

A Timer-free Fault Tolerant K -Mutual Exclusion Algorithm

Mathieu Bouillaguet — Luciana Arantes — Pierre Sens

N° ????

mars 2009

Thème COM



*R*apport
de recherche

A Timer-free Fault Tolerant K -Mutual Exclusion Algorithm

Mathieu Bouillaguet , Luciana Arantes , Pierre Sens

Thème COM — Systèmes communicants
Équipes-Projets Regal

Rapport de recherche n° ???? — mars 2009 — 19 pages

Abstract: This paper proposes a fault tolerant permission-based k -mutual exclusion algorithm which does not rely on timers, nor on failure detectors, neither does it require extra messages for detecting node failures. Fault tolerance is integrated in the algorithm itself and it is provided if the underlying system guarantees the *Responsiveness Property* (\mathcal{RP}). Based on Raymond's algorithm [Ray89], our algorithm exploits the REQUEST-REPLY messages exchanged by processes to get access to one of the k units of the shared resource in order to dynamically detect failures and adapt the algorithm to tolerate them.

Key-words: No keywords

Un Algorithm de K -exclusion mutuelle tolérant aux fautes sans temporisateur

Résumé : Pas de résumé

Mots-clés : Algorithme réparti, tolérance aux fautes, exclusion mutuelle

1 Introduction

Distributed mutual exclusion problem involves Π processes which communicate via message passing and need to access a shared resource by executing a segment of code called the critical section (CS). The problem requires that only one process be in the critical section at any given time. The k -mutual exclusion problem is a generalization of the mutual exclusion one by considering k units of the shared resource. It then allows at most k processes to access these units simultaneously, i.e., one process per unit. Therefore, a k -mutual exclusion algorithm must guarantee that at most k processes can be in its critical section at any time (*safety property*) and that every request for critical section execution is eventually satisfied (*liveness property*).

A number of algorithms have been proposed to solve the distributed k -mutual exclusion algorithms which can be classified into two main categories: permission-based [Ray89], [PMP⁺96], [HJK93] and token-based [SR92], [MBB⁺92], [BV95]. The first category of algorithms is based on the principle that a node gets into critical section only after having received permission from all or a subset of the other nodes of the system. In the second one, the possession of the single token or one of the tokens gives a node the right to enter into the critical section. Although token-based algorithms usually present good performance in respect to the number of messages, they suffer from poor resiliency. On the other hand, due to redundancy of messages, some permission-based algorithms are inherently fault tolerant or can be more easily adapted to become resilient to failures.

We present in this paper a fault tolerant permission-based k -mutual exclusion algorithm. The choice for a permission-based is justified by the reason mentioned above. Our k -mutual exclusion algorithm is inspired by Raymond's algorithm [Ray89], where a node that wants to access one of the k units of the shared resource sends a request to the other processes and thus waits for a sufficient number of permissions (REPLY messages) that ensures that no more than $k - 1$ of the other processes are currently executing the critical section. The novelty of our solution is that fault tolerance is incorporated in the algorithm itself. Unlike the majority of fault tolerant mutual exclusion algorithms [MS94], [NM94], [CSL90], our algorithm does not require extra messages, nor does it need timers for checking the liveness of nodes or broadcasting information about crashes. Such an information is included in the messages of the mutual exclusion algorithm. Furthermore, contrarily to some k -mutual exclusion algorithms [Ray89],[SR92] where the effectiveness of the algorithm drops at every failure because the number of processes that can concurrently execute the CS decreases as well, our fault tolerance approach guarantees that it is always possible to have k processes in the CS simultaneously, despite failures. A third worthy remark is since crashes are dynamically detected by our algorithm, processes can faster satisfy their request to execute the CS when failures occur: both the number of request messages and the number of waited replies of a CS request depend on the current number of non faulty nodes.

Basically, the idea of our approach is that, besides information about the k -mutual exclusion algorithm itself, each reply from p_j to p_i 's request includes information about all nodes that do not reply to p_j 's own request, i.e., those nodes that might be faulty. By gathering information received from these replies, p_i can detect which are the nodes that have crashed. Our algorithm tolerates

at most $k - 1$ faults ($f < k$). However, detection of failures is only possible if the underlying system satisfies a property which we denote the *Responsiveness Property* (\mathcal{RP}), based on the work of Mostefaoui et.al [MMR03]. In other words, our approach relies on an additional assumption which characterizes the synchrony of the system. The \mathcal{RP} property states that, for every process p_k , since the beginning of the algorithm execution, there is a set of at least $f + 1$ processes such that each process p_j of its set has always got a reply from p_k to its request until p_j possibly crashes. Since, our algorithm tolerates f faults and the channels are reliable, the \mathcal{RP} then guarantees if p_i waits for $|\Pi| - f$ messages (where Π is the number of initial nodes of the system) then among the replies received by p_i from p_j to its request, there will be at least one whose message contains the information that p_k has answered to p_j 's last request, if p_k has not crashed. Consequently, p_k will not be suspected as faulty by p_i . On the contrary, if p_k has crashed, the information that it does not reply to requests will eventually be included in all replies received by p_i , which will thus conclude that p_k is faulty. Interestingly, that without any additional failure detection mechanism but just based on the information included in the reply messages of the algorithm and the \mathcal{RP} our k -mutual exclusion algorithm ensures (1) that every crash is eventually detected by every correct process and (2) no correct process is suspected. It is worth remarking that the conjunction of (1) and (2) is respectively equivalent to the *strong completeness* and *perpetual strong accuracy* assumptions of the perfect failure detector \mathcal{P} [CT96], which is sufficient to solve fault-tolerant mutual exclusion problem [DGFGK05].

One could argue that the fault tolerance provided by our k -mutual exclusion algorithm does not work for any kind of system. That is true, but if the system presents the \mathcal{RP} , fault tolerance is offered without much overhead since it is inserted in the k -mutual exclusion itself. Another important point to emphasize is that several environments satisfy the \mathcal{RP} such as a Grid platform where the latency between two nodes of different clusters is higher than the latencies from two nodes of the same cluster, or a fully-connected ring organized system where latencies between no-neighbors nodes are higher than those between neighbors nodes.

The paper is organized as follows. Section 2 defines the computation model. Examples of platforms that satisfy \mathcal{RP} are given in section 3. Our fault-tolerant algorithm is described in Section 4. Simulation performance results are present in Section 5. Some related work is discussed in section 6. Finally, section 7 concludes the paper.

2 Computation model

We consider a distributed system consisting of a finite set of nodes named $\Pi = \{p_1, \dots, p_{|\Pi|}\}$, where $|\Pi| > 1$. The set of participants is known by all nodes. There is one process per node. Hence, the words node and process are interchangeable. Every pair of nodes is assumed to be connected by means of a reliable communication channel and processes communicate by sending and receiving messages.

To simplify the presentation, we take the range \mathcal{T} of the clock's tick to be the set of natural numbers. Processes do not have access to \mathcal{T} : it is introduced for the convenience of the presentation.

The number of units of the resource is k . We assume that k is known to every process. The duration of the CS is bounded.

Nodes can fail by crashing only, and this crash is permanent. A *correct* process is a process that does not crash during a run, otherwise, it is *faulty*. Let f , which is known to every process, denote the maximum number of processes that may crash in the system. We consider that $1 < f < k$.

The underlying system must satisfy a property, that we denoted *Responsiveness Property* (\mathcal{RP}), on top of which our fault tolerant k-mutual exclusion algorithm runs. Such a property characterizes the synchrony of the underlying system. In addition, a REQUEST-REPLY mechanism, as proposed in [MMR03], is necessary: any process p_i that broadcasts a request must wait for the corresponding REPLY messages from $|\Pi| - f$ nodes for then to detect failures.

Let $t \in \mathcal{T}$. We use the following notation:

- $crashed^t$: the set of processes that have crashed at or before t .
- $not_rec_from_i^t$: the set of processes from which p_i has not received a REPLY message to its last request that terminated at or before t .
- rec_i^t : the set of processes p_j that, at time t , have received a REPLY message from p_i to their last request terminated at time t . Thus, $rec_i^t = \{p_j | p_i \notin not_rec_from_j^t\}$.

Notice that we assume that p_i is always included in rec_i^t and is never included in $not_rec_from_i^t$.

The \mathcal{RP} , defined by Mostefaoui et al. in [MMR03], is the following:

Responsiveness Property: (\mathcal{RP})

$$\mathcal{RP} \stackrel{def}{=} \forall p_i : \forall t : (p_i \notin crashed^t) \Rightarrow \\ (|\bigcap_{0 \leq u \leq t} (rec_i^u \cup crashed^u)| > f)$$

Intuitively, the \mathcal{RP} property states that for each process p_i , from the beginning of the algorithm execution and until p_i possibly crashes, there is a set of processes whose size is greater than f such that each process p_j of this set received a REPLY message from p_i to each of its request until p_j possibly crashed.

Notice that the number of faults f also depends on the underlying system. It can not be greater than the number of crashes which violates the \mathcal{RP} of the system since the latter must always hold, despite failures.

3 Examples of systems that satisfy the \mathcal{RP}

An example of a system that satisfies the \mathcal{RP} would be a set Π of processes fully-connected and organized in a logical ring where p_i can communicate to all the nodes but the replies received by p_{i+1} and p_{i-1} (p_i 's neighbors) to their respective requests always includes p_i 's reply until p_i possibly crashes. In order to ensure that the \mathcal{RP} always holds, the system should tolerate at most $f = 2$ faults, which implies that k must be greater than 2. Such a system is feasible if, for instance, both channels $(p_i - p_{i-1})$ and $(p_i - p_{i+1})$ are never the slowest ones among all the channels connecting p_i to another process. Hence, if p_i , p_{i+1} and p_{i-1} are not crashed at time t then $\{p_{i-1}, p_i, p_{i+1}\}$ belong to rec_i^t , i.e.,

$p_i \notin \text{not_rec_from}_{p_{i-1}}^t$, $p_i \notin \text{not_rec_from}_{p_i}$, and $p_i \notin \text{not_rec_from}_{p_{i+1}}^t$. Thus, since a node p_j waits for $|\Pi| - 2$ REPLY messages for its request, if p_i is not crashed at t , among the REPLY messages received by p_j there exists at least one from those three processes. This message can then include the information that p_i has not crashed.

A second example would be a system composed of interconnections of clusters, such as a Grid, where communication latencies between nodes of different clusters are much higher than communication latencies between nodes within the same cluster and where there exists always at least one correct process in every cluster [SP08]. The number of faults must thus be bounded by the number of nodes of the smallest cluster minus one. If we consider that the $|\Pi|$ nodes of a Grid are spread over c clusters ($1 \leq c \leq |\Pi|$) and that the number of nodes of each cluster C_i is equal to nc_i ($nc_i > 1$), then $f < \min(nc_i)$. Such a value ensures that p_i will always receive at least one REPLY message from every other cluster. Furthermore, due to the difference of latencies, replies sent by p_i as an answer to the processes' requests of its own cluster at time t are always received by these processes among their first replies. These processes then are always included in rec_i^t , i.e., the set of processes that received a reply message from p_i at time t . To this same reason, if p_j and p_i does not belong to the same cluster message, a message received from p_j from a process that belong to p_i 's cluster will contain information about p_i aliveness.

4 Timer-free fault-tolerant k-mutual exclusion algorithm

In this section we present our permission-based k-mutual exclusion algorithm that tolerates $1 < f < k$ failures when the underlying system satisfies the *Responsiveness* property (\mathcal{RP}).

4.1 Description of the Algorithm

Algorithm 1 shows the pseudo-code of our fault tolerant k -mutual exclusion algorithm. We consider that each process infinitely calls the functions *Request_resource()* to ask access to a unit of the shared resource, i.e., to execute the critical section (CS), and calls the *Release_resource()* when it releases the CS. Lamport's logical clocks [Lam78] is used for controlling causality of events.

```

1:  $state_i \leftarrow not\_requesting$  ▷ Initialization
2:  $H_i \leftarrow 0$ 
3:  $crashed_i \leftarrow \emptyset$ 
4:  $not\_rec\_from_i \leftarrow \emptyset$ 
5:  $X_i \leftarrow \emptyset$ 
6:  $pending_i \leftarrow \emptyset$ 

   Request_resource(): ▷ Node wishes to enter CS
7:  $state_i \leftarrow requesting$ 
8:  $new\_not\_rec\_from_i \leftarrow \Pi \setminus \{i\}$ 
9:  $X_i \leftarrow \{i, not\_rec\_from_i\}$ 
10:  $last_i \leftarrow H_i + 1$ 
11:  $have\_perm_i[] \leftarrow false$ 
12:  $perm\_count_i \leftarrow 0$ 
13: for all  $j \neq i : j \notin crashed_i$  do
14:   send  $REQUEST(i, last_i, crashed_i)$  to  $j$ 
15: wait until  $(perm\_count_i \geq |\Pi - crashed_i| - k)$ 
16:  $state_i \leftarrow CS$ 

   Release_resource(): ▷ Node exits the CS
17: for all  $(j \neq i : j \in (pending_i \setminus crashed_i))$  do
18:   send  $REPLY(i, PERM, not\_rec\_from_i)$  to  $j$ 
19:  $pending_i \leftarrow \emptyset$ 
20:  $state_i \leftarrow not\_requesting$ 

21: upon receive  $REQUEST(j, H_j, crashed_j)$  do
22:    $H_i \leftarrow \max(H_i, H_j) + 1$ 
23:   for all  $k \in crashed_j \setminus crashed_i$  do
24:     if  $have\_perm_i[k]$  then
25:        $perm\_count_i --$ 
26:        $have\_perm_i[k] = false$ 
27:    $crashed_i \leftarrow crashed_i \cup crashed_j$ 
28:   if  $(state_i = CS)$  or  $(state_i = requesting \text{ and } (last_i, i) < (H_j, j))$  then
29:     send  $REPLY(i, NOPERM, not\_rec\_from_i)$  to  $j$ 
30:      $pending_i \leftarrow pending_i \cup \{j\}$ 
31:   else
32:     send  $REPLY(i, PERM, not\_rec\_from_i)$  to  $j$ 

33: upon receive  $REPLY(j, ack, not\_rec\_from_j)$  do
34:    $new\_not\_rec\_from_i \leftarrow new\_not\_rec\_from_i \setminus \{j\}$ 
35:    $Update(X_i, \langle j, not\_rec\_from_j \rangle)$ 
36:   if  $|new\_not\_rec\_from_i| \leq f$  then
37:      $not\_rec\_from_i \leftarrow new\_not\_rec\_from_i$ 
38:      $crashed_i \leftarrow crashed_i \cup (\bigcap_{\langle -, ls \rangle \in X_i} \langle -, ls \rangle)$ 
39:   if  $(state_i = requesting)$  and  $(ack = PERM)$  and  $(j \notin crashed_i)$  then
40:      $perm\_count_i ++$ 
41:      $have\_perm_i[j] = true$ 

```

Algorithm 1: Fault-tolerant k-mutual exclusion algorithm

Process p_i can issue two types of messages: (1) REQUEST message which is timestamped by the pair (H_i, i) , i.e., the current value of p_i 's logical clock and its identification. Such a timestamp defines a total order for the requests¹. The message also holds the information about the set of faulty process of which p_i is aware; (2) REPLY message which contain p_i 's identification, a tag which denotes if p_i gives its permission (PERM) or not (NOPERM) to the requesting process to execute the critical section, and the set of processes not_rec_from

¹ $(H_i, i) < (H_j, j) \Leftrightarrow H_i < H_j$ or $(H_i = H_j \text{ and } i = j)$

that did not answer to the last request of p_i . In order to uniquely identify the couple (REQUEST, set of REPLIES), each REPLY message also includes the timestamp of the corresponding REQUEST message. For the sake of simplicity such a timestamp is not included in the code of Algorithm 1.

When a process p_i wants to access an unit of the shared resource, it sends a REQUEST message to those processes it believes that are currently not crashed. Each of these processes, if correct, will eventually gives its permission to p_i . However, p_i does not need to wait for a permission from all of them. When it has received a sufficient number of permissions such as to be sure that no more than $(k-1)$ of the other correct processes are executing the critical section, p_i can start executing it too. If there is no crash, this number of permissions is equal to $(|\Pi| - k)$. On the other hand, in our approach, a process dynamically gathers information about node crashes. Hence, p_i just needs to wait for $|\Pi - crashed_i| - k$ permissions, where $|crashed_i|$ is the number of nodes that p_i currently knows to be faulty. Notice that thanks to \mathcal{RP} , there is no false suspicion, i.e, if p_i considers that p_j has crashed, then it really did.

Process p_i handles the following local variables:

- $state_i$: keeps one of the tree possible state of p_i with respect to the critical section: *requesting*, *CS*, *not_requesting*.
- H_i : Lamport's logical clock (counter).
- $last_i$: the value of the logical clock of p_i when it sent its last REQUEST message.
- $perm_count_i$: keeps the number of received permissions for the current request.
- $have_perm_i$: a boolean variable that informs if process p_k has already given its permission to p_i 's current request or not.
- $crashed_i$: the set of processes that p_i currently knows to have crashed.
- $not_rec_from_i$: denotes the set of at most f processes from which p_i has not received a REPLY message to its last request.
- $new_not_rec_from_i$: an auxiliary variable used to construct $not_rec_from_i$.
- X_i : the set of the not_rec_from sets received by p_i . Each element of X_i is a tuple composed of the identification of the process that replied to p_i and the respective not_rec_from set included in the replier's message ,i.e., $\langle j, not_rec_from_j \rangle$.
- $pending_i$: the set of processes to which p_i has postponed the sending of a REPLY message.

The *Initialization* procedure is executed once by each process at the beginning of the algorithm (lines 3-6).

When process p_i requests one unit of the shared resource (function *Request_resource()*), it sets its state to *requesting* and sends a REQUEST message to all processes except those it is aware of being crashed (lines 13-14). Upon receiving at least $|\Pi - crashed_i| - k$ replies, it knows that it can access a unit

of the resource and it then changes its state to *CS* (lines 15-16). It is worth remarking that while waiting for such permissions, the value of $|\Pi - crashed_i|$ dynamically decreases if p_i detects the failure of one or more processes. Note also that even if p_i does not actually send a request to itself, it considers that it does not belong to the set of processes that have not currently replied to the request (line 8). A last remark is that p_i includes in its *REQUEST* message the information it current knows about crashed processes ($crashed_i$) which allows the other processes, specially those that do not request the *CS* very often, to update their knowledge about node failures.

When exiting the *CS* by calling the function *Release_resource()*, p_i gives its permission to all those processes whose requests it has deferred and which it supposes not to be faulty (lines 17-20). It then sets its state to *not_requesting*.

Upon reception of a *REQUEST* message from p_j , node p_i updates its logical clock H_i . It also verifies if in p_j 's *REPLY* message there exists information about the crash of a node from p_k which p_i had already received a permission. In this case, p_i must not consider p_k 's permission (lines 24- 26). It then updates its $crashed_i$ set with the information about crashed nodes of which p_j is aware (line 27). Notice that p_i will detect the crash of p_k when (1) it receives a new request from a node that knows that p_k is crashed and p_i didn't receive a *REPLY* message from p_k to its current request or (2) when eventually p_i executes line 38.

Process p_i then sends back a permission (*REPLY* message tagged with *PERM*) only if it is not in the *CS* or if its current request has not priority over p_j 's one, i.e., the timestamp of p_j 's *REQUEST* message is smaller than the timestamp of p_i 's according to the total order defined by Lamport. (line 32). Otherwise, it sends a *REPLY* message to p_j tagged with *NOPERM* (line 29) and should remember that when it releases the *CS*, it must give its permission to p_j (line 30). In both cases, it includes its *not_rec_from* set in the *REPLY* message.

When p_i receives a *REPLY* message from p_j , it excludes p_j from its *new_not_rec_from* and updates its X_i set with the *not_rec_from* sent by p_j (lines 34-35). Remark that a single node may reply twice to the same request of p_i : upon deferring the request (*NOPERM*) and then when releasing the critical section (*PERM*). Thus, for a given request, *Update*($X_i, \langle j, not_rec_from_j \rangle$) either includes $\langle j, not_rec_from_j \rangle$ in X_i if the latter does not have *not_rec_from_j* or replaces the previous one, otherwise. As at most f processes can crash and the channels are reliable, process p_i receives at least $(|\Pi| - f)$ *REPLY* messages. Furthermore, the \mathcal{RP} property ensures that since the beginning of the algorithm execution, there is a set of at least $f + 1$ processes such that every process p_j of this set always received a *REPLY* message from p_k to its query until p_j possibly crashes. This implies that, since p_i waits for at least $(|\Pi| - f)$ *REPLY* messages, the correct process p_k will not belong to at least one *not_rec_from* set, included in each of these messages. In other words, there is at least one process p_j that replies to p_i 's request which is either equal to p_k or in its turn has received a reply from p_k to its last request and thus p_k is not suspected of being crashed by p_i . After receiving at least $(|\Pi| - f)$ *REPLY* messages (line 36) and based on the *not_rec_from* sets, p_i can update its information about faulty processes, i.e., those processes that belong to all *not_rec_from* sets of received by p_i at time t (line 38).

Finally, in lines 39-40, if p_i is waiting to execute the critical section, the received REPLY message received from p_j contains a permission (PERM), and p_i has not detected the crash of p_j , the variable $perm_count_i$ is incremented.

Notice that p_i might take some time to detect a failure of a process since the detection depends on the REQUEST-REPLY messages but there is no false suspicions. However, since it continuously requests the critical section and receives other processes requests eventually such a failure will be detected by p_i . Lacking the exact view of the number of correct processes of the system for a period of time does not have an impact on the correctness of the algorithm but temporarily on its efficiency: p_i will just need to wait for more REPLY messages than is really necessary as long as it does not detect the failure.

4.2 Sketch of Proof

We start by proving that every crash is eventually detected by every correct process and that no correct process is suspected. We then prove the *safety* and *liveness* properties of our algorithm.

Theorem 1. *Every crashed process is eventually included in the crashed set of all correct processes.*

Proof. Let us consider that a process p_l crashes. At time t all the messages sent by p_l will eventually be received by the respective correct receivers, i.e., after t , no process will receive a REPLY message from p_l to its query.

As processes execute their *Request_resource()* procedure periodically and due to Lemma 3 (liveness), there is a time t' when p_l is included in *not_rec_from* sets (line 37) of all correct processes. Hence, all subsequent REPLY messages received by the correct process p_i from p_j after time $t'' > t'$ will include p_l in the *not_rec_from_j* set of these messages (line 33). Upon execution of line 38, process p_l will then be included in the *crashed_i* set of process p_i . Thus, all correct processes will eventually have p_l in its *crashed* set. \square

Lemma 1. *No correct process is included in any crashed set.*

Proof. Let p_l be a correct process. Suppose that the correct process p_l is included in the *crashed* set of processes. If this happened it is because at least one process, p_i , has executed line 38 and has added p_l to its *crashed_i*. Moreover, in this case, p_i has received at least $(|II| - f)$ replies messages to its last request and p_l is included in the *not_rec_from* of each of these REPLY messages. However, the \mathcal{RP} ensures that since the beginning of the algorithm execution, there is a set of at least $f + 1$ processes such that every process p_j of this set always received a REPLY message from p_l to its query until p_j possibly crashes. Thus, among the replies received by p_i there is always at least one which does not include p_l in its respective *not_rec_from* set. Consequently, line 38 could not have been being executed and p_l will never be included in a *crashed* set of a process. \square

Lemma 2 (Safety). *No more than k processes execute the CS at the same time.*

Proof. Let us suppose that more than k processes can be in the CS at the same time. Assume that at time t , $m > k$ nodes are executing the CS. Let the

pairs $(H_i, i) = (\text{logical clock, node identification})$, included in the REQUEST messages, be the sequence used by the m nodes to gain access to the CS. These pairs define a total order. Hence, the nodes in critical section are labeled with $p_1, \dots, p_k, p_{k+1}, \dots, p_m$ such that $(H_{p_1}, p_1) < \dots < (H_{p_k}, p_k) < (H_{p_{k+1}}, p_{k+1}) < \dots < (H_{p_m}, p_m)$. Consider that p_{k+1} is blocked at line 15 and only one permission is missing to p_{k+1} to enter the CS, i.e., $\text{perm_count}_{p_{k+1}} = |\Pi - \text{crashed}_{p_{k+1}}| - k - 1$. Thus, if it executed line 16, either p_{k+1} received the missing permission or it added a process to its $\text{crashed}_{p_{k+1}}$ set.

Let's examine the first case. In order to enter the CS, p_{k+1} has received $(|\Pi - \text{crashed}_{p_{k+1}}| - k)$ REPLY messages, i.e., at most $k - 1$ nodes did not send a REPLY message to p_{k+1} . Thus, among the k nodes p_1, \dots, p_k in CS one of them, $p_{X(\leq k)}$, sent a reply to p_{k+1} . Consider the reception of the REQUEST message timestamped with the $(H_{p_{k+1}}, p_{k+1})$ by p_X . Four subcases are possible:

- *Subcase 1.* p_X is in the state *not_requesting*. Upon receiving the REQUEST message, H_{p_X} becomes $\geq H_{p_{k+1}}$ (line 22). Consequently, $(H_{p_X}, p_X) > (H_{p_{k+1}}, p_{k+1})$. Hence, p_X could not be in the CS at time t with $(H_{p_X}, p_X) < (H_{p_{k+1}}, p_{k+1})$.
- *Subcase 2.* p_X is in the CS state. In this case, p_X would not send a REPLY message with its permission to p_{k+1} .
- *Subcase 3.* p_X is in the state *requesting*. Two cases are possible:
 - $(H_{p_X}, p_X) < (H_{p_{k+1}}, p_{k+1})$. This is similar to *Subcase 1*.
 - $(H_{p_X}, p_X) > (H_{p_{k+1}}, p_{k+1})$. This is similar to *Subcase 2*.
- *Subcase 4.* p_x crashes. Obviously it can not reply to p_{k+1} .

Thus, it is impossible for any node $p_{X(\leq k)}$ to reply to the request of node p_{k+1} .

Now let us consider the case where node p_{k+1} entered the CS thanks to the addition of a node p_y to its $\text{crashed}_{p_{k+1}}$ set. p_y cannot be one of the k nodes in the CS at time t since this will mean that this node has really crashed (lemma 1) which contradicts the hypothesis that p_1, \dots, p_k are in CS at time t . Neither can node p_y be one of the $|\Pi - \text{crashed}_{p_{k+1}}| - k - 1$ nodes that has already given its permission to p_{k+1} since line 24 prevents p_{k+1} to consider p_y as crashed if it has received a permission from it in the current request. Furthermore, if p_y is added to $\text{crashed}_{p_{k+1}}$, it cannot be one of the $|\text{crashed}_{p_{k+1}}|$ nodes already in this set. Thus, the only remaining case would be $p_y = p_{k+1}$ which is not possible since p_{k+1} is not crashed at time t by hypothesis. \square

Lemma 3 (Liveness). *If a correct process requests to execute the CS, then at some time later the process executes it.*

Proof. In Algorithm 1, there is only one *wait* clause (line 15) that can block the execution. In order to go on executing, a correct process p_i must gather $(|\Pi - \text{crashed}_i| - k)$ permissions. The pair (logical clock, node's identification) used to identify all requests define a total order for the requests. Thus, among all the requests not yet satisfied there is only one request that has the highest priority over the other ones.

At any moment, the number of correct processes is at least $(|\Pi| - f)$. Consider the process p_l whose request has the highest (H_l, p_l) . All correct processes

which are either in the *requesting* or in the *not requesting* state will send a REPLY message to p_l giving its permission (line 32). If there are less than k processes executing the CS, p_l has gathered sufficient permissions since $f < k$ and the number of permission received from those processes is greater than $|\Pi - \text{crashed}_{p_l}| - k$. Thus, p_l will execute line 16. On the other hand, if there are k processes in the CS, p_l will need to wait for a permission from one of these processes. If p_l detects the crash of one of the k processes and since there is no false suspicion (lemma 1), it can enter the CS. Otherwise, since $f < k$, at least one of them will eventually exit the CS and p_l will then execute line 16.

Once p_l exits the CS, the process's request was satisfied and will not be considered anymore. Since requests are totally ordered by the pair (logical clock, node's identification), each of them will eventually have the highest priority, obtaining then right to execute the critical section. \square

Theorem 2. *The algorithm 1 solves the fault tolerant k -mutual exclusion and tolerates $f < k$ failures when the underlying system satisfies the Responsiveness property (RP).*

5 Performance Evaluation

This section describes a set of performance evaluation results aimed at comparing the both our fault tolerant extension of Raymonde's algorithm and the original algorithm.

As explained before, in Raymond's algorithm when a node wants to access one of the k units of the shared resource, it broadcasts a message to the other $|\Pi| - 1$ nodes of the system. It can only access it after having gathered $|\Pi| - k$ REPLY messages. Contrarily to our algorithm, several permissions deferred by p_i to p_j can be combined into a single REPLY message which is then sent to p_j when p_i releases its current critical section. Even if Raymond's algorithm does not explicitly consider failure of nodes, the fact that it does not need to wait for a permission from all the participants implicitly renders it fault tolerant to some extent: a crashed node can be considered as a node that did not give its permission. It tolerates up to $k - 1$ faults, i.e., if $f = k - 1$ nodes were crashed, a node asking to execute a CS would still get it. However, each crash reduces the efficiency of the algorithm since the number of processes that can concurrently execute the CS decreases by one.

5.1 Environment and Parameters

The experiments were conducted on a dedicated machine with a 2.66Hz CPU and 2GB of RAM, running Linux. The algorithms were implemented in Python 2.6, a dynamic object-oriented programming language that supports multi-threads. We simulated a Grid configuration composed of 10 clusters of 10 nodes where latencies between nodes of different clusters are always higher than between nodes of the same cluster. The number of units of the shared resource was fixed to 10, i.e., $k = 10$ and $f \leq 9$.

An application is characterized by:

- α : time taken by a node to execute the critical section;

- β : mean time interval between the release of the CS by a node and a new request by this same node.
- ρ : the ratio β/α , which expresses the frequency with which a unit of the shared resource is requested.

Notice that the greater the value of ρ is, the smaller is the frequency of request.

In the evaluation of the algorithms, the following metrics were considered:

- **obtaining time**: the time between the instant a node requests a unit of the shared resource and the instant it gets it;
- **CS bandwidth**: the average number of critical section execution per unit of time;
- **efficiency**: the number of shared resource's units that are simultaneously in use.
- **waiting queue**: the average number of pending requests, i.e., the average number of processes that are in the requesting state during the whole execution of the algorithm.
- **failure detection time** : the average time between the instant at which the crash takes place and the instant at which all nodes detect it.

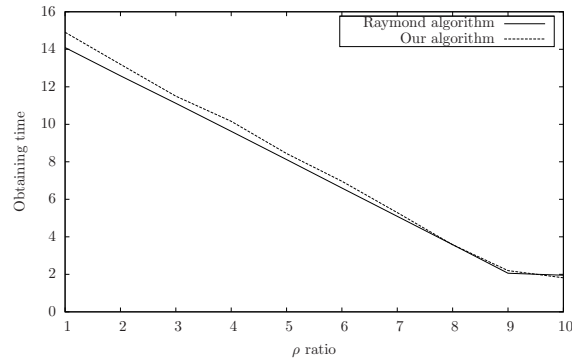
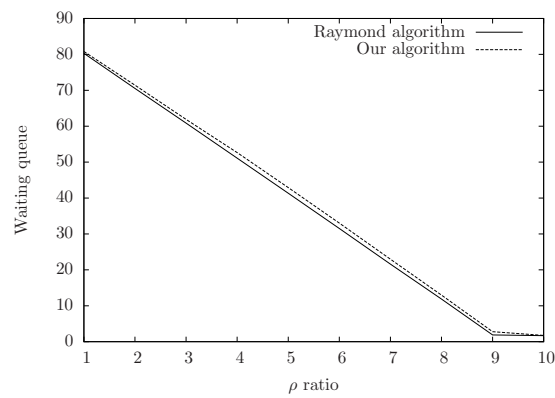
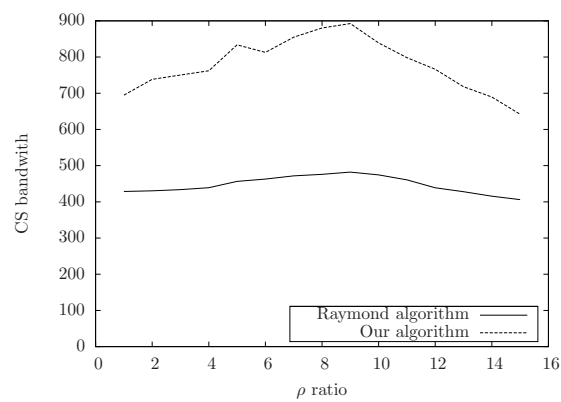
Each experiment was executed 20 times and each thread request X times one unit of the shared resource. The results presented represent the average value. For an experiment, all nodes have the same value of ρ while the value of α was fixed to 2s to all experiments. However, for avoiding that all processes issue their requests at the same time, we applied a Gaussian request distribution.

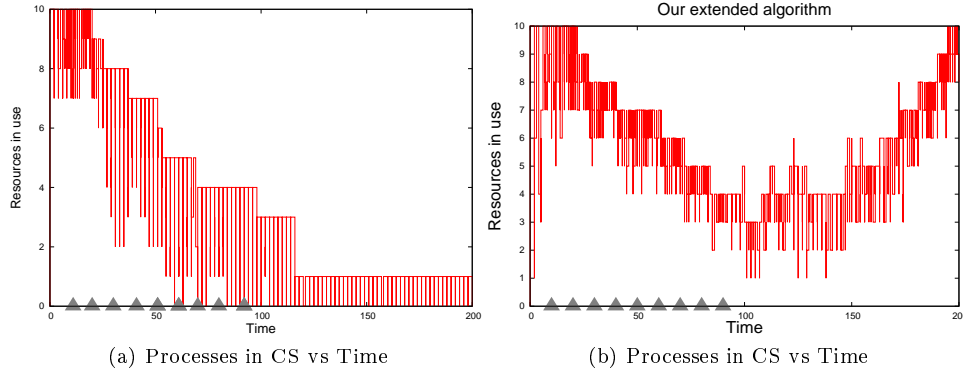
5.2 Performance Results without Failure

In order to validate the behavior of our simulator and measure the overhead that our fault tolerant extension introduces in the original Raymond's algorithm, the *obtaining time* and the *CS bandwidth* were measured, as shown in Figures 5.2 and 5.2 respectively when ρ increases. Nodes do not crash. In order to better understand the behavior of both curves, we have also measured, Figure 5.2, the size of the *waiting queue*, i.e., number of pending requests when ρ varies.

We can observe in Figure 5.2 that the *obtaining time* decreases when ρ increases for both algorithms. Such a behavior can be explained since when ρ increases, the frequency of request decreases and thus the size of the *waiting queue* which is confirmed by Figure 5.2. We can remark that for ρ greater than 9, the *waiting queue* is always almost empty and consequently the average *obtaining time* depends only on message latencies between cluster, i.e., the delay for sending the request to remote nodes and to receive at least on reply for each other cluster (2 seconds in Figure 5.2).

For ρ smaller than 9, the number of critical section execution per second (*CS bandwidth*) keeps constant as we can see in Figure 5.2 despite the fact that the frequency of requests decreases when ρ increases. This can be explained because in this case the *waiting queue* is never empty. Hence, when a unit of the shared resource is released, there exists always a pending request waiting

Figure 1: Obtaining time vs ρ Figure 2: Average size of the waiting queue vs ρ Figure 3: Number of CS per second vs ρ



for it. However, for ρ greater than 9, the CS starts decreasing since there exists no pending requests and the frequency of requests decreases as well.

Another important remark of the results presented in the three figures is that our fault tolerant extension does behaves like the original algorithm and does not introduce a significant overhead in the *obtaining time*, nor does in the *CS bandwidth*, even if our algorithm sends in average more message per critical section execution than Raymond's one. In our experiments such a value is X for Raymond's algorithm and Y for our algorithm which corresponds to an increase of $Z\%$ in the number of messages per critical section.

5.3 Performance Results with Failures

In order to evaluate the efficiency of both Raymond's algorithms and our algorithm, we have measured the number of resource's units that can be simultaneously in use for $\rho = X$ and $\rho = Y$. The triangles represent a crash (up to 9 in our experiments).

We can clearly observe in Figure ?? that independently of ρ , in Raymond's algorithm the maximum number of concurrent accesses decrements by one after each crash. Some time after 9 crashes take place, only one process can be in the critical section at a given time, i.e., the efficiency of the algorithm drops from 9 to 1. Notice that after a crash the number of processes in CS does not drop immediately since some processes were already in the critical section before the crash. However, in Figure ?? for both ρ s, after the 9 crashes the number of shared resource units in use starts going up and at 200s this number is reestablished to 9. The gradual decreasing and increasing behavior of the curves is due to the failure detection time, as shown in Figure 4: a process takes some time to detect a failure which increases with the number of crashes. Furthermore, since crash information is also spread when a node broadcasts a request (line 14 of Algorithm 1), the lower ρ is, the faster the other nodes will update their knowledge about crashes.

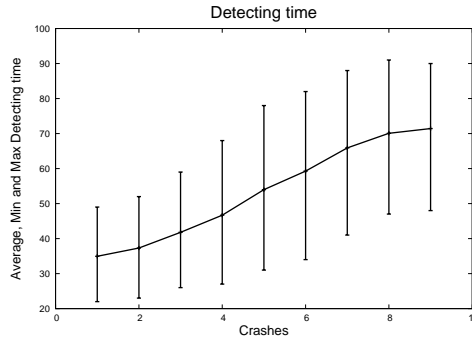


Figure 4: Average, minimum and maximum failure detection time for a number of crashes

6 Related Work

Several authors have proposed fault-tolerant extensions both to token-based [NLM90],[MS94],[CSL90] and permission-based 1-mutual exclusion algorithms [AA91],[CSR99]. The latter usually use a quorum approach.

Like Raymond's [Ray89] algorithm, Srimani and Reddy [SR92] k -mutual exclusion algorithm inherently support failures. It is based on Suzuki and Kasami's algorithm [SK85] and controls k tokens. If a node holds one of the k tokens, it can enter the critical section. Even the algorithm do not explicitly considers failure of nodes, the fact that it keeps k tokens implicitly renders it fault tolerant to some extent. However, each crash reduces the effectiveness of the algorithm since the number of processes that can concurrently execute the CS decreases by one.

In [BAS08], we have proposed an extension to Raymond's algorithm in order to both tolerate up to $N - 1$ node crashes and avoid that the algorithm degrades when failure occurs, i.e., to ensure that it is always possible to have k processes in the CS simultaneously, despite failures. To this end, we have made use of the information provided by unreliable failure detectors of class \mathcal{T} [DGF GK05] since it is the weakest one to solve the fault-tolerant 1-mutual exclusion problem. However, the drawback of our solution is the overhead in terms of number of messages that the failure detector \mathcal{T} incurs.

The majority of fault-tolerant permission-based k -mutual exclusion found in the literature use quorums [HJK93],[JHK97],[CC97],[KFYA93],[NM94]. Some of these algorithms exploit the k -coterie approach [JHK97],[NM94],[KFYA93]. Informally, a k -coterie is a set of node quorums, such that any $(k+1)$ quorums contain a pair of quorums intersecting each other. A process can enter a critical section whenever it receives permission from every process in a quorum. The availability of a coterie is defined as the probability that a quorum can be successfully formed and it is closely related to the degree of fault tolerance that the algorithm supports. On the other hand, Chang et. al propose in [CC97] an extended binary tree quorum for k -mutual exclusion which imposes a logical structure to the network and tolerates in the best case up to $(n - k * (\log_2(2n/k)))$ node failures. Although quorum-based algorithms are resilient to node failures and/or network partitioning, the drawback of such approach is the complexity of constructing the quorums themselves.

Reddy et al. present in [RMG08] a k -mutual exclusion algorithm for Chord P2P system where a dynamic logical tree control global requests by distributing them to the k units of the resources. There are then k distributed queues where each one gather pending requests to the corresponding unit. Without given much details, the authors argue that successors nodes in the logical ring of Chord can act as a replica for the node. Furthermore, in order to ensure data consistency amongst all nodes, atomic broadcast to all successors is used whenever a node changes its state.

Two other k -mutual exclusion algorithms, [WCM01] and [MJ06] provide fault tolerance but for wireless ad-hoc networks. The authors in [WCM01] propose a token-base algorithm which induces a logical direct acyclic graph on the network which dynamically adapts to the changing topology of ad-hoc networks. Mellier et al. address in [MJ06] the problem of at most k exclusive accesses to a communication channel by nodes that compete to broadcast on it, i.e., at most k mobile nodes can simultaneously broadcast on it. Message collision problems are solved by the protocol. However, neither of the algorithms tolerate node failures, but just link failures.

7 Conclusion

Based on Raymond's algorithm, this paper has presented a f -fault tolerant k -mutual exclusion algorithm where $1 \leq f < k$, provided that the underlying system satisfies the *Responsiveness Property*. We have proposed a new approach for detecting and tolerating failures which is integrated in the k -mutex algorithm itself and thus renders the solution not expensive.

Furthermore, even if the performance can temporarily degrade just after a crash, the efficiency of the algorithm is dynamically restored as soon as the remaining processes detect the failure, which does not happen with the original Raymond's algorithm. Performance experiments on a simulated Grid platform that satisfies the \mathcal{RP} have shown the efficiency and benefits of our approach in comparison to the former.

References

- [AA91] D. Agrawal and A. El Abbadi. An efficient and fault-tolerant solution for distributed mutual exclusion. *ACM Trans. Comput. Syst.*, 9(1):1–20, 1991. 16
- [BAS08] M. Bouillaguet, L. Arantes, and P. Sens. Fault tolerant k -mutual exclusion algorithm using failure detector. In *7th International Symposium on Parallel and Distributed Computing*, pages 343–350, 2008. 16
- [BV95] Shailaja Bulgannawar and Nitin H. Vaidya. A distributed k -mutual exclusion algorithm. In *Int. Conference on Distributed Computing Systems*, pages 153–160, 1995. 3
- [CC97] Y. Chang and B. Chen. An extended binary tree quorum strategy for k -mutual exclusion in distributed systems. In *Proc. of the 1997*

- Pacific Rim International Symposium on Fault-Tolerant Systems*, page 110, 1997. 16
- [CSL90] I. Chang, M. Singhal, and M. Liu. A fault tolerant algorithm for distributed mutual exclusion. In *Proc. of the IEEE 9th Symp. on Reliable Distrib. Systems*, pages 146–154, 1990. 3, 16
- [CSR99] G. Cao, M. Singhal, and N. Rische. A delay-optimal quorum-based mutual exclusion scheme with fault-tolerance capability. In *The 8th ACM symposium on Principles of Distributed Computing*, page 271, 1999. 16
- [CT96] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of ACM*, 43(2):225–267, March 1996. 4
- [DGFGK05] C. Delporte-Gallet, H. Fauconnier, R. Guerraoui, and P. Kouznetsov. Mutual exclusion in asynchronous systems with failure detectors. *J. Parallel Distrib. Comput.*, 65(4):492–505, 2005. 4, 16
- [HJK93] S.T. Huang, J.R. Jiang, and Y.C. Kuo. k-coterie for fault-tolerant k entries to a critical section. *Distributed Computing Systems, 1993., Proceedings the 13th International Conference on*, pages 74–81, 1993. 3, 16
- [JHK97] J. Jiang, S. Huang, and Y. Kuo. Cohorts structures for fault-tolerant k entries to a critical section. *IEEE Transactions on Computers*, 46(2):222–228, 1997. 16
- [KFYA93] H. Kakugawa, S. Fujita, M. Yamashita, and T. Ae. Availability of k-coterie. *IEEE Trans. Comput.*, 42(5):553–558, 1993. 16
- [Lam78] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978. 6
- [MBB⁺92] K. Makki, P. Banta, K. Been, N. Pissinou, and EK Park. A token based distributed k mutual exclusion algorithm. *Proc. of the 4th IEEE Symposium on Parallel and Distributed Processing*, pages 408–411, 1992. 3
- [MJ06] R. Mellier and Myoupo J. Fault tolerant mutual and k-mutual exclusion algorithms for single-hop mobile ad hoc networks. *Int. Journal Ad Hoc and Ubiquitous Computing*, 1(3):156–167, 2006. 17
- [MMR03] A. Mostefaoui, E. Mourgaya, and M. Raynal. Asynchronous implementation of failure detectors. In *Proc. of Int. Conf. on Dependable Systems and Networks*, June 2003. 4, 5
- [MS94] D. Manivannan and M. Singhal. An efficient fault-tolerant mutual exclusion algorithm for distributed systems. In *Int'. Conf. on Parallel and Distributed Computing Systems*, pages 525–530, 1994. 3, 16

- [NLM90] S. Nishio, K. F. Li, and E. G. Manning. A resilient mutual exclusion algorithm for computer networks. *IEEE Trans. on Parallel and Distributed Systems*, 1(3):344–355, july 1990. 16
- [NM94] M. L. Neilsen and M. Mizuno. Nondominated k-coterie for multiple mutual exclusion. *Inf. Process. Lett.*, 50(5):247–252, 1994. 3, 16
- [PMP⁺96] N. Pissinou, K. Makki, E. K. Park, Z. Hu, and W. Wong. An efficient distributed mutual exclusion algorithm. In *ICPP, Vol. 1*, pages 196–203, 1996. 3
- [Ray89] K. Raymond. A distributed algorithm for multiple entries to a critical section. *Inf. Process. Lett.*, 30(4):189–193, 1989. 1, 3, 16
- [RMG08] V. A. Reddy, P. Mittal, and I. Gupta. Fair k mutual exclusion algorithm for peer to peer systems. In *Proceedings of the 2008 The 28th International Conference on Distributed Computing Systems*, pages 655–662, 2008. 16
- [SK85] I. Suzuki and T. Kasami. A distributed mutual exclusion algorithm. *ACM Trans. Comput. Syst.*, 3(4):344–349, 1985. 16
- [SP08] N. Schiper and F. Pedone. On the inherent cost of atomic broadcast and multicast in wide area networks. In *ICDCN*, pages 147–157, 2008. 6
- [SR92] P. K. Srimani and R. L. N. Reddy. Another distributed algorithm for multiple entries to a critical section. *Inf. Process. Lett.*, 41(1):51–57, 1992. 3, 16
- [WCM01] J. Walter, G. Cao, and M. Mitrabhanu. A k-mutual exclusion algorithm for wireless ad hoc networks. In *ACM POMC'01*, 2001. 17



Centre de recherche INRIA Paris – Rocquencourt
Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)

Centre de recherche INRIA Bordeaux – Sud Ouest : Domaine Universitaire - 351, cours de la Libération - 33405 Talence Cedex
Centre de recherche INRIA Grenoble – Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier
Centre de recherche INRIA Lille – Nord Europe : Parc Scientifique de la Haute Borne - 40, avenue Halley - 59650 Villeneuve d'Ascq
Centre de recherche INRIA Nancy – Grand Est : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex
Centre de recherche INRIA Rennes – Bretagne Atlantique : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex
Centre de recherche INRIA Saclay – Île-de-France : Parc Orsay Université - ZAC des Vignes : 4, rue Jacques Monod - 91893 Orsay Cedex
Centre de recherche INRIA Sophia Antipolis – Méditerranée : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
<http://www.inria.fr>
ISSN 0249-6399