A Dynamic Distributed Algorithm for Read Write Locks (extended abstract)

Soumeya Hernane, Jens Gustedt, Mohamed Benyettou

A Dynamic Distributed Algorithm for Read Write Locks

(extended abstract)

Soumeya Leila Hernane, Jens Gustedt, Mohamed Benyettou

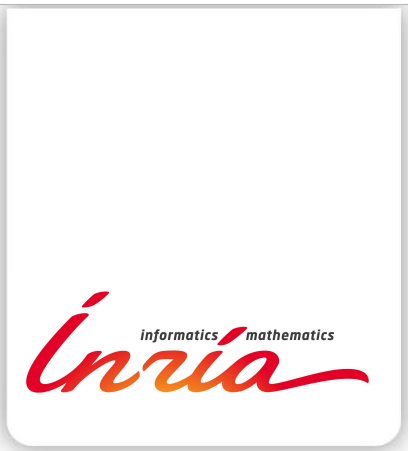
RESEARCH REPORT

N° 7798

November 2011

Project-Team AlGorille

ISSN 0249-6399 ISRN INRIA/RR--7798--FR+ENG



A Dynamic Distributed Algorithm for Read Write Locks (extended abstract)*

Soumeya Leila Hernane[†‡], Jens Gustedt[†§], Mohamed Benyettou[‡]

Project-Team AlGorille

Research Report n° 7798 — November 2011 — 12 pages

Abstract: In this paper, a new algorithm that extends Naimi-Trehel token-based mutual exclusion is proposed. Contributions are twofold.
First, our algorithm evolves within an changing environment; processes can join and leave the system while the ***parent*** tree is in ongoing transformation and, while requests accessing the critical section are inserted. Both the tree structure and the original algorithm are amended. We impose new rules and add new variables to the original structure, so as to maintain the connectivity of the ***parent*** tree as well as the ***Next*** chain, whatever circumstances.
Second, the possibility to share the token between processes is introduced. This allows either multiple concurrent readers or a single writer to enter the critical section. In the ***Next*** chain of successive readers, a newly introduced ***Reading handler*** ensures that all following requesters for read access may enter the critical section simultaneously, and keeps the token as long as at least one reader is in the critical section.
In all cases, our approach can be implemented such that it keeps an average complexity of $O(log(n))$ messages per request.

Key-words: Distributed locks, read-write locks, dynamic locking

---

* This article is accepted for publication in the proceedings of PDP 2012.

† LORIA, Nancy, France
‡ University of Science and Technology, Oran, Algeria
§ INRIA Nancy – Grand Est, France

RESEARCH CENTRE NANCY – GRAND EST

615 rue du Jardin Botanique
CS20101
54603 Villers-lès-Nancy Cedex

Un algorithme pour la gestion dynamique de verrous lecture/écriture (résumé étendu)

Résumé : Nous proposons dans ce papier une extension de l'algorithme d'exclusion mutuelle, à base de jeton de Naimi-Tréhel. Notre contribution se présente sous deux angles différents.

Premièrement, notre algorithme progresse dans un environnement inconstant. Les processus peuvent joindre et quitter le système. En même temps, l'arbre ***parent*** subit des transformations au fur et à mesure que les requêtes d'accès à la ***section critique*** sont insérées. Les modifications portent aussi bien sur l'algorithme que sur la structure arborescente ***parent*** ainsi que sur la chaîne ***Next***.

De ce fait, Nous imposons de nouvelles règles et de nouvelles variables aux structures de départ, de sorte que la connexité de l'arbre ***parent*** ainsi que celle de la chaîne ***Next*** soient maintenues.

Deuxièmement, nous rendons possible l'agencement du jeton partagé à l'exclusif. Ainsi, la ***section critique*** devient accessible, soit par plusieurs lecteurs concurrents soit par un seul écrivain. Dans la chaîne ***Next***. le ***gestionnaire des lecteurs*** est introduit pour assurer l'entrée en ***section critique*** de tous les lecteurs successifs. De même, le ***gestionnaire des lecteurs*** garde le jeton partagé tant qu'au moins un lecteur est en ***section critique***.

Dans tous les cas de figure, l'implantation de notre approche garantit une complexité logarithmique de l'ordre de *O(log(n))* messages par requête.

Mots-clés : verrou repartie, verrous lecture/écriture, verrouillage dynamique

1 Introduction and Overview

Several protocols have been proposed in the literature to solve mutual exclusion problems within distributed systems. They can be either permission-based (Lamport [1], Maekawa [2], Ricart-Agrawala [3]) or token-based (Naimi-Trehel [4], Raymond [5]). The first set allows entering the critical section (CS) after receiving permission from other nodes. They incur a high communication overhead. The second set conditions the entrance into the CS by the possession of a token which is passed between nodes. They differ from each other in the way the request messages are routed to reach the token. This group of algorithms is tree-based and many of them exhibit a *O(log (N))* complexity in terms of the number of messages per request. Our work focuses on this class of algorithms for the sake of this message complexity.

The distributed algorithm of Naimi-Trehel [4] based on path reversal is *the* benchmark for mutual exclusion. In this paper, we will merely refer to it in the version of Naimi-Trehel-Arnold [6], that additionally provides the proofs for the properties that we will use. Many other extensions of this algorithm have been proposed in the literature, we mention a few of them. A Fault tolerant token based mutual exclusion algorithm using a dynamic tree was presented by Sopena [7]. It improves over Naimi-Trehel [4] by ensuring a lower cost in terms of messages in the presence of failures. Quinson-Vernier [8] provide a byte range asynchronous locking of the Naimi-Trehel algorithm based on sub-queues when partial locks are requested. Wagner-Mueller [9] have proposed token based read-write locks for distributed mutual exclusion. The protocol distinguishes read and write requests and presents good performance. However, up to our knowledge it has no proof of *safety* and *liveness*. In [10], Mohamed & Naimi have proposed a solution of *Readers/Writers* in the distributed systems. A specification of the algorithm was given by predicates used to express the axioms and in order to prove the algorithm. However, the complexity has not been studied. Based on the same assumptions as the original algorithm of Naimi-Trehel, this work presents an extended algorithm. Gustedt's *DHO* [11] presents a theoretical programming paradigm and an interface, for locking and mapping data in both read and write modes. It neither provides nor forces any particular algorithm or protocol read-write locks that would be used by that abstract interface.

Contributions in this paper are twofold. First, we extend the original Naimi-Trehel algorithm so that the ***parent*** tree and the ***Next*** chain (for definitions of these terms please see the original Naimi-Trehel algorithm [10]) evolve in a dynamic environment, while maintaining *safety* and *liveness* properties, see Section 2. Second, in addition to exclusive locks hold by a single writer, we allow multiple readers to enter the CS, so that the algorithm covers both exclusive and concurrent locks, Section 3. In this framework, we propose a new data structure that handles additional features. Finally, in Section 4 we conclude, in particular to advertise the use of this approach within an API that handles resources in large scale distributed systems.

2 Mutual exclusion algorithm with volatile processes

The Naimi-Trehel algorithm is based on a distributed queue along which the token circulates, and on a distributed tree structure for queries. The query tree is rooted at the tail of the queue to allow to append new requests to the queue at any moment. The basics of the Naimi-Trehel algorithm can be found in [4].

2.1 Concurrent requests

Within Naimi-Trehel's algorithm, a given ***parent*** p_1 of two processes p_2 and p_3 can be queried simultaneously by both. In such case, if p_1 treats p_2 first, p_3 is put on hold and is disconnected temporarily from the tree. An example from [12] is shown in Fig. 1. Initially, Fig. 1(a), p_1 holds the token and p_3 claims the CS by sending a request to its ***parent***. in turn, p_1 updates its ***parent*** and its ***Next*** to p_3, Fig. 1(b). Then, p_2 and p_5 claim the CS. They send request to p_1 and set forthwith their ***parent*** to *null*. So, p_1 points towards p_2 and forwards the request to p_3, Fig. 1(c). Meanwhile, p_5 waits and is disconnected from the tree.

Once p_1 sent p_2's request to p_3, it switches to p_5's request. Thus, it forwards the request to p_2 and sets its ***parent*** to p_5. Meanwhile, p_2 is cut from the tree, Fig. 1(d). In Fig. 1(e), p_2's request is achieved and p_5's ends in Fig. 1(f).

We notice that, processes set their ***parent*** variable to *null* as soon as they forward the request. Within a system of *n* processes, *n-1* processes may request the token concurrently and this will generate *n* disjoint ***parent***'s trees. This example was presented in [12] in the context of node failures. In our proposal, processes handle one request at a time (Section 2).

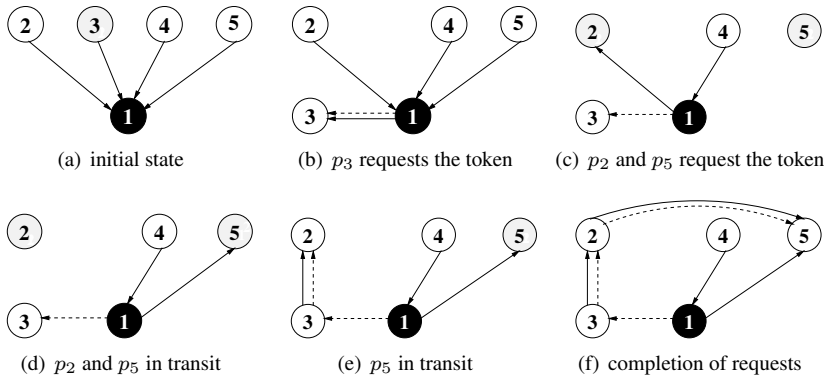


Figure 1: Example of concurrent requests of Naimi-Trehel's Algorithm [12]

We provide an extension of the algorithm that allows processes to enter or to leave the system. Note that if a process wants to enter the group, it just has to choose a relative ***parent*** to join the tree. It will then be connected to the system. So the difficulty in node dynamicity lies in the departure of processes, to which we thus focus in the sequel.

To that aim, we amend the original algorithm by requiring that processes are never cut from the ***parent*** tree, whatever the circumstances. This helps to keep all links alive if a given process leaves the system. We assume that a process does not support a new request before the previous one is completed.

2.2 Atomic Operations

In this subsection we amend the structure and query processing of the mutual exclusion algorithm, such that we avoid overlapping of operations that relate to a given process p_i. In order to meet new requirements of the extended algorithm, we add to the set of variables as defined above:

***Send*:** A Boolean that refers to the process state, initially set to *false*. If *true*, then the process has sent a request that is not completed.

***Req*:** A list of received requests by a process, initially empty.

***Wait_ack*:** A Boolean that refers to an acknowledgment (ACK) set by a given process, initially set to *false*.

Previous*:** Each process knows who will hold the token before him, it is easily updated simultaneously as ***Next. ***Next*** and ***Previous*** form a doubly linked list.

Let $\{p_1..p_i..p_n\}$ a finite set of processes within a distributed system. Processes are bound to states related to handling requests, not on the token possession. A process p_i can be:

Idle*:** If it has a ***parent, it can either send a request to it and then switch ***Send*** to *true*, or receive a request from any other process that points on p_i as ***parent***. ***Req*** list is *null*.

Sending*:** p_i sent a request that is not yet completed. It is not ready to receive any request as ***Send is *true*. ***Req*** is *null*.

Busy*:** p_i handles and registers a request. It is not ready to register others, however it can put them in ***Req list.

We consider a branch of the ***parent*** tree $\{..p_{j-1}, p_j, p_{j+1}..p_n\}$ as a sequence of processes such that p_n is the root, whereas p_{j+1} is the parent of p_j. Hence, p_j points towards p_{j+1}. Suppose initially, p_0 is the root that holds the token.

As in the original Naimi-Trehel algorithm, assume that at a time *t*, p_j requests the token from its ***parent*** p_{j+1}. Additionally we will always require that, before, it sets the ***Send*** variable to *true*.

Conversely, if p_{j+1} has sent a request that is not yet achieved, it puts p_j's request on hold and stores it in ***Req*** list. As soon as p_{j+1} is available, it starts processing the p_j's request. If its ***parent*** exists (*j+2*), p_{j+1} sends it to p_j and waits for ACK. Then, p_j resends the request to p_{j+2}.

Similarly, if p_{j+2} is not *busy*, it sends its ***parent*** to p_j and waits for ACK. Otherwise, it puts the pending request in ***Req*** list. The processing is carried on until p_n is claimed. Then, p_n informs p_j that it is the root of the tree and, it registers p_j as the ***Next***. In turn, p_j sets its ***Previous*** to p_n. Processes $p_{j+1}..p_n$ update their ***parent*** to p_j.

Suppose that p_{j+1} is a parent of another process p_i. As in Figure 1, p_1 is the parent of p_2 and p_5. Suppose that p_i sends a request to p_{j+1}. Knowing that p_{j+1} has not yet received an ACK from p_j, it puts the p_i request in **Req** list.

The processing ends when p_j sends an ACK to its ***parent*** (p_{j+1}) and to all its ascendants into the ***parent***'s tree. It was then that processes $p_{j+1}..p_n$ handle ***Next*** requests that are in ***Req*** list, according to the FIFO policy. Then, p_j sets its ***parent*** variable to *null* and its ***Send*** value to *false*.

Thus, we avoid the overlap of requests. Likewise, we ensure consistency in the logical order of requests.

2.3 Average Message Complexity

The average number of messages needed for a request to reach a root within Naimi-Trehel's algorithm depends on the height of the request tree process. By using the arguments from [6] we will show that this results in an average message complexity of *O(log(N))* , where N is the whole number of processes in the system.

In the original algorithm, a request is propagated through the ***parent*** tree from a son p_j to ***parent*** p_{j+1}, as the requesting process p_j sends, itself a message to each process in the ***parent*** tree. In our algorithm, in addition each process $(p_{j+1}..p_n)$ in the branch of the tree, sends a message to the requesting process p_j containing its ***parent***. Then, from the end of the tail, p_j sends acknowledgments to its ***parent*** and to all its ascendants. So in summary, the message complexity per request grows by 3 and will be remain *O(log(N))* on average, as in the original algorithm.

Notwithstanding, the extra cost of the modified algorithm does not depend of the whole number of processes and thus, remains *O(log N)* . By this addition, we are aiming to provide a mutual exclusion algorithm in a dynamic environment.

2.4 Formal Description

There is no common clock in our system. However, we introduce a local clock to determine the sequence of events that happen at a given process ρ. Values are assigned to variables described above for each state of the process. By that, atomicity for requests is ensured, while overlap is avoided.

Let T=$(t_0, \ldots, t_n)$ an ordered sequence of times with $t_i < t_{i+1}$ refering to request reception by ρ, and let T'=$\{\ldots, t'_i, \ldots\}$ be the set of times of the acknowledgments of those registered requests. Note that ρ doesn't record a new request until it received the ACK of the previous one. We have

$$t_0 < t'_0 < \cdots t_i < t'_i \cdots < t_n < t'_n.$$

Let $State(\rho, t)$ be the state of ρ at time t. A process ρ is in *Idle* state if it is neither sending, or receiving requests, nor waits for an ACK. If ρ is *Busy*, it waits for an ACK from the process whose request has been registered. If ρ is *Sending*, it has sent a request to its ***parent*** and has not finished sending ACKS. In summary, for each *t* in $[t'_i, t_{i+1}]$ the following hold:

$$\text{STATE}(\rho, t) = \textit{Idle} \Rightarrow \begin{cases} \text{SEND} = \textit{false};\ \text{REQ} = Null \\ \text{WAIT\_ACK} = \textit{false} \end{cases}$$

$$\text{STATE}(\rho, t) = \textit{Busy} \Rightarrow \begin{cases} \text{SEND} = \textit{false};\ \text{REQ} \neq Null \\ \text{WAIT\_ACK} = \textit{true} \end{cases}$$

$$\text{STATE}(\rho, t) = \textit{Sending} \Rightarrow \begin{cases} \text{SEND} = \textit{true};\ \text{REQ} = Null \\ \text{WAIT\_ACK} = \textit{false} \end{cases}$$

Also, ρ must be in the *Idle* state before sending or recording a given request, *i.e* for $t_1 < t_2$, if STATE(ρ, t_1) and STATE(ρ, t_2) are *Sending* or *Busy*, then there is an intermediate t' with $t_1 < t' < t_2$ such that STATE(ρ, t')=*Idle*.

2.5 A Dynamic Mutual Exclusion Algorithm

We impose the condition that the tree still always connected. For that, we bring additional modifications that will be dealing with the exit strategy for processes. We add the following variables to the previous structure, to maintain the connectivity of the tree.

Child$[1..m]$**:** An array allocated dynamically, containing *m* process IDs that have ρ as parent.

Del: Boolean, initially *false*. It is permanently set to *true* once ρ leaves the system.

***End_del*:** Boolean, initially *false*. It is set to *true* while ρ is negotiating its departure, and goes back to *false* at the end of that process.

Several events occur before a process ρ leaves the system. We summarize them in chronologically as follows:

1. If ρ is in the CS, it leaves it. Then, it grants its ***Next*** if it exists and, this will be the new root.
2. ρ broadcasts a message to its children. It asks each of them if it is also leaving, *Sending* or *Busy*. ρ waits for a message from each of them, this will avoid deadlock.
3. If one of the children is now leaving the system, its ***End_del*** variable is *true*. Thus, ρ waits until the ***End_del*** of the ***Child*** is *false*. It will receive a message subsequently.
4. Once all children are notified, they won't send any request or message to ρ, until they receive a new ***parent***.
5. Once ρ becomes *Idle* and its ***Req*** list becomes empty, it sets **Del** and ***End_del*** to *true*.
6. If ρ is the current root (and thus ***Next*** was empty in 1), then ρ elects one of its children R to become the new root.
7. ρ sends either this new root R or its ***parent*** to its children.
8. All children update their ***parent*** variable. Thus, the links of the tree are well maintained.
9. If they are set, ρ notifies its ***Next*** and its ***Previous*** such that the doubly linked list can be shortcut by them.
10. At the end of the leaving process, ρ sets ***End_del*** to *false* and, if it was non-null, notifies ***parent*** that it has finished the departing procedure.

By that, as the ***parent*** tree remains connected, the ***Next*** and the ***Previous*** chains links are preserved.

We note that *Liveness* and *safety* properties are maintained. The complete proof of these properties will be presented in the full version of this paper.

Procedure `Exit` exit strategy for node ρ

if *(ρ is in CS_access)* **then**
  CS _release;
  Send grant_message to ***Next***;
for $i \leftarrow 1$ **to** m **do**
  $Send\_mess\_del$ to $child[i]$;
  Wait for child_message; /* ρ waits if its children are leaving */
while *((state($\rho \neq$* Idle*)* $\wedge$ *(Req* $\neq$ null*))* **do**
  Wait;
Del $\leftarrow$ *true*; /* Start to leave */
End_del $\leftarrow$ *true*;
if *(no* **parent***)* **then**
  Elects a ***Child*** as ***parent*** /* ρ elects and sends a new ***parent*** among the children*/
for $i \leftarrow 1$ **to** m **do**
  Send ***parent*** to $child[i]$;
if *(***Next** $\wedge$ **Previous***)* **then**
  /* ρ keeps the link of the ***Next Previous*** chain */
  Send ***Next*** to ***Previous***;
  Send ***Previous*** to ***Next***;

Procedure `Child_parent_receive`;
The child of ρ receives a new parent

if *new_parent* $\neq$ **parent then**
  parent $\leftarrow$ new_parent;
else
  parent $\leftarrow$ *null*; /* The child of ρ was elected as a new parent */

3 Read Write Dynamic Algorithm

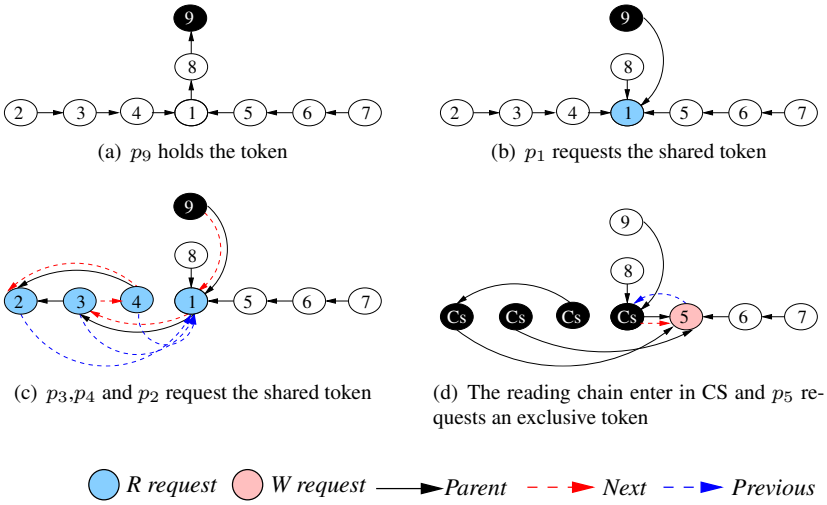
This section relates to the second contribution in which the sharing of the token is introduced to the previous extended algorithm, resulting in a a read-write mutual exclusion algorithm with volatile processes. Contrary to the contribution of [9], in our approach we introduce a ***Reading handler*** of concurrent requests. It will cope with the volatility of processes. However, the exit strategy of the ***Reading handler*** will be dealt with later. First of all, we add the following local variables to the structure of a given process ρ, see Section 2.2:

***Type*:** Request type, W (exclusive write) or R (inclusive read).

***Reading handler*:** Referring to the first process that requests a shared token after an exclusive one. Initially set to *null*.

Reader counter*:** A counter that is incremented if ρ is a ***Reading handler and at each subsequent read request. Set to zero, initially.

In the following, we assume that the reader chain is a ***Next*** chain of pending successive read requesters. Initial values are added within the initialization() function.



(a) p_9 holds the token

(b) p_1 requests the shared token

(c) p_3,p_4 and p_2 request the shared token

(d) The reading chain enter in CS and p_5 requests an exclusive token

Figure 2: Example of mixed requests

3.1 Handling requests

Within the basic algorithm of Naimi-Trehel, the root of the tree updates its ***Next*** to ρ when the latter sends a request to its ***parent*** [4]. We focus on the maintenance of the ***Next*** chain. Assume a chain of processes $\{p_0..p_{j-1}, p_j, p_{j+1}..p_n\}$ that have pending requests and where p_{j+1} is the ***Next*** of p_j and p_{j-1} is the ***Previous*** of p_j. At that time, p_j and p_{j-1} exchange their respective type. Then, p_{j-1} sets its ***Next*** to p_j. Several cases may occur:

- If p_{j-1}'s type is W and p_j's type is R then p_j sets the ***Reading handler*** to itself, its ***Previous*** to p_{j-1} and increments the ***Reader counter***.
- If p_{j-1}'s and p_j's type are R then, p_j sends a message and sets its ***Previous*** to the ***Reading handler***. This one increments the ***Reader counter***. Note that the ***Reading handler*** can be within $[p_0..p_{j-1}]$.
- If p_{j-1}'s type is R and p_j's type is W then p_j sets its ***Previous*** to the ***Reading handler***. The latter sets its ***Next*** to p_j.
- If p_{j-1}'s and p_j's type are W then they update their ***Next*** and ***Previous*** variable respectively, just as before in the case that we only maintained exclusive access to the CS.

Figure 2 shows an example of 4 consecutive pending requests. p_1 is the ***Reading handler*** of p_2, p_3 and p_4. Figure 2(c) summarizes result of events of read requests that have been launched consecutively by p_3, p_4 and p_2. In Fig. 2(d), the reader chain enters in the CS and p_5 sends a write request. In terms of complexity, three additional messages are required, two for the type exchange and one to send the identification of the ***Reading handler***. Thus, the message complexity of handling requests remains *O(log (N))* .

3.2 Entering the CS

Within a chain $\{p_0..p_{j-1}, p_j, p_{j+1}..p_n\}$ of successive read requesters, as soon as p_0 (the ***Reading handler***) accesses the CS, it invites its ***Next*** to do so, and so on until the last read requester p_n. When it enters the CS, each process in the reader chain sends a message to the ***Reading handler*** which in turn increments the ***Reader counter***. So, we have only one additional message per participating process in terms of complexity.

Procedure `CS_accessing` ρ enters the CS

```
if (ρ's type = R) then
    if (ρ = Reading handler) then
        Reader counter ++;
    else
        Send access_message to Reading handler;
    if (NEXT_access_type = R) then
        Send CS _accessing message to Next;
```

3.3 Token release

Once they release the token, processes send a message to the ***Reading handler***, which decrements the ***Reader counter***. We note that the *previous* of p_{n+1} is the ***Reading handler*** (p_0). The ***Reading handler*** doesn't release the token to the first process requesting an exclusive token following the reader chain, until the ***Reader counter*** is zero. Obviously, a process ρ never shares an exclusive token with a shared one.

Procedure `CS_release` ρ releases the CS

```
if (ρ's type = R) then
    if (ρ ≠ Reading handler) then
        Send CS _release to Reading handler;
    else
        Reader counter- -;
        if (Reader counter = 0) then
            /* Release the token */;
            Token_release;
            send CS _accessing message to Next;
else
    /* Release the token */;
    Token_release;
    send CS _accessing message to Next;
```

The correctness of the algorithm for the extension to shared tokens is will be available in the full version of this paper.

4 Conclusion

A new approach of an extended model for a class of distributed mutual exclusion algorithms has been presented in this paper. Our model is an extension of the Naimi-Trehel token-based algorithm. It presents two main benefits; first, it allows a continuous evolution of dedicated distributed environment and secondly, it offers two different access modes to the CS. This contribution should reveal useful in context of grid computing, peer to peer computer networking or any large scale distributed infrastructure. In those systems, resources are set to disappear and reappear. The first modification of the algorithm provides this feature. Furthermore, our approach has a low (=logarithmic) complexity in expected number of messages per operation.

We are currently implementing our approach and plan to measure the ability of that algorithm to provide a robust and scalable framework such as grids. In ongoing work, we aim to join the current algorithm to an application programming interface, Data Handover *DHO* [11], which is a framework for an efficient management for locking and mapping data (an subranges of it) in both read and write modes. Both, our first and second modification of the Naimi-Trehel algorithm will support these ideas. The *DHO* functions allow the user to insert requests, to release resources and to leave the system, seamlessly to the internal structure of the algorithm provided in our previous work [13].

References

[1] L. Lamport, "Ti clocks, and the ordering of events in a distributed system," *Commun. ACM*, vol. 21, pp. 558–565, July 1978.

[2] M. Maekawa, "An algorithm for mutual exclusion in decentralized systems," *ACM Trans. Comput. Syst.*, vol. 3, pp. 145–159, May 1985.

[3] G. Ricart and A. K. Agrawala, "An optimal algorithm for mutual exclusion in computer networks," *Commun. ACM*, vol. 24, pp. 9–17, January 1981.

[4] M. Naimi and M. Trehel, "How to detect a failure and regenerate the token in the $\log(N)$ distributed algorithm for mutual exclusion," in *Proceedings of the 2nd International Workshop on Distributed Algorithms*. London, UK: Springer-Verlag, 1988, pp. 155–166.

[5] K. Raymond, "A tree-based algorithm for distributed mutual exclusion," *ACM Transactions on Computer Systems*, vol. 7, pp. 61–77, 1989.

[6] M. Naimi, M. Trehel, and A. Arnold, "A $\log(N)$ distributed mutual exclusion algorithm based on path reversal," *J. Parallel Distrib. Comput.*, vol. 34, pp. 1–13, April 1996.

[7] J. Sopena, L. B. Arantes, M. Bertier, and P. Sens, "A fault-tolerant token-based mutual exclusion algorithm using a dynamic tree." in *Euro-Par'05*, 2005, pp. 654–663.

[8] M. Quinson and F. Vernier, "Byte-range asynchronous locking in distributed settings," in *17th Euromicro International Conference on Parallel, Distributed and network-based Processing - PDP 2009*, Weimar, Germany, 2009. [Online]. Available: http://hal.inria.fr/inria-00338189/en/

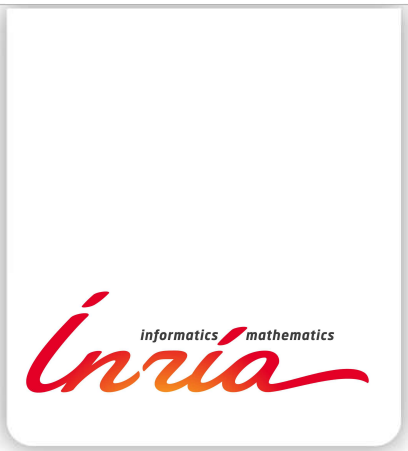
[9] C. Wagner and F. Mueller, "Token-based read/write-locks for distributed mutual exclusion," in *Proceedings from the 6th International Euro-Par Conference on Parallel Processing*, ser. Euro-Par '00. London, UK: Springer-Verlag, 2000, pp. 1185–1195.

[10] A. Mohamed and B. Naimi, "Distributed concurrent control with readers and writers," in *International Conference "PARALLEL 90", Springer Verlag, Southampton United Kingdom*, 1990.

[11] J. Gustedt, "Data handover: Reconciling message passing and shared memory," in *Foundations of Global Computing*, ser. Dagstuhl Seminar Proceedings, J. L. Fiadeiro, U. Montanari, and M. Wirsing, Eds., no. 05081, Dagstuhl, Germany, 2006. [Online]. Available: http://drops.dagstuhl.de/opus/volltexte/2006/297

[12] M. Naimi and M. Trehel, "How to detect a failure and regenerate the token in the $\log(N)$ distributed algorithm for mutual exclusion," in *WDAG*, ser. Lecture Notes in Computer Science, J. van Leeuwen, Ed., vol. 312. Springer, 1987, pp. 155–166.

[13] S. Hernane, Leila, J. Gustedt, and M. Benyettou, "Modeling and Experimental Validation of the Data Handover API," in *Advances in Grid and Pervasive Computing*, ser. LNCS, Riekki, J., Ylianttila, M., Guo, and M., Eds., vol. 6646. Oulu, Finland: Springer, May 2011, pp. 117–126. [Online]. Available: http://hal.inria.fr/inria-00547598/en



RESEARCH CENTRE
NANCY – GRAND EST

615 rue du Jardin Botanique
CS20101
54603 Villers-lès-Nancy Cedex

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

ISSN 0249-6399