Fault Tolerant K-Mutual Exclusion Algorithm Using Failure Detector

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

We present in this paper a fault tolerant permission-based k-mutual exclusion algorithm, which is an extension of Raymond's algorithm [18]. Tolerating up to $n-1$ failures, our algorithm keeps its effectiveness despite failures. It uses information provided by unreliable failure detectors to dynamically detect crashes of nodes. Performance evaluation experiments show the performance of our algorithm compared to Raymond's when faults are injected.

1 Introduction

The k-mutual exclusion problem is a fundamental distributed problem which guarantees the integrity of the k units of a shared resource by restricting the number of process that can simultaneous access them. It then involves N processes which communicate via message passing and ask for accessing one of the k units of the shared resource, i.e., to execute a critical section (*CS*). Hence, a distributed k-mutual exclusion algorithm must ensure that at most k processes are in the *CS* at a given time (*safety property*) and that every *CS* request is eventually satisfied (*liveness property*).

Distributed k-mutual exclusion algorithms can be classified into two families: *permission-based* [18], [17], [8], [9], [15] and *token-based* [20], [12], [2] [23]. In the algorithms of the first family, a node gets into the critical section only after having received permission from all or a subset of the other nodes of the system. In the second family, the possession of the single token or one of the tokens gives a node the right to enter into the *CS*. The latter usually presents an average lower message cost of messages, but is less fault tolerant than permission-based algorithms which, by using broadcast, are naturally more resilient to failures.

Raymond's k-mutual exclusion algorithm [18] is an extension of Ricart-Agrawala's [19] permission-based 1-mutual exclusion algorithm. When a node wants to enter the *CS*, it broadcasts a message to the other $(N-1)$ nodes of the system. The requesting node can enter the *CS* if no more than $k-1$ nodes are currently executing the *CS*. That is, only after gathering $N-k$ permissions from the other nodes.

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. It tolerates up to $k-1$ faults. In other words, if instead of executing the *CS*, $k-1$ nodes were crashed, a node asking to execute a *CS* would still get it. However, each crash reduces the effectiveness of the algorithm since the number of processes that can concurrently execute the *CS* decreases by one. Therefore, we propose in this paper to extend Raymond's algorithm in order to both tolerate up to $N-1$ node crashes, instead of just $k-1$, 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. Another reason that motivates the current work is the fact that fault tolerant permission-based k-mutual exclusion algorithms which don't use a quorum approach are quite rare in the literature.

In order to get information about crashes of nodes, our solution exploits the information about the *liveness* of processes provided by distributed unreliable failure detectors. An unreliable failure detector (FD) [4] is a well-known basic block which offers information about process failures. It can be informally considered as a per process oracle, which periodically informs the list of current processes suspected of being crashed. It is unreliable in the sense it can make mistakes. Our solu-

tion basically relies on the unreliable detector of class $\mathcal{T}$ [7] since it is the weakest one to solve the fault-tolerant 1-mutual exclusion problem. Thus, a per process failure detector $\mathcal{T}$ module periodically gives information to the corresponding process about the actual state of the system. Such an information allows each process to dynamically update its knowledge about the actual number of running processes and therefore our algorithm becomes more effective in presence of failures than Raymond's algorithm. Just at the initialization phase, our algorithm also needs an unreliable failure detector of class $\mathcal{S}$.

The paper is organized as follows. Section 2 describes our model. The concept of unreliable FD and the properties of the FDs of class $\mathcal{T}$ are presented in section 3. Section 4 briefly describes Raymond's algorithm. Section 5 presents our fault-tolerant algorithm while section 6 outlines its proof. Some related work is given in section 7. Simulation performance results are shown in section 8. Finally, section 9 concludes the paper.

2 System model

We consider a distributed system consisting of a finite set of $N > 1$ nodes. The set of participants as well as N are known by all nodes. There is one process per node and processes communicate by message passing. No assumptions on the relative speed of processes neither on message transfer delays are made. Thus the system is asynchronous. Communication channels are reliable, but messages might be delivered out of order.

A process can fail by crashing and crashes are permanent. A *correct* process is a process that does not crash during a run; otherwise, it is *faulty*. The maximum number of processes that may crash in the system is equal to f ($f < N$).

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.

As we consider one process per node, the words node and process are interchangeable.

3 Unreliable Failure Detectors

Chandra and Toueg [4] have introduced the concept of unreliable failure detectors which are distributed oracles that provide information about *liveness* of system's nodes. Each process has access to a local failure detector module which outputs the list of processes that it currently suspects of having crashed. A local failure detector is *unreliable* since it can make mistake by erroneously adding to its list a process which is actually correct or not suspecting a crashed node. However, if later the FD realizes its mistake it corrects it by either removing or adding the node to its list.

Failure detectors (FDs) are formally characterized by two abstract properties: (1) the *completeness* property which characterizes the FD capability of suspecting every faulty node permanently and the (2) *accuracy* property which characterizes the FD capability of not suspecting correct nodes.

A *class* of FD is a set of FDs that have both the same *completeness* and *accuracy* properties. The strongest one is the class of perfect FD $\mathcal{P}$. It is characterized by the *strong completeness* property which defines that eventually every faulty process is permanently suspected by every correct process and the *strong accuracy* property where no process is suspected before it crashes. The class of FD $\mathcal{S}$ keeps the *strong completeness* property but relaxes the accuracy property to *weak accuracy* which states that *some* correct process is never suspected.

Delporte-Gallet and al. [7] have introduced the FD of class $\mathcal{T}$, also called the *trusting* FD. The authors proved that this class of FD is the weakest one to solve the fault-tolerant 1-mutual exclusion problem, i.e., the FD of class $\mathcal{T}$ is sufficient and necessary to solve such a problem. This class of FD has the *strong completeness* property and satisfies the following *accuracy* properties:

Eventual strong accuracy: There is a time after which correct processes are not suspected by any correct process.

Trusting accuracy: Every process j that is suspected by a process i after being trusted once by i (i.e., j was never suspected by i before) is crashed.

FDs of class $\mathcal{T}$ are strictly weaker than FDs of class $\mathcal{P}$. Roughly speaking, $\mathcal{T}$ can temporarily suspect a correct process, as long as it had never been removed from its list of suspects.

Figure 1 depicts a possible scenario of failure detection using the FD $\mathcal{T}$. For sake of simplicity, the messages exchanged between the four nodes are not shown in the figure. $H(i,t)$ denotes the set of processes that i suspects (does not trust) at time t.

Initially, the FD at process 1 outputs $\{2,3,4\}$, i.e., process 1 does not trust anyone, but itself. Thus, process 1 falsely suspects the other processes since no process has actually crashed. At time $t_2 > t_1$, process 2 and 3 get trusted by 1 ($H(1,t_2) = 4$). However, after that, process 3 crashes and process 1 suspects it again at t_3, after having trusted it at t_2. Therefore, considering the *trusting accuracy* property of $\mathcal{T}$, process 1 can

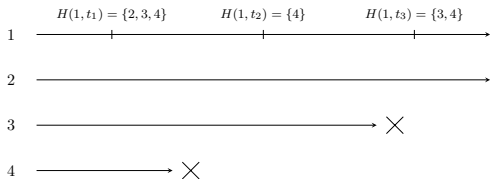


Figure 1: Example of execution with FD $\mathcal{T}$

be sure of process 3's crash. However, even if processes 4 has also crashed, process 1 can not be sure of its crash since it never trusts it.

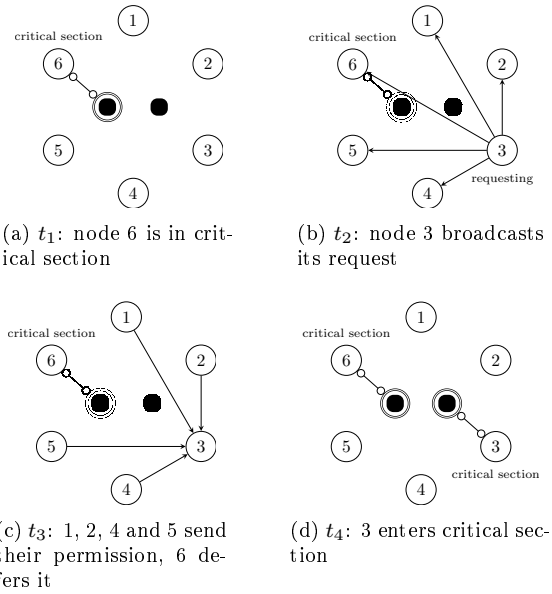
4 Raymond's algorithm

In Raymond's algorithm [18], when a node i wants to execute the critical section, it broadcasts a $REQUEST$ message to the other $(N-1)$ nodes. Each request is timestamped with Lamport's logical clock (sequence number) + the identity of the node [11]. Upon receiving this message, if node j is not requesting a unit of the resource, it immediately gives its permission to i by sending it back a $REPLY$ message. If j is in the critical section, it defers the sending of the $REPLY$ message till it ends the critical section. Finally, if j is also requesting a unit of the shared resource, the sequence numbers of both requests are compared. If they are equal, the identity of the nodes breaks tie. If i's request takes priority, j sends it back a REPLY message, otherwise j defers it till it releases the CS. When i has gathered $(N-k)$ $REPLY$ messages it enters its CS since it is sure that no more than $(k-1)$ of the other nodes are currently executing the critical section, which ensures the *safety* property. The timestamp of request messages guarantees the *liveness* property of the algorithm since it defines a total order for the requests.

As previously said, Raymond's algorithm implicitly tolerates $k-1$ crashes, i.e., the *safety* property still holds until up to $k-1$ failures occur. Thus, even if one or more nodes crash a second node is still able to access a unit of the resource if it can collect $(N-k)$ permissions. On the other hand, each time a failure occurs, the maximum number of processes that can concurrently execute the critical section decreases by one, reducing the effectiveness of the algorithm.

Figure 2 depicts a possible scenario of Raymond's algorithm. The system consists of 6 processes and 2 units of a shared resource. At t_1, node 6 has exclusive access to the first unit of the resource. Node 3 then requests a unit of the resource at t_2 by broadcasting a request to all the processes but itself. Since nodes 1, 2, 4, 5 are not requesting a resource, they immediately send back a $REPLY$ message to 3. On the other hand, node 6 defers its reply since it is in critical section. As soon as node 3 has received 4 $(N-k)$ $REPLY$ messages, it enters the critical section.

If a node other than node 3 had crashed at t_1, node 3 would be able to access a resource as it could still collect four permissions. However, the number of nodes that would concurrently execute the critical section would drop to one. In this particular case, node 3 would need to wait node 6 to release the critical section in order to get the four permissions, which degrades the effectiveness of the algorithm.



(a) t_1: node 6 is in critical section

(b) t_2: node 3 broadcasts its request

(c) t_3: 1, 2, 4 and 5 send their permission, 6 defers it

(d) t_4: 3 enters critical section

Figure 2: Example of Raymond's algorithm execution

5 The k-mutual exclusion problem in presence of failures using FDs

Raymond's algorithm has no information about node crashes. The number of participants of the algorithm is fixed to N despite node failures. On the other hand, in our algorithm every process i keeps in its local variable n_i the current number of correct processes. Whenever i is notified about the crash of a node, it decrements n_i. Hence, contrarily to Raymond's algorithm, the number of $REPLY$ messages needed by a requesting process is $(n_i - k)$, which decreases at every new crash of which i is aware.

In order to obtain information about node failures, our solution uses a FD of class $\mathcal{T}$. The *trusting accuracy* property of $\mathcal{T}$ guarantees that if a node j is suspected by i of having crashed after previously being trusted by i, j is actual crashed. Notice that the FD $\mathcal{T}$ is used by our algorithm all along its execution for

detecting crashes. However, just at the initialization phase, our algorithm also needs a FD of class $\mathcal{S}$ in order to guarantee that at the end of this phase, for each correct process i, there is at least one correct process that trusts i.

5.1 Description of the algorithm

Algorithm 1 is the complete pseudo-code of our fault tolerant k-mutual exclusion algorithm. We consider that each process infinitely executes the functions *Request_resource()* to ask access to one of the k units of the shared resource, i.e., to execute the critical section (*CS*), and *Release_resource()* to release it.

There are five types of messages: (1) *REQUEST* messages are broadcast by a process which executes the *Request_resource*() in order to inform the other processes that it wants to access one unit of the resource. They include the identity of the sender and the current value of the local logical clock. (2) *REPLY* messages are permission ones sent by processes in response to a *REQUEST* message. Multiple permissions can be aggregated into a single *REPLY* message which then carries an additional counter whose value equals to the number of deferred replies included in the message. (3) An *INIT* message is sent once by each process during the initialization phase. When a process receives such a message, it acknowledges its reception by returning an (4) *ACK* message. Finally, when a process detects a crash of a second one, it broadcasts a (5) *CRASH* messages in order to inform the other nodes of the process failure.

Process i keeps the following local variables:

- n_i : the number of correct processes of which i is current aware
- $state_i$: the current state of i (*requesting*, *not_requesting*, or *CS*)
- H_i: i's logical clock
- $last_i$: the value of H_i when i sent its last request
- $perm\_count_i$: the current number of permissions received by i to its last request
- $reply\_count_i[N]$: number of outstanding *REPLY* messages still to be received from each other node. It is necessary for preventing a *REPLY* message of an earlier request to be considered as a reply to the current request
- $defer\_count_i$[N]: number of replies that have been deferred by i to each other node
- $trusted_i$, $crashed_i$: sets which respectively keep the set of nodes that i once trusted and the set of crashed ones

Algorithm 1 Raymond's extended algorithm

```
1: n_i = N; state_i := not_requesting                    ▷ Initialization
2: H_i := 0; last_i := 0
3: perm_count_i := 0
4: reply_count_i[N] := 0
5: defer_count_i[N] := 0
6: trusted_i := ∅; crashed_i := ∅
7: send INIT(i) to all
8: wait until receive ACK from all j ∉ S_i

   Request_resource():                    ▷ Node wishes to enter CS
9: state_i := requesting
10: last_i := H_i + 1
11: perm_count := 0
12: for all j ≠ i : j ∉ crashed_i do
13:     send REQUEST(i, last_i) to j
14:     reply_count_i[j] + +
15: wait until (perm_count_i ≥ n_i − k)
16: state_i := CS

   Release_resource():                    ▷ Node exits the CS
17: state_i := not_requesting
18: for all (j ≠ i : defer_count_i[j] ≠ 0 and j ∉ crashed_i) do
19:     send REPLY(i, defer_count_i[j]) to j
20:     defer_count_i[j] := 0

21: upon receive REQUEST(j, H_j) do
22:     H_i := max(H_i, H_j)
23:     if (j ∉ crashed_i) then
24:         if (state_i = CS) or (state_i = requesting and
25:                                (last_i, i) < (last_j, j)) then
26:             defer_count_i[j] + +
27:         else
28:             send REPLY(i, 1) to j

29: upon receive REPLY(j, x) do
30:     if (j ∉ crashed_i) then
31:         reply_count_i[j] := reply_count_i[j] − x
32:         if (state_i = requesting) and (reply_count_i[j] = 0)
    then
33:             perm_count_i + +

34: upon receive INIT(j) do
35:     wait until j ∉ T_i
36:     trusted_i := trusted_i ∪ {j}
37:     send ACK(i) to j

38: upon receive CRASH(j) do
39:     if (j ∉ crashed_i) then
40:         crashed_i := crashed_i ∪ {j}
41:         if (state_i = requesting) and (reply_count_i[j] = 0)
    then
42:             perm_count_i − −
43:         n_i − −

44: upon (j ∈ trusted_i and j ∈ T_i) do
45:                          ▷ A crash of process j is detected
46:     trusted_i := trusted_i − {j}
47:     for all k ≠ i : k ∉ crashed_i do
48:         send CRASH(j) to k
```

Node i can always interrogate its local FD $\mathcal{T}$ and $\mathcal{S}$ (initialization phase) about node failures. They pro-

vide the list of suspected nodes in $\mathcal{T}_i$ and $\mathcal{S}_i$ sets respectively.

The *Initialization* phase (lines 1-8) is executed once by each process at the beginning of the algorithm. The "wait" condition of line 8 and the use of failure detector of class $\mathcal{S}$ ensure that at the end of this phase each correct process is included in at least one *trusted* set. By the *strong completeness* property of $\mathcal{S}$, eventually all processes not in $\mathcal{S}_i$ are correct. Thus, these processes eventually receive the $INIT$ message of i. Upon receiving it, they will execute lines 34 to 37. Notice that by the *weak accuracy* property of $\mathcal{S}$, there is at least one correct process that is never suspected which implies that the "wait" condition will not block, i.e., i will receive at least one ACK message from this process.

When node i requests a unit of the resource (lines 9-15), it broadcasts a $REQUEST$ message to all other processes it believes to be correct. It then increments $reply\_count[j]$ for each node $j \neq i$ and waits for $(n_i - k)$ $REPLY$ messages before entering the *CS*, $(perm\_count_i \geq n_i - k)$.

Upon reception of a $REQUEST$ message (lines 21-28), node j updates its logical clock and sends back a $REPLY$ message (line 28) only if it is not in the *CS* or if its current request has no priority over i's one. Otherwise, it defers the request (line 26). When i receives a $REPLY$ message from j it decrements $reply\_count_i[j]$ and if j has replied to all the previous requests sent by i (line 32), $perm\_count$ is incremented.

When i exits the *CS* by calling the $Release\_resource()$ (lines 17-20), it replies to all the deferred requests of those nodes that it believes to be correct.

If a node crashes, at least one process executes lines 44-48. It thus broadcasts a $CRASH$ message to all the other processes it supposes to be correct. When i receives a $CRASH$ message which informs that j is faulty, if i was not already aware of it, it decrements the number of current correct processes (line 43). In addition, if j had previously given its permission to i, such a permission is canceled, i.e., $perm\_count_i$ is decremented (line 42).

5.2 Example of execution

Figure 3 depicts a possible execution of our algorithm. The system consists of N=4 processes and k=2 units of a shared resource. The initialization phase is not shown. We consider that node 2 is in *CS* since t_0.

Node 1 requests a unit of the resource at t_1 by broadcasting a $REQUEST$ message to all the other processes. At t_2, node 3 sends a $REPLY$ message to 1 and node 4 crashes. When node 3 inquires its local FD

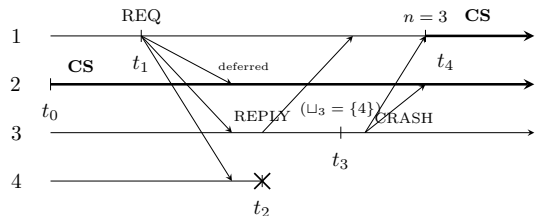


Figure 3: Example of an execution of our algorithm with $N = 4$ and $k = 2$

$\mathcal{T}$ module, the condition of line 44 ($4 \in trusted_3$ and $4 \in \mathcal{T}_3$) is verified. Hence, node 3 learns that node 4 is faulty and it then broadcasts a $CRASH$ message to 1 and 2. At time t_4, node 1 receives this message and thus executes lines 38-43 of the same algorithm where n_1 is decremented and the condition to enter critical section ($perm\_count_1 \geq 3 - 2$) is verified (line 15). Node 1 can then execute its critical section. It is worth remarking that if we considered the same scenario with Raymond's algorithm, node 1 would have to wait till node 2 exited its *CS* in order to receive the two $(N-k)$ permissions necessary to get into the *CS*.

6 Sketch of proof

We must prove that our algorithm satisfies the *safety* and *liveness* properties. Notice that in our approach, we consider that processes do not crash before the initialization phase. On the other hand, if they crash during the initialization, the *safety* property would still be ensured up to $k - 1$ failures.

6.1 Safety

Lemma 1. *No more than k different processes are in their critical section 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_c, $m > k$ nodes are executing the *CS*. Let the pairs (S,N)= (sequence number, node identity), 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 $N_1, \ldots, N_k, N_{k+1}, \ldots, N_m$ such that $(S_{N_1}, N_1) < \cdots < (S_{N_k}, N_k) < (S_{N_{k+1}}, N_{k+1}) < \cdots < (S_{N_m}, N_m)$. Consider the node N_{k+1}. In order to enter the *CS*, N_{k+1} has received $(n - k)$ $REPLY$ messages, i.e., at most $k - 1$ nodes did not send a $REPLY$ messages to N_{k+1}. Thus, among the k nodes $N_1, \ldots, N_k$ one of them $N_{X(\leq k)}$ sent a reply to N_{k+1}. Consider the reception of the $REQUEST(S_{N_{k+1}}, N_{k+1})$ by N_X. Four cases are possible:

- *Case 1.* N_X is in the state *not_requesting* or *requesting* with sequence number $(S_{N_X}, N_X) > (S_{N_{k+1}}, N_{k+1})$. Upon receiving the $REQUEST$ message, S_{N_X} became $\geq S_{N_{k+1}}$. Hence N_X could not be in the *CS* at time t_c with $(S_{N_X}, N_X) < (S_{N_{k+1}}, N_{k+1})$

- *Case 2.* N_X is in the state *CS* or *requesting* with sequence number $(S_{N_X}, N_X) < (S_{N_{k+1}}, N_{k+1})$. In this case, N_X would defer replying to N_{k+1}.

- *Case 3.* N_X is executing or attempting to execute the critical section in a previous request with sequence number R such that $(R, N_X) \leq (S_{N_X}, N_X) < (S_{N_{k+1}}, N_{k+1})$. Hence S_{N_X} would become $\geq S_{Nk+1}$ and so N_X could not be in the *CS* at time t_c with $(S_{N_k}, N_k) < (S_{N_{k+1}}, N_{k+1})$.

- *Case 4.* N_x crashes. Obviously it can not reply to N_{k+1}.

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

6.2 Liveness

Lemma 2. *If a correct process requests to execute the critical section, and it has the most priority request then at some time later the process executes it.*

Proof. Suppose that a correct process i is requesting a unit of the resource at some time t_c with $last_i = l_i$, i's request has priority over all the others, and i is never in its critical section after t_c, i.e., i never reaches line 16 of algorithm 1. Thus, i is blocked at a "wait" clause either at line 8 or at line 15.

The "wait" of line 8 can not block the process due to the *strong completeness* property of $\mathcal{S}$. Eventually all processes not in $\mathcal{S}_i$ are correct. Therefore, these processes eventually receive the $INIT$ message of i. Upon receiving such a message, they execute lines 34-37. Furthermore, by the *weak accuracy* property of $\mathcal{S}$, there is at least one correct process that is never suspected. Hence, i waits for the reply of at least one correct process. By the eventual *strong accuracy* property of $\mathcal{T}$, every correct process is eventually trusted by all correct processes. Hence, the "wait" clause of line 35 is not blocking, and the processes add i to their *trusted* set and send back an ACK message to i, unblocking the "wait" clause line 8.

Consider then that i is blocked at the "wait" clause of line 15 after having sent a $REQUEST(i, last_i)$ message to all processes such that $j \neq i$ and $j \notin crashed_i$. Four cases are possible for j:

(a) Process j is in the state *not_requesting*. The condition of line 25 is not satisfied and the process sends a permission (line 28).

(b) Process j is in the state *requesting*. Since i has priority over all the others requests, j sends back its permission.

(c) Process j is in its critical section. The duration of the critical section is bounded so it will eventually send back a reply message to i when executing the **Release_resource()** routine (lines 17-20).

(d) Process j crashes. By the *trusting accuracy* property of $\mathcal{T}$, some correct process m eventually and permanently will suspect it. In other words, the condition of line 44 is eventually satisfied at some process m for j ($j \in trusted_m$ and $j \in \mathcal{T}_m$). Thus, m will send a $CRASH$ message to all correct processes and every correct process will eventually receive it. Upon receiving the $CRASH(j)$ message, i decrements the number of participating nodes n_i, and, if it had already received a permission from j, it also decrements the number of received permissions. Thus, the condition of line 44 will inform the new state of the system, since n_i eventually represents the number of correct processes. Hence i will eventually receive exactly n_i replies, with n_i characterizing the number of correct processes. But i is blocked at line 15. It's a contradiction.

Thus as process i is never blocked at the "wait" of line 8 neither at the "wait" of line 15 of algorithm 1, it reaches line 16 and thus executes the critical section. □

Lemma 3. *If a correct process requests to execute the critical section, then at some time later the process executes it.*

Proof. By lemma 2, the process that has priority over the others will eventually executes its critical section. Once it exits the critical section, the process's request was satisfied and will not be considered anymore. Since requests are totally ordered, each of them will eventually have the highest priority, obtaining then right to execute the critical section. □

Theorem 1. *The algorithm 1 solves the fault tolerant k-mutual exclusion problem using FDs of class $\mathcal{T}$ and $\mathcal{S}$, in an environment $\mathcal{E}_f$ with $f < N - 1$ faults provided that no process crashes before the initialization.*

Proof. The theorem 1 follows directly from Lemmas 1 and 3. □

7 Related work

Several authors have proposed fault-tolerant extensions both to token-based [16],[13],[5] and permission-based 1-mutual exclusion algorithms [1],[3]. The latter usually use the quorum approach.

Similarly to Raymond's algorithm [18], the token-based k-mutual exclusion algorithm proposed by Srimani and Reddy [20] naturally supports failures. It is inspired in Suzuki and Kasami's algorithm [21] and controls k tokens. If a node holds one of the k tokens, it can enter the critical section. However, likewise Raymond's, each crash reduces the number of nodes that can concurrently execute the critical section.

The majority of fault-tolerant permission-based k-mutual exclusion found in the literature use quorums [8],[9],[6],[10],[15]. Some of these algorithms exploit the k-coteries approach [9],[15],[10]. 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 [6] 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.

Two other k-mutual exclusion algorithms, [22] and [14] provide fault tolerance but for wireless ad-hoc networks. The authors in [22] 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 [14] 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.

8 Performance evaluation

8.1 Effectiveness

In order to evaluate the efficiency of Raymond's algorithms and our algorithm, we have developed a simulator. The initial number of nodes N is equal to 15 and the number of resource's units k is fixed to 5. Both algorithms execute the same scenario. Faults are injected during the run (signaled by a triangle in Figure 4).

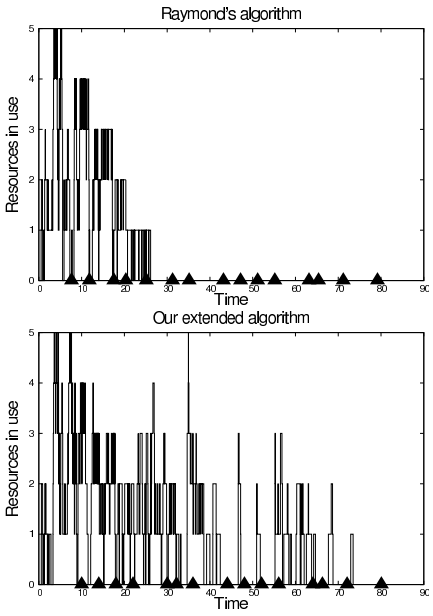


Figure 4: Efficiency comparison of Raymond algorithm and our extension

For both algorithms, we have measured the number of resource's units that can be simultaneously in use. We can clearly observe in Figure 4 that in Raymond's algorithm at every crash, the maximum number of concurrent accesses is decremented by one. When faults start being injected, some of the requests can still be satisfied provided that the total number of crashes is smaller than $k = 5$. However, after this bound, no new request is satisfied. On the other hand, our algorithm goes on progressing till $N - 1 = 14$ failures. Furthermore, the maximum number of units of the resource concurrently in use is not bounded by the number of failures. It just decreases because the number of concurrent requests decreases as well when faults are injected.

8.2 Number of messages

In our algorithm, at the initialization, each process sends once between $N-1$ and $2(N-1)$ messages. When no crash occurs, the number of messages per CS of our algorithm is equivalent to Raymond's algorithm, i.e., between $2N - k - 1$ and $2N - 1$ messages. In the presence of crashes, $N - 1 - |crashed_i|$ messages are sent by node i which detects the failure. However, in this case, the number of $REQUEST$ messages per CS decreases to $N - 1 - |crashed_i|$ and the number of

REPLIES decreases as well from $N - k - |crashed_i|$ to $N - 1 - |crashed_i|$.

9 Conclusion

We have presented in this article a new fault tolerant k-mutual exclusion algorithm. We assume an asynchronous network augmented with the failure detector of class $\mathcal{T}$, which is the weakest failure detector to solve the mutual exclusion problem, and the failure detector $\mathcal{S}$ for the initialization phase. Contrarily to Raymond's, our algorithm can dynamically detect node failures, tolerates $(N-1)$ failures instead of just $k-1$ as Raymond's naturally does, and always allows at most k processes to simultaneously execute the critical section, i.e., failures do not degrade the effectiveness of the algorithm as happens in Raymond's.

References

[1] D. Agrawal and A. E. Abbadi. An efficient and fault-tolerant solution for distributed mutual exclusion. *ACM Trans. Comput. Syst.*, 9(1):1–20, 1991.

[2] S. Bulgannawar and N. H. Vaidya. A distributed k-mutual exclusion algorithm. In *Int. Conference on Distributed Computing Systems*, pages 153–160, 1995.

[3] G. Cao, M. Singhal, and N. Rishe. 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.

[4] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of ACM*, 43(2):225–267, March 1996.

[5] 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.

[6] Y.-I. Chang and B.-H. 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.

[7] 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.

[8] S. Huang, J. Jiang, and Y. Kuo. k-coteries for fault-tolerant k entries to a critical section. *Distributed Computing Systems, 1993., Proceedings the 13th International Conference on*, pages 74–81, 1993.

[9] J.-R. Jiang, S.-T. Huang, and Y.-C. Kuo. Cohorts structures for fault-tolerant k entries to a critical section. *IEEE Transactions on Computers*, 46(2):222–228, 1997.

[10] H. Kakugawa, S. Fujita, M. Yamashita, and T. Ae. Availability of k-coterie. *IEEE Trans. Comput.*, 42(5):553–558, 1993.

[11] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[12] 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*, pages 408–411, 1992.

[13] 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.

[14] 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*, 1(3):156–167, 2006.

[15] M. L. Neilsen and M. Mizuno. Nondominated k-coteries for multiple mutual exclusion. *Inf. Process. Lett.*, 50(5):247–252, 1994.

[16] 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.

[17] 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.

[18] K. Raymond. A distributed algorithm for multiple entries to a critical section. *Inf. Process. Lett.*, 30(4):189–193, 1989.

[19] G. Ricart and A. K. Agrawala. An optimal algorithm for mutual exclusion in computer networks. *Commun. ACM*, 24(1):9–17, 1981.

[20] 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.

[21] I. Suzuki and T. Kasami. A distributed mutual exclusion algorithm. *ACM Trans. Comput. Syst.*, 3(4):344–349, 1985.

[22] J. Walter, G. Cao, and M. Mitrabhanu. A k-mutual exclusion algorithm for wireless ad hoc networks. In *ACM POMC'01*, 2001.

[23] S. Wang and S. D. Lang. A tree-based distributed algorithm for the k-entry critical section problem. In *Proc. of the 1994 International Conference on Parallel and Distributed Systems*, pages 592–599, 1994.