will be addressed in Section 5.

3.1 Principle

Initially, nodes are arranged according to an open-cube structure, and the token is located at the root of the tree. When a node wants to enter the critical section, it issues a request to have the token; the right to enter the critical section is granted by the possession of the token. According to the progression of this request, the rooted tree will possibly evolve; however, the algorithm is designed in such a manner that this evolution maintains the open-cube structure. According to the Theorem 2.1, this will be achieved if this evolution involves only b-transformations. In order to meet this requirement, each node maintains local information, whose initial value reflects the initial open-cube structure; the description of the algorithm consists in explaining how this information is used and updated. Each node has thus local variables describing its local state (with regard to the token and to the critical section), its position in the logical rooted tree, and its behavior.

Local state of node *i*

The presence of the token is indicated by the boolean variable $token\_here_i$, whose value is *true* if, and only if, the node *i* has the token. Moreover, the boolean variable $asking_i$ has the value *true* if, and only if, node *i* is currently waiting for the token or

executing a critical section. Managing these two variables is straightforward.

Position in the open-cube

Each node *i* maintains the following data : an array $dist_i$ such that, for any *j*, $dist_i(j)$ is the distance between *i* and *j* in the open-cube; the power of the root, denoted by *pmax*; a variable $father_i$ denoting the father of *i* in the open-cube. From this information, the power of *i* can be deduced : by proposition 2.1, $power(i)=dist_i(father_i)$-1 if $father_i \neq nil$, $power(i)=pmax$ otherwise. Initial values of these data are set upon initialization of the open-cube. Let us remark that, if the evolution of the open-cube involves only b-transformations, the data $dist_i$ and *pmax* are *constants*, whereas $father_i$ is a *variable*.

Each node *i* has also a variable $lender_i$: its value indicates the node to which *i* will have to give back the token when leaving the critical section. Its value is meaningful only when *i* is in the critical section : it is updated upon the token receipt granting the right to enter the critical section, and used upon leaving the critical section. The information needed by a node *i* in order to update the value of $lender_i$ is carried over by the token: the latter is thus implemented by a message *token*(*j*) where *j* is a node identity: if $j \neq nil$, $lender_i$ is set to *j*; if *j=nil* it means that *i* will keep the token; consequently, $lender_i$ is set to *i* (*i* becomes the root and $father_i$ is set to *nil*).

The two variables $father_i$ and $lender_i$ have “dual” meanings since the former indicates from which node the token should be requested, whereas the latter indicates to which one give back this token.

Requests and behavior of a node

When a node *i* wants to get the token, it sends a message *request*(*i*) to $father_i$, sets $asking_i$ to *true*, then waits for the token arrival. When a node *i* processes a message *request*(*j*) from one of its sons, say *k*, it reacts to this event with two different behaviors, according to the position of *k* :

- If *k* is the *last son* of *i*, the node *i* will either give up the token to node *j* (if *i* is the root and has the token, it sends *token(nil)* to *j*) or forward the message *request(j)* to $father_i$. Afterwards, *i* considers that, in the future, it will have to send requests to *j*: consequently it sets $father_i:=j$. In other words, the edge $(i, father_i)$ is replaced by the edge (*i*, *j*). We will say that the node *i* conforms to a *transit* behavior. Let us note that when a *request* message is processed by a transit *node*, the first assignment of a b-transformation is performed ($father_i:=j$).

- If *k* is *not* the last son of *i*, the node *i* considers *j* as its *mandator* and, consequently, either lends the token to *j* (if *i* is the root and has the token, it sends *to-*

ken(i) to *j*, hereby meaning that, after being used, the token will have to return to *i*) or requests the token for itself, by sending a message *request(i)* to $father_i$, thus becoming an asking node ($asking_i$:=*true*). We will say that the node *i* conforms to a *proxy* behavior (*i* is *proxy* for *j*, or, equivalently, *j* is the *mandator* of *i*). Let us note that when a request message is processed by a *proxy* node, the tree is not modified: updating $father_i$ is postponed until the token is received.

Each node has thus a variable $mandator_i$. The value of this variable is a node identity and is meaningful only when *i* has requested the token for satisfying a request. $mandator_i=i$ means that *i* wants to enter the critical section; $mandator_i=j$, $j \neq i$, means that *i* has processed a message *request(j)* and conformed to *proxy* behavior; the variable $mandator_i$ will be reset to *nil* when *i* will receive the requested token: *i* will cease its mandate for this request. So, $mandator_i=nil$ means that the node *i* has no current request.

Let us remark that performing the test whether *k* is the last son of *i* is very easy : *k* is the last son of *i* if, and only if, *power*(*k*)=*power*(*i*)-1. But, since *k* is a son of *i*, we have, from proposition 2.1, $power(k)=dist_i(k)$-1; thus, *k* is the last son of *i* if, and only if, $dist_i(k)=power(i)$, or, equivalently : $dist_i(k)=dist_i(father_i)$-1.Moreover, it is important to note that, since this test is performed upon the processing of the message *request(j)* sent by *k*, the latter is an ancestor of *j*; thus, $dist_i(k)=dist_i(j)$. From this follows that *k* is the last son of *i* iff $dist_i(j)=dist_i(father_i)$-1. Hence, the node *i* does not need to be aware of the identity *k*: the value *j* brought up by the *request* message is sufficient for *i* to perform the test.

When a node *i* receives the token, this can result from a request previously made by *i* and still pending (in that case, $mandator_i \neq nil$), or from a return to *i* after a loan (in that case, $mandator_i=nil$).

1. $mandator_i \neq nil$ and the node *i* receives *token(j)*.
   If $j \neq nil$, it means that the token is lended by node *j*. The node *i* sets $father_i$ to *k*, the node from which it has received the token: as will be shown in Section 4, this updating restores the open-cube property which could have been “temporarily” destroyed when the *transit* nodes located on the path between *i* and the root have performed the first part of a b-transformation; then the node *i* honors the request (entering critical section with $lender_i=j$ if $mandator_i=i$ or sending *token(j)* to $mandator_i$ if $mandator_i \neq i$) and resets $mandator_i$ to *nil*
   If $j=nil$, it means that the token has no lender. The node *i* sets $father_i$ to *nil* (it becomes the root); then it honors the request (entering critical section with $lender_i=i$ if $mandator_i=i$ or sending *token(i)* to *man-*

dator$_i$ if *mandator*$_i \neq i$) and resets *mandator*$_i$ to *nil*. If the node *i* has entered the critical section with *lender*$_i \neq i$, it will send back *token*(*nil*) to *lender*$_i$ upon leaving the critical section.

2. *mandator*$_i$=*nil*. The node *i* receives *token*(*nil*). This is a return of the token after a loan.

Relation with the general algorithm

In the general token- and tree-based algorithm presented in [1], the behavior of each node is caught in a local variable *behavior*$_i$, having at any time one of the two values: *transit* or *proxy*; a fundamental characteristic of the general algorithm is the possibility of arbitrary static or dynamic assignment of these variables. In consequence, any static or dynamic rule of assignment can be considered, each yielding a particular algorithm. Actually a particular choice for the behavior of nodes can be controlled according to the supposed evolution of the underlying tree (the efficiency of a tree-based mutual exclusion algorithm indeed depends on this structure). For instance, Raymond's algorithm is obtained when the behavior of each node is *transit* when it has the token and *proxy* otherwise, i.e., *behavior*$_i$=*transit*$\Leftrightarrow$*token_here*$_i$. The structure of the tree, initially defined, doesn't change, except the direction of the edges. On the opposite, Naimi and Trehel's algorithm is obtained when each node is permanently *transit*, and thus the tree can meet any possible configuration. The particular algorithm presented in the present paper is thus obtained by applying the following rule: for each node *i*, the value of *behavior*$_i$ is updated upon every *request* message processing: it is assigned the value *transit* if the message has been sent by the last son, and the value *proxy* otherwise. It will be proved, in Section 4, that this rule allows the open-cube structure to be maintained. Another important consequence of this remark is that the *safety* and *liveness* proofs given for the general case hold for each particular instance resulting from particular rules of assignment.

Queues

If several nodes *j* are such that *father*$_j$=*i*, the node *i* can receive several "simultaneous" requests; moreover, the process associated with the node *i* may wish to enter the critical section. In order to deal with this multiplicity of requests, a *waiting-queue* is associated with each node. Its service policy is implicit; the only assumption is *fairness*, meaning that every waiting request will wait a finite time before being processed. For example, the FIFO policy is fair. No waiting request can be processed by *i* unless the boolean variable *asking*$_i$ has the value *false*. Thus, each node can be seen as a *request server*, whose busy periods correspond to the time during which *asking*$_i$ is *true*, service corresponds to the request of the token (on current mandator's account), and clients are pending requests waiting in the queue. In the algorithmic ex-

pression, the primitive "**wait** (**not** $asking_i$)" expresses the precondition to the execution of actions related to events *local call to enter_cs* and *receive request(j)*; it corresponds to the fact that process *i* is occupied to serve another request.

3.2 Example

The Figure 6 below depicts the initial situation (16-open-cube) : the node 1 has lended the token to the node 6 which is currently in critical section. We examine the case where nodes 10 and 8 both require the right to enter the critical section (in our example, the request of 10 will be satisfied before the request of 8, but this is irrelevant to the discussion).

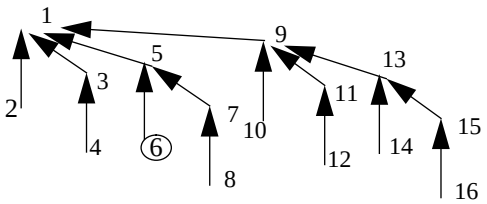


Figure 6 : initial situation

pmax=4

Node 10 wishes to enter the critical section and **not** $token\_here_{10}$ and **not** $asking_{10}$:
send *request*(10) to $father_{10}$=9; $asking_{10}$:=*true*; $mandator_{10}$:=10

Node 9 receives *request*(10) from 10 and **not** $token\_here_9$ and **not** $asking_9$: -- *behavior=proxy*
send *request*(9) to $father_9$=1; $asking_9$:=*true;* $mandator_9$:=10

Node 1 receives *request*(9) from 9 and $asking_1$:
request(9) is queued

Node 8 wishes to enter the critical section and **not** $token\_here_8$ and **not** $asking_8$:
send *request*(8) to $father_8$=7; $asking_8$:=*true*; $mandator_8$:=8

Node 7 receives *request*(8) from 8 and **not** $token\_here_7$ and **not** $asking_7$: -- *behavior=transit*
send *request*(8) to $father_7$=5; $father_7$:=8
-- $asking_7$ remains *false*, the power of 7 becomes 0 and the power of 8 becomes 1.
-- The variable $father_8$ will be updated later

Node 5 receives *request*(8) from 7 and **not** $token\_here_5$ and **not** $asking_5$: -- *behavior=transit*
send *request*(8) to $father_5$=1; $father_5$:=8;
--$asking_5$ remains *false*, the power of 5 becomes 1 and the power of 8 becomes 2.
-- The variable $father_8$ will be updated later

Node 1 receives *request*(8) from 5 and $asking_1$:
request(8) is queued

Node 6 exits CS:
send *token*(*nil*) to $lender_6$=1 ; $token\_here_6$:=*false*; $asking_6$:=*false*

Node 1 receives *token*(*nil*) and $mandator_1$=*nil* : -- the token comes back after a loan by the root 1
$token\_here_1$:=*true*; $asking_1$:=*false*

Node 1 processes *request*(9) and $token\_here_1$:
-- *behavior=transit* since *power(1)*=4 and $dist_1$(9)=4 : 1 *gives up the token to* 9
send *token*(*nil*) to 9; $father_1$:=9; $token\_here_1$:=*false*

At that point, the configuration of variables *father* is the following (Figure 7):

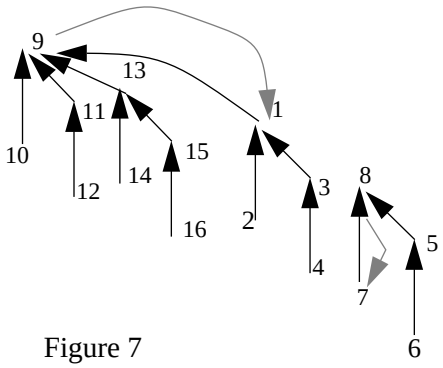


Figure 7

• the edge (9,1) will be removed when 9 will receive the token. At that time, $father_9$ will be updated according to the content of the message *token* Until then, node 9 remains busy ($asking_9$=*true*)

• the edge (8,7) will be removed when 8 will receive the token. At that time, $father_8$ will be updated. Until this, node 8 remains busy ($asking_8$=*true*)

Node 1 processes *request*(8) and **not** $asking_1$ and **not** $token\_here_1$:
-- $behavior_1$=*transit* since *power*(1)=$dist_1$(9)-1=3 and $dist_1$(8)=3
send *request*(8) to $father_1$=9; $father_1$:=8

Node 9 receives *token*(*nil*) and $mandator_9$=10: -- 9 *becomes the lender*
$lender_9$:=9 ; $father_9$:=*nil;*
send *token*(9) to $mandator_9$=10 ; $mandator_9$:=*nil*
-- *its mandate for node* 10 *is now completed but* $asking_9$ *remains true until the token returns*

Node 9 receives *request*(8) from 1 and $asking_9$=*true* :
request(8) is queued

Node 10 receives *token*(9) from 9 and $mandator_{10}$=10:
$lender_{10}$:=9; $father_{10}$:=9 ; -- the token comes from node 9

$mandator_{10}$:=*nil*; $token\_here_{10}$:=*true*;
ENTER CS

Node 10 EXITS CS and $lender_{10}$=9 :
send *token*(*nil*) to $lender_{10}$=9; $token\_here_{10}$:=*false*; $asking_{10}$:=*false*

Node 9 receives *token*(*nil*) and $mandator_9$=*nil*: -- the token comes back after a loan by the root 9
$token\_here_9$:=*true*; $asking_9$:=*false*

Node 9 processes *request*(8) and **not** $asking_9$ and $token\_here_9$:
-- *behavior*=*transit* since *power*(9)=*pmax*=4 and $dist_9(8)$=4
send *token*(*nil*) to 8; $token\_here_9$:=*false*; $father_9$:=8

Node 8 receives *token*(*nil*) from 9 and $mandator_8$=8:
$lender_8$:=8; $father_8$:=*nil*; $mandator_8$:=*nil*; $token\_here_8$:=*true*;
ENTER CS

Node 8 EXITS CS and $lender_8$=8 :
-- 8 keeps the token (it is the root)

The final situation is shown on Figure 8:

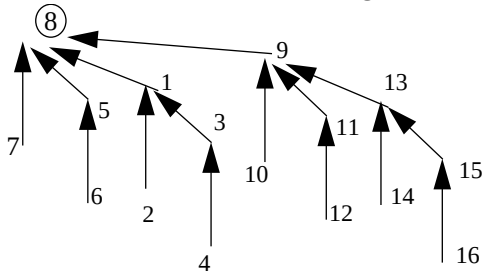


Figure 8: final configuration

3.3 Formal description of the algorithm

The text of the algorithm describes the actions performed by each node *i* upon the occurrence of each of the four possible events: *i* wishes to enter the critical section (local call to the procedure *enter_cs*), *i* exits the critical section (local call to the procedure *exit_cs*), *i* receives a *request* message, *i* receives a *token* message. Apart from the precondition **wait** (**not** $asking_i$) which may delay the beginning of the actions *enter_cs* and *receive request*, each of these four actions is processed atomically, i.e., without interruption.

Upon a call to *enter_cs* **by** *i*
begin
wait (**not** $asking_i$);
$asking_i$:=**true**;

```
        if not token_here_i then
            mandator_i:=i;
            send request(i) to father_i;
            wait (token_here_i) -- receipt of token sets lender_i
        endif
end -- enter_cs
Upon a call to exit_cs by i
    begin
        if lender_i≠i   then
            send token(nil) to lender_i;
             token_here_i:=false
        endif;
        asking_i:=false
end -- exit_cs


Upon receipt of request(j) by i
begin
    wait (not asking_i);
    case of dist_i(j) ≠dist_i(father_i)-1
        begin       -- i becomes proxy for j
             asking_i:=true;
            if token_here_i
                    then -- i temporarily lends the token
                        send token(i) to j;
                        token_here_i:=false

                     else -- i requires the token
                        mandator_i:=j;
                        send request(i) to father_i
            endif
         end
    dist_i(j) =dist_i(father_i)-1
        begin -- i has a transit behavior
            if token_here_i
                    then  -- i gives up the token
                        send token(nil) to j;
                         token_here_i:=false

                      else  -- i forwards the request
                        send request(j) to father_i
                        endif;
            father_i:=j
         end
```

```
    endcase
end -- request

Upon the receipt of token(j) from k by i
    -- j is the token lender; if j=nil the token does not have to be given back; asking_i=true
begin
    token_here_i:=true;
                    case of mandator_i=nil
          begin -- return of the token after loan
                    asking_i:=false

           end
          mandator_i=i
              begin  -- the claim of i to enter critical section is satisfied
                    if  j=nil then   -- the token has no lender, i becomes the lender
                                     lender_i:=i; father_i:=nil
                                        else          -- i will have to give back the token
                                     lender_i:=j; father_i:=k
                                     endif;
                    mandator_i:=nil -- asking_i remains true until leaving the critical section
               end
          mandator_i≠i,nil
              begin  --  i honors the request of its mandator
                    if  j=nil then   -- the token has no lender, i becomes the root and
                                     --   lends the token
                                     father_i:=nil;
                                           send token(i) to mandator_i;
                                     -- asking_i remains true until the tokenreturns

                           else      -- j is the lender of the token
                                     father_i:=k;
                                     send token(j) to mandator_i          ;
                                     asking_i:=false
                    endif ;
                    mandator_i:=nil ; token_here_i:= false
           end
    endcase
end -- token
```

4 Proof and performances

We will not give here the proof of safety and liveness properties: in fact, the algorithm presented here is an instance of the general one (allowing arbitrary assign-

ments, at arbitrary times, of variables $behavior_i$), whose complete proof is given in [1].

We will content ourselves to show that the open-cube structure of the tree is maintained, even when modifications of the variables *father* occur, due to node swapping. More precisely, we are going to show that, if the tree is an open-cube when a node sends a message *request* to its father, then it is still an open-cube when *i* eventually receives the token.

We begin with the case when no concurrent request reaches one of the nodes on the path from the requesting node *i* to the root. Let $i=i_0, i_1, \ldots, i_r$ denote the path from *i* to the root i_r in the open-cube existing when the node *i* sends a *request* to its father i_1. Two cases are to consider :

1. The path comprises only boundary edges. All the ancestors of *i* have a *transit* behavior, thus all nodes $i_l \ (1 \le l \le r-1)$ will forward the *request* of *i* to i_{l+1}, then set their father to *i*; the root i_r will send the token to *i*, then sets its father to *i*; finally, upon receiving the token, *i* will set its father to *nil*. Thus, the overall transformation of variables *father* is equivalent to the sequence of b-transformations:
   $father(i_0) := father(i_1)\ (= i_2)$; $father(i_1) := i_0$;
   ...
   $father(i_0) := father(i_l)\ (= i_{l+1})$; $father(i_l) := i_0$;
   ...
   $father(i_0) := father(i_r)\ (=nil)$; $father(i_r) := i_0$

By the theorem 2.1, each of these transformations keeps the open-cube structure. Thus, the overall transformation keeps the open-cube structure

Example with *r*=4 :

Figure 9 : transformation of a boundary path

2. At least one edge of the path is not a boundary edge. Let i_k be the first ancestor of *i* such that (i_{k-1}, i_k) is not a boundary edge $(1 \le k \le r)$. In other words, all the nodes $i_1, \ldots, i_{k-1}$ will have a *transit* behavior, and i_k will have a *proxy* behavior. Thus all nodes i_l $(1 \le l \le k-1)$ will forward the request of *i* to i_{l+1}, then set *father* to *i*; the node i_k will receive the request of *i*, then record *i* as its man-

dator, and send the token (immediately or after having required it) to *i*; the node *i* will eventually receive the token from i_k and, at that time, will set its father to i_k. As in the preceding case, the overall transformation is equivalent to the sequence of b-transformations:

$father(i_0) := father(i_1)\ (= i_2)\ ;\ father(i_1) := i_0\ ;$

...

$father(i_0) := father(i_{k-1})\ (= i_k)\ ;\ father(i_{k-1}) := i_0$

which, by the theorem 2.1, keeps the open-cube structure.

The case where several concurrent requests occur is similar : it is sufficient to observe that new incoming requests are either delayed if the node is busy (*asking* is *true*), or follow the path towards the current father otherwise.

Maximum number of messages per request

Let *i* be a node issuing a request to enter the critical section, and *r* be the length of the path going from *i* to the root. On that path, there are $n_1+n_2+1 = r+1$ nodes, where n_1 is the number of nodes which are not last sons and n_2 is the number of nodes which are last sons. By the proposition 2.3, $n_1 + n_2 \leq \log_2 N - n_1$. On the other hand, $r-1=n_1+n_2-1$ *request* messages are necessary to reach the root, but only n_2 *token* messages are necessary to reach back *i*, and may be 1 message to return the token (if $n_1 \neq 0$). Thus, the total number of messages is $2n_1 + n_2 + 1 \leq \log_2 N + 1$

Average number of messages per request

The actual number of messages necessary to satisfy a request issued by a node depends of the position of the node in the tree. Let $c(i)$ be that number for the node *i*. We compute the average number $c = \frac{\sum c(i)}{N}$ where the sum is over the *N* nodes of the open-cube.

If $N=2$, $\alpha_1 = \sum c(i) = c(1) + c(2) = 0 + 2 = 2$

If $N=2^p$ we have (see Figure 10) : $\alpha_p = \sum_{i \in T \cup T'} c(i) = \sum_{i \in T} c(i) + \sum_{i \in T'} c(i)$

If $N=2^{p+1}$ we have (see Figure 11) :

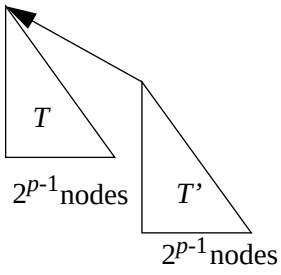


Figure 10

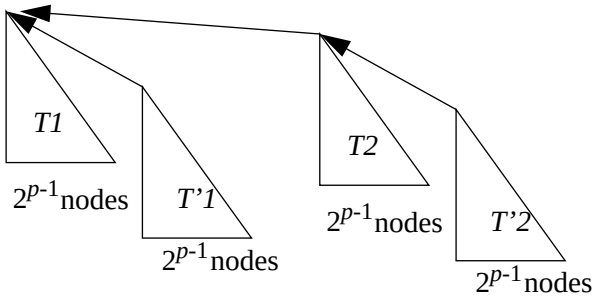


Figure 11

$$\alpha_{p+1} = \sum_{i \in T1} c(i) + \sum_{i \in T'1} c(i) + \sum_{i \in T2} c(i) + \sum_{i \in T'2} c(i)$$

But, $\sum_{i \in T1} c(i) = \sum_{i \in T} c(i)$ and $\sum_{i \in T'1} c(i) = \sum_{i \in T'} c(i) + p$ since each of the p boundary nodes of $T'1$ need one more message than in T' (to return the token to the root of $T1$)

similarly, $\sum_{i \in T2} c(i) = \sum_{i \in T} c(i) + 2 \times 2^{p-1}$ since each node of $T2$ needs two more messages than in T to reach the root, and, like in T, the root of $T2$ is proxy for the nodes of $T2$.

Finally, $\sum_{i \in T'1} c(i) = \sum_{i \in T'} c(i) + 2^{p-1}$ since each node of $T'2$ needs one more message than in T' to reach the root.

Thus we have the following recurrence relation :

$$\begin{cases} \forall p : p \geq 1 : \alpha_{p+1} = 2\alpha_p + 3 \times 2^{p-1} + p \\ \alpha_1 = 2 \end{cases}$$

whence $\alpha_p \approx \frac{3}{4} p \times 2^p + \frac{5}{4} \times 2^p$. From there follows the average complexity per request:

$$c \approx \frac{\frac{3}{4} p \times 2^p + \frac{5}{4} \times 2^p}{2^p} = \frac{3}{4} \log_2 N + \frac{5}{4}$$

5 Node failures

In this section, we address the node failures issue. In order to make the algorithm resilient to these failures, some assumptions have to be made, and actions have to be undertaken by non-failed nodes suspecting such failures.

Assumptions

- At any time, a node can fail. Only fail-stop will be considered; more precisely, when a node fails, it cannot do any action, that is to say can neither send or receive messages, nor process any pending request message; all the messages in-transit towards this node on the communication channels as well as all the information locally stored on this node are lost (however, the constant values *pmax* and *dist* can be stored on a stable storage if node recovery is considered).
- The underlying communication system provides a service ensuring a maximum delay δ for the transmission of messages between any pair of non-failed nodes (in particular, the communication remains possible between every pair of non-failed nodes). The value of δ is available to each node.

Consequences of a node failure

There are three situations in which the failure of a node *i* has consequences :

1. the node *i* is asking the token for itself or is in the critical section,
2. the node *i* is asking the token for the account of its mandator,
3. the node *i* has pending requests in its queue or will receive request messages

(note that the situation iii. is not exclusive from the two others). Situations i. and ii. imply the loss of the token; moreover, situations ii. and iii. imply the loss of all requests - current, pending or future - to be processed by the faulty node and thus has consequences on all the asking nodes having the faulty node as ancestor. Thus, two different types of actions are susceptible to be undertaken by appropriate nodes at appropriate times: token regeneration on the one hand, reconfiguration of the open-cube and request regeneration on the other hand.

Suspicion of failure

Each asking node is able to suspect a failure when, after some delay, it has not received the token. When a node suspects a failure, it undertakes an *enquiry* action, whose conclusion - after a finite time - can be either : the suspicion is *ill-founded*: keep waiting for the token and eventually try another enquiry, or the suspicion is *well-founded*: undertake a regeneration action according to the information obtained

through the enquiry. Let us note that suspecting a failure does not necessarily imply that there *is* a failure; but an enquiry must be *live* (it will conclude in a finite time) and *safe* (if it is well-founded then there is a failure and the regeneration action must be consistent). Below, we examine the enquiry procedure, according to the position of the suspecting node (at the root or not). To begin with, we assume that at most one node can fail: this assumption is only for the sake of clearness and will be removed later.

Root

The root *r* is expecting the token (satisfying $asking_r \wedge \neg token\_here_r$) only when it has lended the token to a node *j* . Let *s* denote the source of the request, that is to say the node whose wish to enter the critical section has triggered the current loan of the token. Two cases are to consider :

1. $j=s$. The token is sent directly from *r* to *s* and vice-versa. If *s* is not down, the root expects the return of the token before a delay equal to $2\delta+e$, where *e* is an estimation of the critical section duration for *s*. When this time is out, *r* suspects the failure and sends an enquiry message to *s*. If *r* does not receive an answer from *s* before the delay 2δ it concludes that *s* is down and regenerates the token. Otherwise the answer comes back and can mean either "wait, *I'm still in the critical section"* or "I've *already sent back the token"*. It is easy to see that, in any case, this enquiry is live and safe.

2. $j \neq s$. The token travels from *r* to *s* via the node *j* and perhaps some other nodes, and comes back directly from *s* to *r.* The token can be lost only if one of these nodes, including *s*, fails before receiving the token (or during the critical section in the case of *s*). The root expects the return of the token before a delay equal to $(pmax+1)\delta+e$. When this time is out, *r* suspects a failure and sends an enquiry message to *s*. If *s* is not down and has received the token, it answers to *r* as in the preceding case. If *s* is not down but has not yet received the token, the only reason is that the token is lost (recall that the token never waits when it is received by an asking node) and thus some node on the path is down; in that case, *s* answers to *r* that the token is lost. If *s* is down, no answer comes back to *r* within a delay of 2δ. In the two latter cases, *r* regenerates the token.

In order to implement this strategy of token regeneration, the root has to be aware of the identity *s* of the source of the request. This information can be added in the request message.

Asking nodes with father≠nil

In the absence of failure, an asking node *i* will eventually receive the token, and upon this receipt, updates its father to the node from which the token has been received (its closest ancestor having acted as *proxy* when the request has moved up to the root) or, if no ancestor was *proxy*, to *nil*. If a failure prevents the node *i* to receive the token, this updating is no more possible and consequently the node *i* has to undertake some action in order to consistently update its father and then regenerate its request. Two cases are to consider :

1. if the closest ancestor *j* having acted as *proxy* when the request has moved up to the root is located between the asking node *i* and the failed node *f*, then the father of *i* must be *j*, as would have been the case in the absence of failure (figure 12). It is now from the responsibility of the node *j* - which is also asking and not receiving the token - to enquire for its new father in order to reconnect the path towards the root.

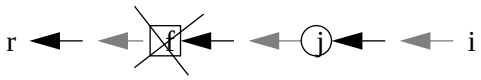


Figure 12

2. if all the nodes on the path between the asking node *i* and the failed node *f* have acted as *transit*, none of them can be the father of *i* (indeed, upon processing the message *request(i)* they have become sons of *i*). Consequently, the node *i* must reconnect the path towards the root by taking, as new father, the former father of *f* (before its failure).

Practically, the node *i* performs a procedure - called *search_father* - when, after a delay at least equal to $2pmax.\delta$ after sending the request, the token has not yet arrived. Its principle is based upon the corollary 2.1, stating that, in an open-cube, $father_i$ is the only node *j* satisfying (i) $dist_i(j)=power(i)+1$ and (ii) $power(j)\geq dist_i(j)$. Thus, the node *i* will perform an iterative research: each phase *d* of this research consists in sending a message *test(d)* to every node located at distance *d* from *i*. It starts with $d=power(i)+1$, and the phase $d+1$ is performed only if $d<pmax$ and the phase *d* did not succeed: this means that the father of *i* cannot be located at a distance $\leq d$ from *i*; hence, while performing the phase *d*, the node *i* evaluates its power as $d-1$. A phase *d* succeeds if one of the nodes at distance *d* from *i* sends to *i* a positive answer: in that case, this node becomes the father of *i*. Let us describe the behavior of a node *k* receiving a message *test(d)* from *i*. There are three cases:

1. $power(k)<d$ and $asking_k=false$ (or *k* is down): the node does not answer (it cannot be the father of *i*). Thus, after a maximum time delay of 2δ the node *i* considers that *k* will not send any answer and discards *k* from its possible fathers.

2. *power(k)<d* and $asking_k$. Since the power of *k* could increase (or *k* could fail) before its current request terminates, it sends back immediately a message *answer*("*try later*"); thus, before the time out 2δ, *i* will receive this answer; the decision of *i* concerning *k* and the current phase will be postponed; some time later, *i* will test *k* again, until it can conclude by case i. or case iii. below.

3. *power(k)≥d*. Even if the node *k* is currently waiting for the token, its power cannot decrease upon the receipt of the latter. The node *k* meets all the requirements to be the father of *i* and consequently sends a message *answer*("*ok*") to *i*. Thus, before the time-out 2δ, the node *i* will receive this answer and conclude the phase *d* with success, terminating the procedure by setting $father_i$ to *k* and regenerating its request.

The node *i* concludes the phase *d* with *no success* if all the nodes at distance *d* have been discarded as possible fathers. If finally the phase *pmax* does not succeed, the node *i* becomes the root: $father_i$ is reset to *nil*, and the token is regenerated by *i*.

Beyond this practical implementation, whose details seem rather intricate (a small example is given at the end of this Section) it is worth to remark that each phase of the *search_father* procedure involves only a subset of the nodes; in fact, only 2^{d-1} nodes are at distance *d* of a given node ($1 \leq d \leq pmax$), independently of the node. The worst case occurs when the searching node has a power 0 and no phase succeeds: in that case, the entire open-cube is finally tested by the testing node. In the average, the number of tested nodes during a *search_father* procedure is $O(\log_2 N)$. This result, and the rather easiness for implementing the reconfiguration procedure and regenerating the requests, is due, in its essence, to the exploitation of structural properties of the open-cube, and particularly to the "locality" property. Another practical consequence of this localization is that, while a *search_father* procedure is executing in a subtree, the normal execution of the algorithm (processing requests) in other parts of the cube can go on.

Concurrent suspicions of failure

It may happen that several nodes having the failed node as ancestor simultaneously suspect a failure (for example if they have issued requests concurrently). In that case, it is possible that a node *i*, while searching at the phase d_i, receives a message *test*(d_j) from a node *j*. Since the power of *i* can increase during the *search_father* procedure, the first reaction could be, for *i*, to postpone its decision concerning *j* by sending a message *answer*(*"try later"*) as in the case ii above. But this policy could cause some deadlock between nodes *i* and *j* since *j* could also receive a message *test*(d_i) from *i*; thus, each of the two nodes will stay in their respective phase d_i and d_j forever. Hence

each node must either send a message *answer*(“*ok*”) or not answer. Here, three cases are to consider :

1. $d_i > d_j$. In that case, $power(j)+1=d_j=dist_j(i)$ and $power(i)=d_i-1 \geq d_j=dist_j(i)$. Since the power of *i* can only increase, *i* must be the father of *j*. Thus, *i* sends immediately the message *answer(“ok”)* to *j*.

2. $d_i < d_j$. In that case, according to the procedure, the node *i* does not answer. However, it is possible to show that, when the *search_father* procedure for *i* terminates, the node necessarily conclude by $father_i:=j$; we will not prove this property, but it clearly allows the node *i* to conclude immediately. To illustrate, consider the following example on the 4-open-cube, where the root **a** failed before the receipt of the concurrent requests of nodes **b** and **c** (Figure 13). Both nodes **b** and **c** start a *search_father* procedure.

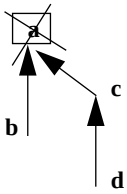


Figure 13 : Initial Situation

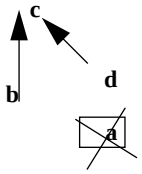


Figure 14 : Final Situation

b sends *test*(1) to **a**, and **c** sends *test*(2) to **a** and **b**.
While waiting in phase 1, **b** receives *test*(2) from **c**, and doesn’t answer.
While waiting in phase 2, **c** receives *test(*1) from **b** and answers “ok”.
Since **c** had no answer in phase 2=*pmax*, it concludes with $father_{\mathbf{c}}:=nil$; since **b** has a positive answer from **c**, it concludes with $father_{\mathbf{b}}:=\mathbf{c}$. The proposed optimization allows **b** to conclude as soon as it receives *test*(2) from **c.**

3. $d_i=d_j$. If *i* answers “ok” to *j*, then *j* will also answer “ok” to *i* and the result will be inconsistent. If, on the contrary, none answers to the other, it can also be inconsistent as shows the following example: the initial situation is the same than the preceding one (Figure 13), and we have the following scenario :

b sends *test*(1) to **a**. (no answer).
b sends *test*(2) to **c** and **d**, and simultaneously **c** starts searching by sending *test*(2) to **a** and **b**.
b receives *test*(2) from **c** while waiting in phase 2 and doesn’t answer.
c receives *test*(2) from **b** while waiting in phase 2 and doesn’t answer.
b and **c** both conclude by setting $father_{\mathbf{b}}:=nil$, $father_{\mathbf{c}}:=nil$ *and both regenerate the token*, hence the conclusion is inconsistent.

This situation occurs when two nodes suspect the same failure and concur to the same father (*nil* in the example). This concurrency can be broken, e.g. by using the node identities : if the identities are totally ordered, the node with the "smallest" identity becomes the father of the other. In the example above, **b** would answer "ok" to **c**, and **c** would not answer to **b**.

Node recovery

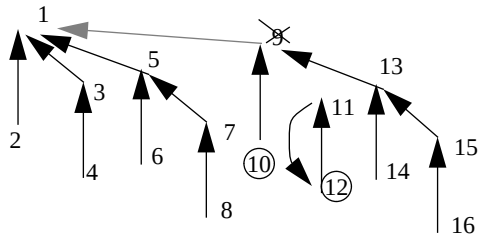
When a node *f* recovers from a failure, it must be consistently reconnected to the open-cube; but all the data stored in its local context has been lost when the failure occurred; however, in order to make its reconnection possible, we assume that the value *pmax* and the array $dist_f$ can be retrieved by *f* (they can be stored once for all at initialization time on a stable memory since their values remain constant). The reconnection action essentially consists in retrieving $father_f$, according to the current open-cube configuration. For that purpose, the node *f* merely executes the procedure *search_father* starting from the phase *d*=1; in other words, upon recovery, a node considers that it is a leaf (recall that each phase of this procedure requires only the knowledge of the data *pmax* and $dist_f$).

But a problem remains, due to the fact that, when a node *f* recovers, it may have some descendants which have not yet concluded a *search_father* procedure during the failure of *f* (e.g nodes which have not required the token during the failure); in that case, *f* is not a leaf, and the reconnection protocol may result in a wrong open-cube structure. However this problem is easily overcome: as long as a descendant does not require the token, this violation of the structure does not matter since it is limited to the subtree rooted at the node *f*. When such a descendant node *i* requires the token, an anomaly can be encountered when the request message reaches *f*. In that case *f* sends back a special *anomaly* message to *i* and, upon receipt of this message, *i* behaves exactly as if its father *f* was down by starting a *search_father* procedure: obviously, such an anomaly means that, after the reconnection of *f*, the latter should not remain the father of *i*. On the other hand, the anomaly is detected by *f* when it performs the "last-son" test involved by the processing of the message *request(i)*: recall that, in an open-cube, the following relation between a node *i* and its father *f* must hold: $power(f) \geq dist_f(i)$ (the equality holds if, and only if, *i* is the last son). Consequently, an anomaly is detected by *f* when, processing *request(i)*, it finds $power(f) < dist_f(i)$.

A small example

The following example illustrates the failure of a node, concurrently detected by two nodes, then the recovery of the failed node and the reparation of an anomaly detected after this recovery. Initially, we have the 16-open-cube, the nodes 10 and 12 have both issued a request, and the node 9 fails before processing their requests. After a finite delay, the two requesting nodes suspect a failure. At that time, the configura-

tion is as follows (Figure 14):



ⓘ : asking node

Figure 14: node 9 is down

The node 10 starts a *search_father* procedure:

node 10: test(1) to 9. Since 9 is down, no answer;
node 10: test(2) to 11, 12.

Concurrently, the node 12 starts a *search_father* procedure:
node 12: test(1) to 11.

While waiting in phase 1, the node 12 receives the message test(2) from 10, and thus concludes its search with $father_{12}$:=10;

The phase 2 for the node 10 does not succeed. Thus it performs the next phase:
node 10: test(3) to 13, 14, 15, 16 : no answer;
node 10: test(4) to 1, ..., 8 : the node 1 answers “ok” and thus 10 concludes its search with $father_{10}$:=1

At the end of these searches, the situation is as follows (Figure 15) :

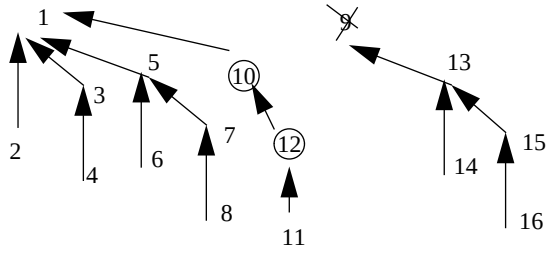


Figure 15

The node 10 processes its own request. Since *power*(1)=4 and $dist_1$(10)=4, 10 becomes the root. Then, after leaving the critical section, 10 processes the request of 12, loans the token and finally gets it back. At that time, the situation is as depicted in the Figure 16.

The node 9 recovers. It starts a *search_father* procedure:

node 9: test(1) to 10. Since *power*(10)=4, 10 answers “ok” and 9 concludes the search with $father_9$:=10 (indicated in dot line on the Figure 16). Consequently, 9

computes its power as $dist_9(10)-1=0$, although it has some descendants. Obviously,

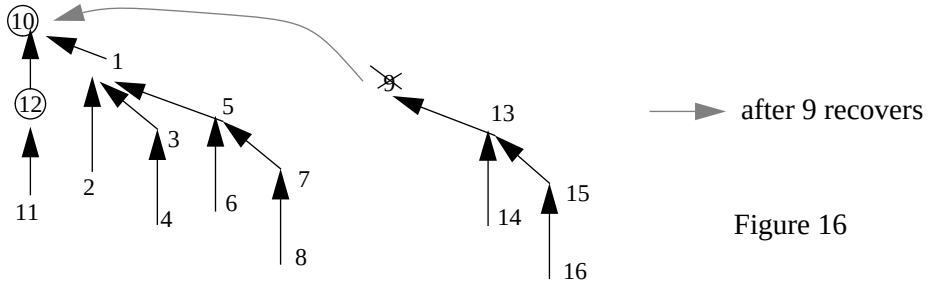


Figure 16

the Figure 16 does not depict an open-cube structure!

Now, the node 13 requests the token. When 9 processes the message *request*(13), the comparison between *power*(9)=0 and $dist_9(13)=3$ raises an anomaly. Thus, 9 sends back to 13 an *anomaly* message. When 13 receives it, it starts a *search_father* procedure starting at phase *power*(13)+1=$dist_{13}(9)=3$

node 13: test(3) to 9, 10, 11, 12. Since *power*(10)=4, the node 10 answers “ok” to 13 which concludes its search with $father_{13}$:=10. Now the situation is (Figure 17):

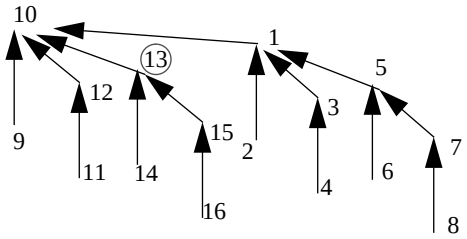


Figure 17 : after reconnection and repair

and the request of 13 is processed in an open-cube context.

Case of several failures

Several failures can occur simultaneously, provided that the network remains able to give the service of bounded delay communication between every pair of nodes (the network is not partitioned). The procedures followed by suspecting nodes is exactly the same as in the single failure case. All the failed nodes will be eliminated from the remaining open-cube as their descendants will issue requests and then undertake their *search_father* procedures.

6 Conclusion

Most of previously known mutual exclusion distributed algorithms using a token and based upon a rooted tree structure can be unified in a general algorithmic scheme [1]. The present algorithm belongs to this class, and on this account it inherits from the general features, in particular the safety and liveness properties. When compared to

other previously known mutual exclusion distributed algorithms - belonging or not to this class - the present one has several advantages : good performances in terms of the maximum number of messages per request, adaptativity of each node workload according to the frequency of requests to enter the critical section, high tolerance to node failures and easy recovery protocol.

For the sake of simplicity, the formal presentation has been made only in the case where it is assumed that nodes do not fail. However the more realistic case of node failures has been carefully analyzed and explained. Let us note that, although this point has not been detailed in the present paper, the complete algorithm, including multiple nodes failures, has been implemented and tested in Estelle on hypercube machine (Intel iPSC/2) with 32 physical nodes; when run with N=32, the average number of overhead messages per failure was 8 msg/failure (300 failures); with N=64 (two processes per physical node), this average number was 9.75 msg/failure (200 failures); these tests - and many others which are not reported here- confirm the average number of $O(\log_2 N)$.

These good results are mainly due to the stability and locality properties of the open-cube structure introduced here. We think that this original structure could be favorably used as a communication tool, offering good performances and high tolerance to node failures, in many other distributed applications.

References

[1] J.M. Hélary, A. Mostefaoui, M. Raynal.
A general scheme for token and tree based distributed mutual exclusion algorithms.
Research Report #1692, INRIA-IRISA, Univ. of Rennes, France (may 1992).
Submitted to publication

[2] M. Naimi, M. Trehel.
An improvement of the log(n) distributed algorithm for mutual exclusion.
Proc. 7th IEEE Int. Conf. on Dist. Comp. Systems, Berlin, (1987), pp. 371-375.

[3] M. L. Neilsen, M. Mizuno.
A dag based algorithm for distributed mutual exclusion.
Proc. 11th IEEE Int. Conf. on Dist. Comp. Systems, Dallas, (1991), pp. 354-360.

[4] K. Raymond.
A tree based algorithm for distributed mutual exclusion.
ACM Trans. on Computers Systems, Vol. 7,1, (1989), pp. 61-77.

[5] M. Raynal.
A simple taxonomy for distributed mutual exclusion algorithms.
ACM Op. Systems Review, Vol. 25,2, (1991), pp. 47-50.

[6] B. Sanders.
The information structure of distributed mutual exclusion algorithms.
ACM Trans. on Prog. Languages and Systems, Vol. 5,3, (1987), pp. 284-299.

[7] J.L.A. Van de Snepsheut.
Fair mutual exclusion on a graph of processes.
Distributed Computing, Vol. 2, (1987), pp. 113-115.

Unité de recherche INRIA Lorraine, technopôle de Nancy-Brabois, 615 rue du jardin botanique, BP 101, 54600 VILLERS-LÈS-NANCY
Unité de recherche INRIA Rennes, IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 46 avenue Félix Viallet, 38031 GRENOBLE Cedex 1
Unité de recherche INRIA Rocquencourt, domaine de Voluceau, Rocquencourt, BP 105, LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
Inria, Domaine de Voluceau, Rocquencourt, BP 105 LE CHESNAY Cedex (France)

ISSN 0249-6399