A Timer-free Fault Tolerant K-Mutual Exclusion Algorithm

Mathieu Bouillaguet, Luciana Arantes, and Pierre Sens
Université Pierre et Marie Curie-Paris 6, LIP6/Regal,
UMR 7606, 4 place Jussieu, 75252 Paris cedex 05, France;
INRIA Paris - Rocquencourt,
Domaine de Voluceau, BP 105, 78153 Le Chesnay Cedex, France
{mathieu.bouillaguet,luciana.arantes,pierre.sens}@lip6.fr

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 [21], 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.*

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*). Hence, only one process can be in the critical section at any given time. The k-mutual exclusion problem (*k-mutex*) is a generalization of the mutual exclusion problem 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-mutex 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*).

Several k-mutex algorithms have been proposed in the literature and they can be classified into two main categories: permission-based [21], [20], [9] and token-based [23], [13], [4]. The first one is based on the principle that a node gets into critical section only after having received permissions 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 k tokens gives a node the right to enter into the CS. 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 inherently tolerate failures or can be adapted to tolerate them more easily.

We present in this paper a fault tolerant permission-based k-mutex algorithm. The choice for a permission-based is justified by the reason mentioned above. Our algorithm is inspired by Raymond's algorithm [21], where a process 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 integrated in the algorithm and its messages. Unlike the majority of fault tolerant mutual exclusion algorithms [15], [18], [7], our algorithm does not require extra messages for broadcasting information about crashes, neither does it require timers nor failure detectors for checking the liveness of nodes. Information about node failures is included in the messages of the algorithm themselves. Furthermore, contrarily to some k-mutual exclusion algorithms [21],[23] where the efficiency of the algorithm drops at every failure because the number of processes that can simultaneously execute the *CS* decreases as well, our fault tolerance approach guarantees that even if the algorithm might temporarily degrade, its efficiency is reestablished (i.e., k processes in the *CS* simultaneously), despite failures.

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 $k-1$ node crashes. However, detection of failures is only possible if the underlying system satisfies a property, denoted the *Responsiveness Property* ($\mathcal{RP}$), which is based on Mostefaoui et al.'s work [17]. In other words, our approach relies on an additional assumption which characterizes the synchronism 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 this set has always got a reply from p_k to its request until p_j possibly crashes. The $\mathcal{RP}$ then guarantees that if p_i waits for $|\Pi| - f$ responses to its broadcast request (where Π is the number of initial nodes of the system and f maximum number of crashes with $1 \leq f < k$) among the received replies, there will be at least one from the $f+1$ processes of the above set. Such a reply message will contain information about p_k liveness, if p_k has not crashed. Consequently, p_k will not be suspected to be faulty by p_i. Otherwise, the information that p_k does not answer 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 fault tolerant k-mutex algorithm ensures (1) that every crash is eventually detected by every correct process and (2) no correct process is suspected. It is worth pointing out that the conjunction of (1) and (2) is respectively equivalent to the *strong completeness* and *perpetual strong accuracy* assumptions of the perfect failure detector $\mathcal{P}$ [6], which is sufficient to solve fault-tolerant mutual exclusion problem [8].

One could argue that the fault tolerance provided by our k-mutual exclusion algorithm does not work for every 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 algorithm itself. Examples of such systems will be discussed in the article.

The paper is organized as follows. Section 2 defines the computation model. Examples of systems 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 while 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 $\Pi = \{p_1, \ldots, p_{|\Pi|}\}$, 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 $1 \leq f < k$.

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 synchronism of the underlying system. In addition, a REQUEST-REPLY mechanism, as proposed in [17], is necessary: a process p_i that broadcasts a request must wait for the corresponding REPLY messages from $|\Pi| - f$ nodes.

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 $not\_rec\_from_i^t$.

Based on Mostefaoui et al. in [17], the **Responsiveness Property**, $\mathcal{RP}$, is defined as follows:

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

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.

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 among the replies received by p_{i+1} and p_{i-1} (p_i's neighbors) to their respective requests there is always p_i's reply until p_i possibly crashes. In order to ensures that the $\mathcal{RP}$ always holds, the system should tolerate at most two faults, which implies that $f \leq min(2, k-1)$. 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. Since a node p_j waits for $|\Pi| - f$ 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. The message informs that p_i has not crashed ($p_i \notin not\_rec\_from_{replier}$).

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 is at least one correct process in every cluster. The number of faults must thus be bounded by the number of nodes of the smallest cluster minus one and k: if the $|\Pi|$ nodes of a Grid are spread over c clusters and the number of nodes in the smallest one is equal to nc_i, then $f \leq min(nc_i - 1, k - 1)$. 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, responses 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 ones. Therefore, if p_j and p_i do not belong to the same cluster, a REPLY message received by p_j from a process that belong to p_i's cluster will contain information about p_i liveness.

4. Timer-free fault-tolerant k-mutex

In this section we present our permission-based k-mutual exclusion algorithm that tolerates $1 \leq f < k$ failures when the underlying system satisfies the ($\mathcal{RP}$). Its pseudo-code is shown in Algorithm 1.

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 [12] is used for controlling causality of events.

Process p_i can issue two types of messages: (1) REQUEST message which is timestamped by the pair (C_i, i), i.e., the current value of p_i's logical clock and its identification. Such a timestamp defines Lamport's total order for the requests: $(C_i, i) < (C_j, j) \Leftrightarrow C_i < C_j\ or\ (C_i = C_j\ and\ i < j)$. The message also holds information about the set of faulty processes p_i is aware of; (2) REPLY message which contains 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 that did not answer to the last request of p_i. In order to uniquely identify the pair (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 pseudo-code of Algorithm 1.

The following local variables are handled by p_i:

- $state_i$: keeps one of the possible p_i's states with respect to the critical section: *requesting*, *CS*, *not_requesting*.
- C_i: Lamport's logical clock (counter).
- $last_i$: the value of the logical clock of p_i when it sent its last REQUEST message.
- $have\_perm_i$: a boolean vector 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.
- $not\_rec\_from_i$: the set of 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 $\langle j, not\_rec\_from_j \rangle$.
- $pending_i$: the set of processes to which p_i sent a REPLY message with a NOPERM tag.

When a process p_i wants to access a unit of the shared resource (*Request_resource()*), it sets its state to *requesting* and sends a REQUEST message to all processes except those which p_i knows to be faulty (lines 12-13). Notice that there is no false suspicion, i.e., if p_i considers that p_j has crashed, then it really did. Each of those processes, if they are not faulty, will reply to p_i. However, p_i does not need to wait for a permission from all of them to enter the CS. When it has received a sufficient number of permissions

1: $state_i \leftarrow not\_requesting$ ▷ Initialization
2: $C_i \leftarrow 0$
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: $X_i \leftarrow \{\langle i, \mathrm{not\_rec\_from_i}\rangle\}$
10: $last_i \leftarrow C_i + 1$
11: $have\_perm_i[\,] \leftarrow false$
12: **for all** $j \neq i : j \notin crashed_i$ **do**
13: send $REQUEST(i, last_i, crashed_i)$ to j
14: **wait until** $(Count\_perm(have\_perm_i) \geq |\Pi - crashed_i| - k)$
15: $state_i \leftarrow CS$

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

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

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

Algorithm 1: Fault-tolerant k-mutex algorithm

such as to be sure that no more than $(k-1)$ of the other correct processes are executing the CS, p_i can start executing it too. More explicitly, p_i just needs to wait for $|\Pi - crashed_i| - k$ permissions. The call to $Count\_perm(have\_perm_i)$ returns the current number of permissions received by p_i (line 14). Thus, upon receiving the necessary number of permissions, p_i knows that it can access a unit of the resource and it then changes its state to *CS* (line 15). Note that while waiting for the permissions, the value of $|\Pi - crashed_i|$ dynamically decreases if p_i detects a new failure of one or more processes. A second remark is that p_i includes in its REQUEST message the information it currently knows about crashed processes which allows the other processes, specially those that do not request the CS very often, to update their knowledge about failures.

Process p_i exits the CS by calling the function *Release_resource()*: it sends a permission to all processes to whom p_i sent a reply with a NOPERM tag in it and which it supposes not to be faulty (lines 16-19). It then sets its state to *not_requesting*.

Upon reception of a REQUEST message from p_j, node p_i updates its logical clock C_i. It also verifies if in p_j's REPLY message there exists information about the crash of a node p_k which sent its permission earlier. In this case, p_i must not consider p_k's permission (lines 23- 24). It then updates its $crashed_i$ set with the information about crashed nodes carried in p_j's message (line 25) and sends back a permission (REPLY message tagged with PERM) only if it is not in the CS or if its current request does not have priority over p_j's one according to the total order defined by Lamport (line 30). Otherwise, it sends a REPLY message to p_j tagged with NOPERM (line 27) and should remember that when it releases the CS, it must give its permission to p_j (line 28). In both cases, it includes its $not\_rec\_from_i$ 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 32-33). 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, when process p_i receives at least $(|\Pi| - f)$ REPLY messages, it can update its information about faulty processes, i.e., those processes that belong to all not_rec_from sets received by p_i at time t (line 36).

Finally, in lines 37-38, if p_i is waiting to execute the CS, the received REPLY message contains a permission (PERM), and p_i has not detected the crash of p_j, p_i considers the reply of p_j as a permission.

4.1. Sketch of Proof

Due to the lack of space, we are going to present just the arguments and the outline of the proof of Algorithm 1. The complete proof can be be found in [3].

Theorem 1: Eventually, every process that crashes is permanently suspected of failure by every correct process (*strong completeness* property) and no process is suspected before it crashes (*perpetual strong accuracy* property) when the underlying system satisfies the *Responsiveness* property ($\mathcal{RP}$)..

Proof: Let consider the correct process p_i and the the faulty one p_f.

Strong completeness: We must prove that p_f is eventually and permanently included in $crashed_i$. Just after its crash, p_f will stop sending REPLY messages. Since processes execute their *Request_resource()* procedure periodically, there exists then a time t when p_f is included in not_rec_from sets (line 35) of all processes that has not crashed till t. Thus, after t, p_i will detect the failure of p_f either when gathering the REPLY messages to its request issued after t (line 36) or upon the reception of a request from p_j (line 13) which detected the crash of p_f by executing line 36 before p_i. Once p_f is included in a $crashed_i$, it is never removed from it.

Perpetual strong accuracy: We must prove that p_i is never included in the $crashed_j$ set of p_j. This follows directly from the $\mathcal{RP}$ and the REQUEST-REPLY mechanism: among the $|\Pi| - f$ first REPLY messages received by p_j there is always at least one process that does not include p_i in its respective not_rec_from set. Consequently, when line 36 is executed by p_j, p_i will never be included in $crashed_i$. □

Theorem 2: Algorithm 1 solves the fault tolerant k-mutual exclusion and tolerates $f < k$ failures when the underlying system satisfies the *Responsiveness* property ($\mathcal{RP}$).

Proof: We must prove that Algorithm 1 ensures the *safety* and the *liveness* properties.

Safety: At most k nodes are in the critical section at a given time. The proof is by contradiction. Let assume that at time t there exists $k + 1$ processes $p_1, p_2, \ldots, p_{k+1}$ in the critical section. Such processes are not crashed at time t, i.e., they are not in $crashed_{k+1}$ at time t (*perpetual strong accuracy* property, Theorem 1).

To enter the critical section, node p_{k+1} received at most $|\Pi - crashed_{k+1}| - (k + 1)$ permissions from processes other than $p_1, p_2, \ldots, p_{k+1}$. If a permission is received from a node included in $crashed_{k+1}$ due to the detection of its crash (*strong completeness* property, Theorem 1), such a permission is not considered (line 24) by p_{k+1}. Hence, amongst the k nodes $p_1, p_2, \ldots, p_k$, at least one of them sent a REPLY message to p_{k+1}.

Let (C_i, i) be the (logical clock, identity) pair included in the REQUEST message of p_i, and $(C_{p_1}, p_1) < \ldots < (C_{p_k}, p_k) < (C_{p_{k+1}}, p_{k+1})$ be the total ordered sequence used by the $k + 1$ nodes $p_1, p_2, \ldots, p_{k+1}$ to gain access to the critical section. Let $p_x (x \leq k)$ be a process that replied to the message *REQUEST(p_{k+1},$C_{p_{k+1}}$,$crashed_{k+1}$)*. Upon reception of it, if p_x was either in CS or in the *requesting* state with (C_x, p_x), it would not reply to p_{k+1}; if p_x was in $not\_requesting$ state, or either in *requesting* or CS state but with (L, p_x) such that $(L, p_x) \leq (C_x, p_x)$, the logical clock of p_x would become $\geq C_{p_{k+1}}$ (line 21). Consequently, p_x could not be in the CS at time t with $(C_{p_x}, p_x) < (C_{p_{k+1}, p_{k+1}})$. Hence, it is impossible that a node p_x, $x \leq k$ replied to the request of p_{k+1}. Process p_{k+1} thus could not have gathered the $(|\Pi - crashed_i| - k)$ permissions necessary to enter the critical section.

Liveness: If a correct process p_i requests the critical, then eventually it gets it, i.e., if p_i is in the *requesting* state then at some time later it executes line 15 ($state_i \leftarrow CS$). In Algorithm 1, there is only one $wait$ clause (line 14) that can block the execution of a *requesting* process p_i. To enter the CS, p_i must gather $(|\Pi - crashed_i| - k)$ permissions.

The sequence of pairs (logical clock, identity) of the pending REQUEST messages defines a total order. Let p_l be the correct process in the $requesting$ state which has the highest priority over all the other processes that are in this same state. Let also assume that all processes that are not crashed have already received the REQUEST message from p_l and that there are x ($x \leq k$) processes in the CS when p_l is blocked at the $wait$ clause. Hence, all processes which neither crashed nor are in CS will send a REPLY message to p_l (line 30). Let nb_perm be the number of permissions received by p_l from these processes. Since there are no false failure suspicions (Theorem 1), if $nb\_perm \geq (|\Pi - crashed_l| - k)$, p_l can enter the CS; otherwise, it must wait for the right number of permissions before entering the CS. As $f < k$, the number of missing permissions, $miss\_perm$ is at most equal to x (in the worst case, $k - 1$ crashes took place and they have not been detected by p_l yet, i.e., p_l has currently received $|\Pi| - x - k$ permissions; since p_l needs $|\Pi - crashed_l| - k$ permissions, $crashed_l = \emptyset$ and $miss\_perm = x$). However, $miss\_perm$ will decrease either at each reception of a REPLY message sent by one of the x processes upon releasing its CS (line 17) or at each new detection of a node failure by p_l (lines 36 and 25). Since there were x processes in CS, and failures are eventually detected (Theorem 1), eventually $miss\_perm = 0$. Process p_l then enter the critical section. □

5. Performance Evaluation

This section describes a set of performance evaluation results which compare Raymond's algorithm to our fault tolerant one. Even if the former 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 resilient to failures 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, but the number of processes that can simultaneously execute the *CS* decreases by one at each crash.

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 platform composed of 10 clusters of 10 nodes where latencies between nodes of different clusters are 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 $f = 9$. Crashes are inserted after 100 sec. Each experiment was executed 10 times.

The *degree of parallelism* of an application is characterized by $\rho = \beta/\alpha$, where α is the time taken by a node to execute the critical section and β is the time interval between the release of the CS by a node and a new request by this same node. Notice that the higher the value of ρ, the higher the degree of parallelism of the application, i.e., an application whose ρ is high does not ask for a unit of the shared resource very frequently.

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

- *CS bandwidth*: the average number of critical section completed per unit of time;
- *efficiency*: the number of processes currently executing a critical section;
- *waiting queue*: the average number of processes waiting for a unit of the shared resource;
- *obtaining time*: the time between the instant a node requests a unit of the shared resource and the instant it gets it;

The presented results are average value. For a given experiment, all nodes have the same value of ρ. To study different values of ρ we fixed α while β varied.

Performance in absence of failures: In order to validate the behavior of our simulator and evaluate the overhead that our fault tolerant extension introduces into the original Raymond's algorithm, we measured the *CS bandwidth* and the size of the *waiting queue* when there are no crashes and ρ varies from 1 to 16, as shown in Figure 1.

We can observe in Figure 1(a) that for both algorithms, the *CS bandwidth* remains constant (around 5) for ρ smaller than 9. This happens because, even if the *waiting queue* decreases when ρ increases (Figure 1(b)), whenever a unit of the shared resource is released, there always exists a requesting process waiting for it. However, for ρ greater than 9, the *CS bandwidth* starts going down since the *waiting queue* is almost empty and processes requests are less frequent. Another important remark is that our fault tolerant extension behaves like the original algorithm and does not add a significant overhead in the *CS bandwidth*, nor in the size of the *waiting queue*, even if our algorithm sends on average between 3 and 42% more messages per CS than Raymond's one. Considering such figures, it is interesting to notice that in our solution a process which is in critical section or in the waiting queue sends two REPLY messages to requesting processes: one with $NOPERM$ tag since it has more priority and one with $PERM$ tag when it releases the CS. Thus, the number of extra REPLY messages of our algorithm is around 42% when $\rho = 1$ but it decreases with ρ since the *waiting queue* decreases as well (Figure 1(c)). On the other hand, when the queue is empty, at most $k-1$ processes, which defer to give their permission to a requesting process will send two REPLY messages to this process. Therefore, the overhead of extra message drops (around 3%) as shown in the same figure.

CS bandwidth and waiting queue: Figure 2 shows the same experiments than Figure 1 but when 9 crashes are inserted. In the case of Raymond's algorithm, the *CS bandwidth* logically drops for any ρ compared to executions without crashes due to the loss of efficiency after each crash. This metric does not fall when ρ increases because the lower frequency of requests is balanced by the increased size of the *waiting queue* as seen in figure 2(b). For our algorithm the drop in *CS bandwidth* is less important thanks to the failure detection mechanism which permits to recover the full efficiency of the algorithm. The small raise observed between $\rho = 1$ and $\rho = 9$ is the consequence of the rapid decline of the *waiting queue* on the same interval. From $\rho = 9$ onwards, the *CS bandwidth* decreases rapidly since the average size of the *waiting queue* is stable and does not compensate anymore the lower frequency of requests.

Efficiency: Aiming at evaluating the *efficiency* of both algorithms, we have measured in Figure 3 the number of resource's units that are simultaneously in use (left

(a) Number of CS per second vs ρ (b) Average size of the waiting queue vs ρ (c) Message overhead

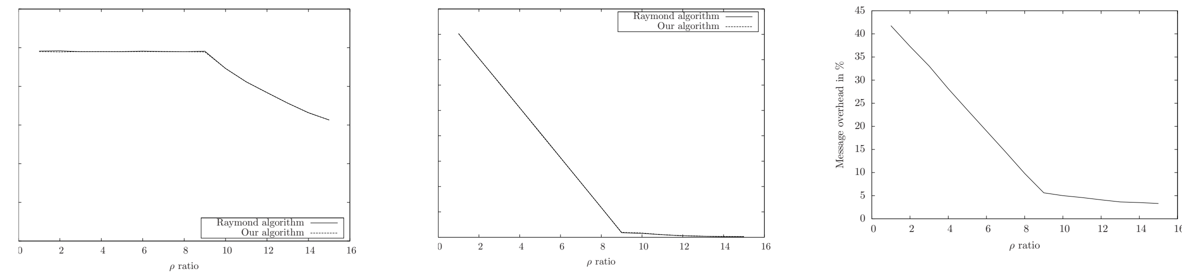

Figure 1: CS bandwidth, waiting queue and message overhead vs ρ in absence failure

(a) Number of CS per second vs ρ (b) Average size of the waiting queue vs ρ

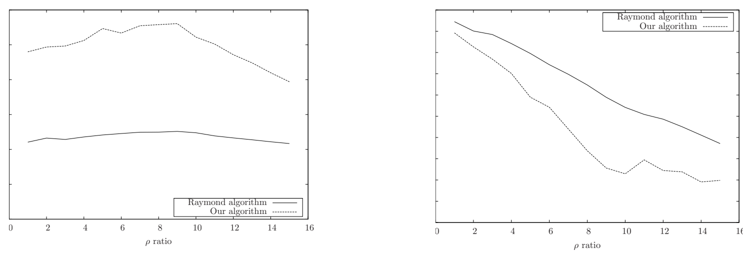

Figure 2: CS bandwidth, and waiting queue vs ρ in presence of failures

(a) Raymond algorithm (b) Our algorithm

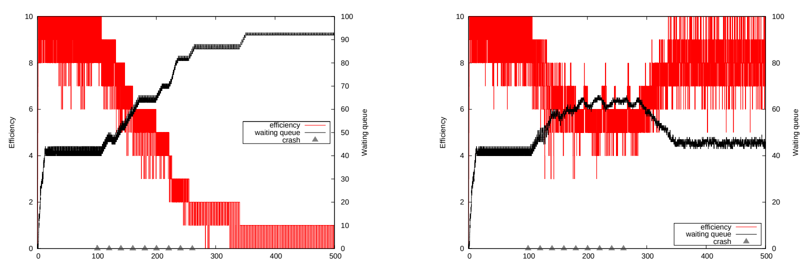

Figure 3: Efficiency and waiting queue vs time

side of both curves) for $\rho = 5$. We added in the same graph the *waiting queue* (right side of the curves). Each triangle represents a crash (up to 9 in our experiments).

We can note in Figure 3(a) that Raymond's efficiency degrades after each crash. Some time after all crashes take place, only one process can be in the critical section at a given time, i.e., the efficiency of the algorithm drops to 1 and never recovers. After a crash, the number of processes in CS does not decrease immediately since some processes were already in the critical section when the crash happened. Notice also that the *waiting queue* grows up with the drop of the algorithm's efficiency. When the efficiency is equal to 1, all correct processes except one wait for a unit of the shared resource. On the other hand, in our algorithm (Figure 3(b)), after the 9 crashes occurred the number of shared resource units simultaneously in use is reestablished to 9, i.e. 9 processes can be in the CS simultaneously again. The gradual decreasing and increasing behavior of the curves is due to the fact that processes take some time to detect failures: upon each crash and before it is detected, the efficiency drops by one and thus the number of processes in the waiting queue increases; otherwise, when the crashes start being detected, the efficiency starts increasing and consequently the number of processes in the *waiting queue* decreases.

Obtaining time: Figure 4 shows the *obtaining time* of a process for the two algorithms when $\rho = 5$. In absence of crashes, the *obtaining time* of a process is around $8s$ for both of them. When crashes take place, their respective *obtaining time* increases since their efficiency declines as just described in Figure 3, i.e., a processes must wait longer in order to get a unit of the shared resource because less processes can execute the CS simultaneously. Nevertheless, in the case of our algorithm, the *obtaining time* decreases when failures start being detected by processes and it gets back to its initial value when the efficiency of the algorithm is fully re-established at time 350s. Contrarily to our algorithm, after the 9 crashes, the *obtaining time* of processes in Raymond's algorithm starts growing and stabilize around $180s$. Such a high value will never decrease and can be explained since the efficiency of the algorithm dropped down to 1: before having the right to execute the critical section a process will need to wait for the CS execution of all processes currently in their critical section and all processes in the waiting queue, which accounts for a little less than 90 processes in this case and the duration of critical section is $2s$.

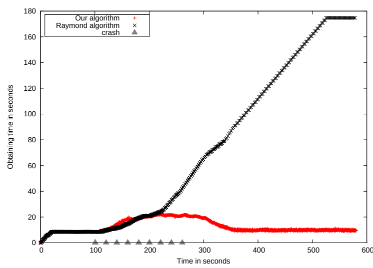

Figure 4: Comparison of obtaining time for $\rho = 5$

6. Related Work

Several authors have proposed fault-tolerant extensions both to token-based [19],[15],[7] and permission-based 1-mutual exclusion algorithms [1],[5]. The latter usually use a quorum approach.

Like Raymond's [21] algorithm, Srimani and Reddy [23] k-mutex algorithm inherently support failures. It is based on Suzuki and Kasami's algorithm [24] and controls k tokens. Even if the algorithm does not explicitly consider failure of nodes, the fact that it keeps k tokens implicitly renders it fault tolerant to some extent. However, each crash reduces the efficiency of the algorithm.

In [2], we propose an extension to Raymond's algorithm in order to both tolerate up to $N-1$ node crashes and avoid the degradation of the algorithm efficiency in the presence of failure. To this end, we have made use of the information provided by unreliable failure detectors of class $\mathcal{T}$ [8] 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 [9],[10],[11],[14]. They exploit the k-coteries approach. 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 its quorum. The availability of a coterie it is closely related to the degree of reliability that the algorithm supports. Although these algorithms are resilient to node failures, the drawback of such approach is the complexity of constructing the coteries themselves.

Reddy et al. present in [22] a k-mutex 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 for gathering pending requests to the corresponding unit. Without giving much details, the authors argue that fault tolerance can be provided if successors nodes in the logical ring of Chord can act as a replica for the node.

Two other k-mutex algorithms, [25] and [16], offer fault tolerance but for wireless ad-hoc networks. The authors in [25] propose a token-base algorithm which dynamically adapts to the changing topology of ad-hoc networks. Mellier et al. address in [16] the problem of at most k exclusive accesses to a channel by nodes that compete to broadcast on it. 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

[1] D. Agrawal and A. E. Abbadi, "An efficient and fault-tolerant solution for distributed mutual exclusion,"

ACM Trans. Comput. Syst., vol. 9, no. 1, pp. 1–20, 1991.

[2] M. Bouillaguet, L. Arantes, and P. Sens, “Fault tolerant k-mutual exclusion algorithm using failure detector,” in *ISPDC*, 2008, pp. 343–350.

[3] M. Bouillaguet, L. Arantes, and S. Sens, “Un algorithme de k-mutual exclusion tolérant aux fautes sans temporisateur (in french),” INRIA - France, Tech. Rep., 2009.

[4] S. Bulgannawar and N. H. Vaidya, “A distributed k-mutual exclusion algorithm,” in *ICDCS*, 1995, pp. 153–160.

[5] G. Cao, M. Singhal, and N. Rishe, “A delay-optimal quorum-based mutual exclusion scheme with fault-tolerance capability,” in *PODC*, 1999, p. 271.

[6] T. D. Chandra and S. Toueg, “Unreliable failure detectors for reliable distributed systems,” *J. ACM*, vol. 43, no. 2, pp. 225–267, March 1996.

[7] I. Chang, M. Singhal, and M. Liu, “A fault tolerant algorithm for distributed mutual exclusion,” in *SRDS*, 1990, pp. 146–154.

[8] C. Delporte-Gallet, H. Fauconnier, R. Guerraoui, and P. Kouznetsov, “Mutual exclusion in asynchronous systems with failure detectors,” *JPDC*, vol. 65, no. 4, pp. 492–505, 2005.

[9] S. Huang, J. Jiang, and Y. Kuo, “k-coteries for fault-tolerant k entries to a critical section,” in *ICDCS*, 1993, pp. 74–81.

[10] J. Jiang, S. Huang, and Y. Kuo, “Cohorts structures for fault-tolerant k entries to a critical section,” *IEEE Trans. on Computers*, vol. 46, no. 2, pp. 222–228, 1997.

[11] H. Kakugawa, S. Fujita, M. Yamashita, and T. Ae, “Availability of k-coterie,” *IEEE Trans. Comput.*, vol. 42, no. 5, pp. 553–558, 1993.

[12] L. Lamport, “Time, clocks, and the ordering of events in a distributed system,” *Commun. ACM*, vol. 21, no. 7, pp. 558–565, 1978.

[13] K. Makki, P. Banta, K. Been, N. Pissinou, and E. Park, “A token based distributed k mutual exclusion algorithm,” *Proc. of the 4th IEEE Symposium on Parallel and Distributed Processing*, pp. 408–411, 1992.

[14] Y. Manabe, R. Baldoni, M. Raynal, and S. Aoyagi, “k-arbiter: A safe and general scheme for h-out of-k mutual exclusion,” *Theor. Comput. Sci.*, vol. 193, no. 1-2, pp. 97–112, 1998.

[15] D. Manivannan and M. Singhal, “An efficient fault-tolerant mutual exclusion algorithm for distributed systems,” in *IPDPS*, 1994, pp. 525–530.

[16] R. Mellier and M. J., “Fault tolerant mutual and k-mutual exclusion algorithms for single-hop movile ad hoc networks,” *Int. Journal Ad Hoc and Ubiquituos Computing*, vol. 1, no. 3, pp. 156–167, 2006.

[17] A. Mostefaoui, E. Mourgaya, and M. Raynal, “Asynchronous implementation of failure detectors,” in *DSN*, 2003.

[18] M. L. Neilsen and M. Mizuno, “Nondominated k-coteries for multiple mutual exclusion,” *IPL*, vol. 50, no. 5, pp. 247–252, 1994.

[19] S. Nishio, K. F. Li, and E. G. Manning, “A resilient mutual exclusion algorithm for computer networks,” *IEEE TPDS*, vol. 1, no. 3, pp. 344–355, 1990.

[20] N. Pissinou, K. Makki, E. K. Park, Z. Hu, and W. Wong, “An efficient distributed mutual exclusion algorithm.” in *ICPP*, 1996, pp. 196–203.

[21] K. Raymond, “A distributed algorithm for multiple entries to a critical section,” *IPL.*, vol. 30, no. 4, pp. 189–193, 1989.

[22] V. A. Reddy, P. Mittal, and I. Gupta, “Fair k mutual exclusion algorithm for peer to peer systems,” in *ICDCS*, 2008, pp. 655–662.

[23] P. K. Srimani and R. L. N. Reddy, “Another distributed algorithm for multiple entries to a critical section,” *IPL*, vol. 41, no. 1, pp. 51–57, 1992.

[24] I. Suzuki and T. Kasami, “A distributed mutual exclusion algorithm,” *ACM TOCS*, vol. 3, no. 4, pp. 344–349, 1985.

[25] J. Walter, G. Cao, and M. Mitrabhanu, “A k-mutual exclusion algorithm for wireless ad hoc networks,” in *ACM POMC’01*, 2001.