Performance evaluation of a fair fault-tolerant mutual exclusion algorithm

Julien Sopena, Luciana Arantes, and Pierre Sens
LIP6 - Université de Paris 6 - INRIA Rocquencourt
4, Place Jussieu 75252 Paris Cedex 05, France.
email: [julien.sopena,luciana.arantes,pierre.sens]@lip6.fr

Abstract

This paper presents an efficient and fair fault-tolerant token-based algorithm for achieving mutual exclusion. It is an extension of the Naimi-Trehel algorithm that uses a distributed queue of token requests and a dynamic tree. In case of failures, our algorithm tries to recover the requests' queue by gathering intact portions of the one which existed just before the failure. Thus, fairness of token requests is preserved despite failures. Furthermore, the use of broadcast is minimized when rebuilding the dynamic tree.

Experiment results with different fault injection scenarios show that our approach presents a fast failure recovery and low message broadcast overhead.

1 Introduction

Mutual exclusion is a fundamental paradigm in distributed systems ensuring that only one process can execute a critical section (CS) at a given time. Several distributed mutual exclusion algorithms exist in the literature ([3], [11], [12], [14], [10], [7], [8],[2]). Among them, an important class provides mutual exclusion based on a unique token circulating among nodes ([14], [10],[7],[8], [2]). The node that currently holds the token has exclusive permission to enter into its critical section.

Token mutual exclusion algorithms have a low average message cost. In particular, tree-based ones have logarithmic message complexity $\mathcal{O}(log(N))$, where N represents the number of nodes in the system. However, it is well-known that they suffer from poor failure resiliency. Thus, in this paper we propose an extension to the Naimi-Tréhel [7] tree-based token mutual exclusion algorithm in order to make it fault tolerant. Our solution tolerates multiple node failures. The paper also presents performance results comparing our fault tolerant extension and the one proposed by Naimi and Tréhel in [6]. A preliminary version of our extension, describing its basic principles and a sketch of proof, can be found in [13].

Naimi-Tréhel's algorithm manages two dynamic structures: (1) a logical tree, where the root is the last node that will receive the token among the current requesting ones and (2) a distributed queue of pending token requests. In order to add fault tolerance capability to Naimi-Tréhel's algorithm, we have modified the original one by adding one message per token request. This has minimal impact on performance in absence of failures. The aim of introducing such a message is to enable the reconstruction of the distributed queue of token requests by gathering intact portions of the queue which existed just before the failure. By recovering this queue, the algorithm avoids both token request retransmissions and the reinitialization of the above mentioned logical tree of requests. Furthermore, fairness is an important property in mutual exclusion distributed algorithms. In the context of Naimi-Tréhel's algorithm, this property is provided by satisfying token requests in the order kept in the distributed queue. Therefore, our proposal for recovering the queue in case of failure contributes to preserve the fairness of the algorithm.

The organization of this paper is as follows. Some related work is given in section 2. Section 3 introduces the original Naimi-Tréhel algorithm and their fault tolerant extension. In Section 4, we describe our fault-tolerant extension to the original algorithm. A performance comparison of the two extensions is presented in section 5, and the last section concludes our work.

2 Related Work

Several authors have proposed fault-tolerant extensions to token-based mutual exclusion algorithms. Nishio and al. [9] have developed a resilient extension to Suzuki-Kasami's algorithm [14] using broadcast. To regenerate the token, the algorithm requires an acknowledgement from every other site. Thus, the failure of a single site will delay the to-

ken's regeneration until the site comes back up. Arguing that Nishio and al.'s approach was very time consuming, Manivannan and Singhal proposed a new algorithm in [4] which requires to collect information from non faulty nodes only. However, this algorithm requires that at least two specific sites be up (the last site i to have executed the CS and the site to which i sent the token).

A tree-based mutual exclusion algorithm which tolerates failures is presented in Chang and al.[1]. Resiliency is achieved by using both redundant communication paths and an election mechanism. Nevertheless, the addition of alternative paths and avoidance of cycles may increase traffic communication. Naimi and Tréhel present in [6] a fault tolerant extension to their own algorithm. In the absence of failure, the original algorithm is not modified. However, recovery from failure is very expensive in terms of messages since it requires multiple broadcasts. This algorithm is described in section 3. Muller [5] also proposes a fault-tolerant extension to the original Naimi-Tréhel algorithm. In his solution, a ring communication structure, which includes all nodes of the system, is used for detecting a node failure. This solution has two drawbacks : it lacks scalability and does not allow multiple node failures.

3 The original algorithm of Naimi-Tréhel and their fault tolerant extension

The Naimi-Tréhel algorithm [7] is token-based. It maintains a logical dynamic tree structure, named *last tree*, such that the root of the tree is always the last node that will receive the token among current requesting nodes. A second structure, named *next queue*, is a distributed queue that controls the nodes which are waiting for the token. Each node just holds information about its $last$ and $next$ nodes.

Initially, the root of the $last$ tree is the token holder and the $last$ of all other nodes points to the root. A token request travels along a path of $lasts$ to the root. When receiving this request, each node along this path sets its $last$ pointer to the current requester i.e., the tree is modified dynamically. When a request arrives at the root, the latter updates its $next$ to point to the requester. When a site releases the CS, the token is sent to the site indicated by its *next* pointer.

Figure 1 shows an example of a Naimi-Tréhel execution with 4 nodes. Initially (a), site A is the root and holds the token. A new configuration of the *next* queue and the *last tree* is shown in (b) and (c) after node B and then C have requested the token. When A releases the CS, it sends the token to B and sets its $next$ to NIL, as shown in (d).

In [6], Naimi and Tréhel propose a fault tolerant version of their algorithm to recover from multiple crash failures. In the absence of failure, the original algorithm is not modified. It is extended to detect site failures, recover from them, and regenerate the token. A requesting node S_i suspects a failure when it does not receive the token after a certain timeout. In this case, S_i broadcasts a *CONSULT* message in order to check the state of its predecessor node in the *next* queue. Upon receiving this message, node S_j answers S_i only if the latter is S_j's $next$. If S_i does not receive another response within a timeout, it suspects that either its predecessor has crashed or its token request has been lost. S_i then broadcasts a *FAILURE* message to detect the presence of the token. S_i then has two alternative approaches, an *individual failure recovery* or a *global reinitialization*, depending on whether the token was lost or not:

- Individual failure recovery: if S_i receives an answer from the token holder S_k, then S_i retransmits its request directly to S_k. The broken *next* queue is not rebuilt.

- Global reinitialization: if after a timeout, S_i has not received an answer to its *FAILURE* message, it becomes a candidate to regenerate the token. S_i then broadcasts an *ELECTION* message. If several nodes are candidate, the site with the smallest identifier wins the election and broadcasts an *ELECTED* message to inform all sites that it is the new token owner. When receiving this message, a site sets its $last$ to the new token owner and its $next$ to NIL. Pending requests are lost, and sites must renew their requests.

Notice that, if different sites detect the failure simultaneously, many concurrent individual failure recoveries may take place, inducing a considerable message overhead. The total cost of these individual recoveries is much higher than a global reinitialization. Another remark is that the longer the *next* queue, the higher the probability that the token is lost, because this increases the probability that the crashed node belongs to the *next* queue and that the token will be sent to it.

4 Fair fault-tolerant algorithm

In this section we present our own fault tolerant extension to the original Naimi-Tréhel algorithm.

We consider a distributed system consisting of N sites $\Pi = \{S_1, S_2, \ldots, S_N\}$, which are fully connected, communicating only by message passing. Communication channels are reliable, but messages might be delivered out of order. The system is synchronous i.e., both process speed and message transmission times are bounded. However, there is no bound assumption about the time for executing a critical section. Sites can fail only by crashing, and crashes are permanent. The model tolerates $N-1$ node failures. The words site and node are interchangeable.

4.1 Description of the algorithm

The main insight of our approach is to reconstruct the distributed queue of pending requests (the *next* queue) by

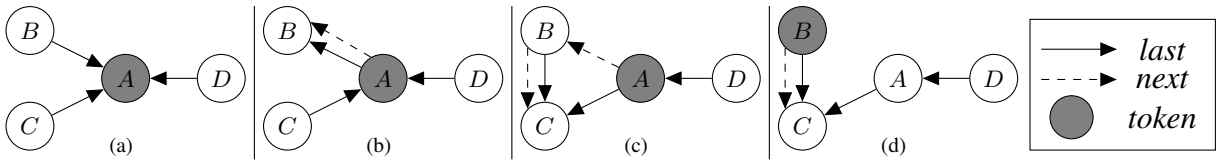


Figure 1. Example Naimi-Tréhel execution

re-assembling disconnected portions of the queue, as it existed before the failure. In the case where this is not possible, token requests must be resent and a new *last* tree created in accordance with the new *next* queue.

A site's *next* indicates its successor, i.e. the next site that will receive the token after it. However, a site is neither aware of its predecessors in the *next queue*, neither of its position in the *next* queue. Thus, in order to give such information to a requesting node, we have added an acknowledgment message to each critical section request message. This message informs a requesting node about both its predecessors and the position that it will keep in the *next* queue.

Figure 2 describes our fault tolerant extension to the Naimi-Tréhel algorithm.

For each site S_i, the original Naimi-Tréhel algorithm defines the following local variables:

- $last$: points to the last node that has requested the token to S_i.
- $next$: points to the node that will receive the token when S_i releases it.
- req: boolean variable that indicates if S_i has requested the token or not.

We add the following variables :

- *pos*: current position of S_i in the *next* queue. The site holding the token keeps the smallest position. A site loses its position when it leaves the queue.
- *newPred* : a tuple that holds the identifier, position, and the following node of the probable new predecessor of site S_i in the $next$ queue after failure recovery.
- *candidate* : boolean variable that indicates if node S_i has started a failure recovery process or not.

For controlling the reception of messages and detection of failures our algorithm uses three timers : *CommitTimer*, *TokenTimer* and *ReconnectionTimer*. When they trigger, *TimeoutCommit*, *TimeoutToken* or *TimeoutReconnection* routines are executed respectively. Only one timer can be active at a given time. If a new timer is set, the previous one is canceled.

As in the original Naimi-Tréhel's algorithm, when a site S_i wishes to execute a CS, it sends a request (*REQ* message) to its $last$ and sets its $last$ pointer to NIL (lines 4 and 5). In our fault-tolerant algorithm, before waiting for the token, S_i arms *CommitTimer* (line 6). Notice that in this case, S_i expects to receive an acknowledgment to its request (a *COMMIT* message) from its closest predecessor node before receiving the token itself.

When the root of the *last* tree, S_j, receives S_i's token request, if it is waiting for the token or executing the CS, it updates its *next* to S_i, as in the original algorithm (line 18). However, it also sends a *COMMIT* message to S_i confirming the reception of the request (lines 19-20). A *COMMIT* message contains the two following informations:

- S_i's k predecessors i.e, S_j and S_j's $k-1$ predecessors, k being a configurable parameter.

- S_j's position in the *next* queue.

It may happen that when S_j sends a *COMMIT* message to S_i it does not have all the above informations yet. In this case, it still sends a *COMMIT* message to S_i with the information it holds, which consists of at least its identifier. As soon as it receives the remaining information, it sends it in a new message or includes it in a request message to S_i.

Failure recovery - acquaintance with predecessors

Upon receiving a *COMMIT* message (line 29), S_i registers its position in the *next* queue as well as the information about its predecessors, and goes on waiting for the token. However, during this waiting time, S_i periodically checks the liveness of its closest predecessor (line 34). The frequency of this check is controlled by *TokenTimer* (line 45).

If S_i's closest predecessor does not answer (line 35), S_i checks the liveness of the other $k-1$ predecessors (line 36). Among all the answers received from its non faulty predecessors, S_i chooses the one, S_x, with the greatest position to become its closest predecessor. S_i then sends a *CONNECTION* message to S_x (line 37). When receiving this message (lines 57), S_x takes S_i as its new successor and sends a *COMMIT* message to S_i in order to inform the latter of its new position and its k predecessors (line 59). However, it may happen that all of S_i's k predecessors have crashed. In this case, S_i diffuses a *SEARCH_POS* message (line 40) which contains its current position and the list of its k predecessors, which are all faulty. It then arms *ReconnectionTimer*, and waits for position information messages (line 43). When a site Sj receives the *SEARCH_POS* message, it will send a *POSITION* message to S_i only if its position is smaller than $S_i's$ (lines 62). Furthermore, it may happen that S_j's *last* is currently one of the k faulty predecessors of S_i. In this case, it updates its *last* to S_i. In its turn, S_i (by means of the variable $NewPred$) keeps track of the greatest position received during *ReconnectionTimer* delay (lines 66-69). When this time elapses (function *TimeoutReconnection*), the site with the greatest position,

```
1  ReqCS ()
2      req ← true
3      if last ≠ NIL then
4          Send ⟨REQ, S_i⟩ to last
5          last ← NIL
6          Alarm (CommitTimer)
7          Wait for ⟨TOKEN⟩
8          /* Enter critical section */

9  ReleaseCS ()
10     req ← false
11     if next ≠ NIL then
12         Send ⟨TOKEN⟩ to next
13         next ← NIL
14         pos ← -1;

15 ReceiveREQ (S_j)
16     if last = NIL then
17         if req = true then
18             next ← S_j
19             Send ⟨COMMIT,
20               [S_i,pred[0],..,pred[k-2]], pos⟩ to S_j
21         else
22             Send ⟨TOKEN⟩ to S_j
23             pos ← -1;
24     else
25         Send ⟨REQ, S_j⟩ to last
26     last ← S_j

27 ReceiveToken ()
28     Reset alarm

29 ReceiveCommit ([S_o,..,S_{k-1}], pos_S_j)
30     pos ← pos_S_j + 1
31     pred[..] ← [S_o,..,S_{k-1}]
32     Alarm (TokenTimer)

33 TimeoutToken ()
34     check the liveness of pred[0]
35     if pred[0] failed then
36         if ∃ a minimal x, s.t. pred[x] is alive then
37             Send ⟨CONNECTION, S_i⟩ to pred[x]
38             Alarm (TokenTimer)
39         else
40             broadcast ⟨SEARCH_POS,pos,pred[]⟩
41             newPred.id ← NIL
42             newPred.pos ← -1
43             Alarm (ReconnectionTimer)
44     else
45         Alarm (TokenTimer)

46 TimeoutReconnection ()
47     if newPred.id = NIL then
48         Regenerate new Token
49         pos ← 0
50     else
51         if newPred.next ≠ NIL then
52             Send ⟨CONNECTION, S_i⟩ to newPred.id
53         else
54             Send ⟨REQ, S_i⟩ to newPred.id
55         Alarm (CommitTimer)
56     candidate ← false

57 ReceiveCONNECTION (S_j)
58     next ← S_j
59     Send ⟨COMMIT, [S_i,pred[0],..,pred[k-2]], pos⟩ to S_j

60 ReceiveSEARCH_POS (S_j, pos_j, faultyPred[])
61     if pos ≠ -1 ∧ pos < pos_j then
62         Send ⟨POSITION, S_i, pos, next⟩ to S_j
63     if ( last ∈ faultyPred[] ) then
64         last ← S_j

65 ReceivePOSITION (S_j, pos_j, next_j)
66     if newPred.pos < pos_j then
67         newPred.id ← S_j
68         newPred.pos ← pos_j
69         newPred.next ← next_j

70 TimeoutCommit ()
71     newPred.id ← NIL
72     newPred.next ← NIL
73     newPred.pos ← -1
74     candidate ← true
75     broadcast ⟨SEARCH_QUEUE⟩
76     Alarm (ReconnectionTimer)

77 ReceiveSEARCH_QUEUE (S_j)
78     if S_j is the winner of the last election then
79         if pos ≠ -1 then
80             Send ⟨POSITION, S_i, pos, next⟩ to S_j
81         if candidate then
82             Send ⟨REQ, S_i⟩ to S_j
83             Alarm (CommitTimer)
84             candidate ← false
85         if not req ∨ pos ≠ -1 then
86             last ← S_j
87         else
88             last ← next
```

Figure 2. Fair fault-tolerant mutual exclusion algorithm

$NewPred.id$, will become S_i's closest predecessor. In order to reconnect to this site, S_i sends a *CONNECTION* message to it (line 52). On the other hand, if S_i does not receive any answer at all to its *SEARCH_POS* query, it concludes that all its predecessors have crashed and the token is consequently lost. S_i then regenerates the token (lines 48-49).

Failure recovery - no acquaintance with predecessors

Let us now discuss the case when S_i does not receive a *COMMIT* message in response to its *REQ* message when *CommitTimer* elapses. S_i will try to reconnect itself to the $next$ queue by choosing the site with the greatest position to become its closest predecessor. To this end, S_i diffuses a *SEARCH_QUEUE* message and arms the *ReconnectionTimer* timer, waiting for replies (lines 75-76). When receiving a *SEARCH_QUEUE* message, every site S_j that has a position in the *next* queue answers to S_i with a *POSITION* message which contains S_j's position in the *next queue*, as well as whether if it has a *next* or not (lines 79-80).

In order to explain S_i's failure recovery procedure at this point, we first consider that S_i is the only site which detects the failure, or that it is the winner of the election among concurrent sites that have also detected the failure (the election is detailed below). Similarly to a *SEARCH_POS* query, S_i chooses the site $NewPred.id$ with the greatest position among all the *POSITION* messages received within *ReconnectionTimer* (lines 66-69). However, S_i can only reconnect itself to $NewPred.id$ if the latter has also informed that it has a $next$. If such information was given, S_i concludes that $NewPred.id$'s *next* has failed. S_i then sends a CONNECTION message to $NewPred.id$ (line 52) in order to force it to reconnect itself to S_i, If $NewPred.id$ has no *next*, S_i directly resends a token request to it (line 54), as $NewPred.id$ is the last node of the *next* queue and therefore the root of the *last* queue. In both cases, S_i arms *CommitTimer*, waiting for a *COMMIT* message from $NewPred.id$, its new closest predecessor (line 55). On the other hand, if S_i does not receive any *POSITION* message for its *SEARCH_QUEUE* query during *ReconnectionTimer* delay, it regenerates the token (line 48).

Contrary to the case when S_i receives a *COMMIT* message, the order of previous token requests is not preserved anymore. The *last tree* must be reconstructed to be consistent with the new *next queue*. This reconstruction is done dynamically without any additional overhead in terms of message since all the information a site needs is transmitted to it in the *SEARCH_QUEUE* message. The *last tree* is reconstructed as follows: sites that either do not wait for the token or waiting sites that know their position (line 85) set their *last* pointer to S_j (line 86) since S_j is considered by them as the last site which asked for the CS; sites waiting for the token without a position in the *next* queue set their *last* pointer to the same value as their *next* pointer (line 88). A site in such a case is sure that its $next$ has requested the token after it. This approach avoids cycles in the *last* tree.

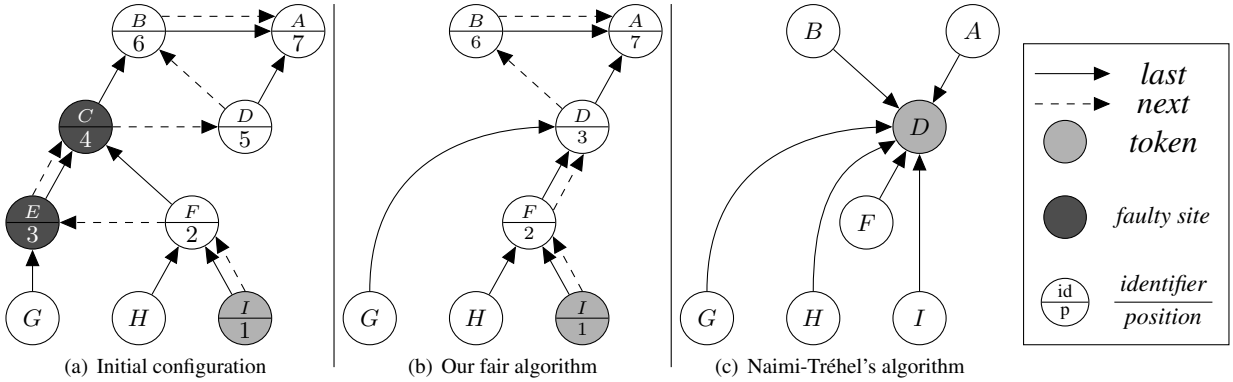
It remains to explained how our algorithm manages concurrent detections of failure. Simultaneous failure detections may bring the *next* queue and *last* tree to an incoherent state or even imply the lost of the token uniqueness property. Therefore, an election mechanism is necessary in this case for deciding which site will actually perform the failure recovery. In order to detect concurrent attempts at failure recovery, our algorithm uses the Lamport logical clock [3]. The value of the logical clock (counter, identifier) of S_i is included in the *SEARCH_QUEUE* message sent by S_i. For the sake of simplification, we have not included logical clock timestamps in the pseudo codes of Figure 2. We indicated that when S_i receives a *SEARCH_QUEUE* message from S_j, it verifies if the latter is the winner or not of the election (line 78). S_i does so by comparing logical clock timestamps of the *SEARCH_QUEUE* messages it received. If they are concurrent, i.e. they have the same counter value, the winner is the site with the greatest identifier.

Whenever a candidate S_is for receives a *SEARCH_QUEUE* message from another candidate S_j which indicates that the latter is the current winner of the the election, S_i gives up failure recovery by sending a token request directly to S_j (lines 82-84). We consider that a node is the final winner of the election if it still is a candidate after *ReconnectionTimer* has elapsed.

4.2 Example of Failure Recovery

Figure 3 shows an example of failure recovery for both algorithms. We consider that there are two faulty sites, C and E, as shown in figure 3.a. Therefore, the *next queue* is broken into two portions I,F and D,B,A. All sites in the *next* queue have a position (i.e., they have received a *COMMIT* message) and keep information about its two closest predecessors ($k = 2$). Site I is the current token holder.

D eventually detects the failures of C and E when checking the liveness of its two predecessors (lines 35-36). D then broadcasts a *SEARCH_POS* message (line 40), which includes its current position ($pos_D = 5$) and its k faulty predecessors ($Pred[]_D = C, E$). Sites F and I, whose position is smaller than 5, respond to D with a *POSITION* message (lines 61-62). Furthermore, sites G and F, whose *last* variables point to faulty nodes, set their *last* to D (lines 63-64). When *ReconnectionTimer* elapses, D chooses F, which has the greatest position ($pos_F = 2$) among its predecessors to become its closest predecessor, and sends a *CONNECTION* message to F (line 52). Upon receiving such message (lines 57-59), F sets its *next* to D and sends it a *COMMIT* message which contains F's position and two of the new predecessors of D. Finally, D updates its position ($pos_D = 3$) in the *next queue* and the



(a) Initial configuration (b) Our fair algorithm (c) Naimi-Tréhel's algorithm

Figure 3. Example of failure recovery

information about its two predecessors ($Pred[]_D = F, I$), going on waiting for the token (lines 29-32). Figure 3.b shows the final configuration after failure recovery.

In our execution example of the Naimi-Tréhel fault-tolerant algorithm, we also consider the initial configuration of figure 3.a and that node D detects the failure after the token was lost when sent to E (i.e., F and I have executed their critical sections). Getting no response either to its *CONSULT* broadcast or its *FAILURE* broadcast, D concludes that the token is lost. Supposing that D is the only site which has detected the failure after having broadcasted an *ELECTION* message, it regenerates a new token and broadcasts an *ELECTED* message to inform that it is the new token holder. Both the *last* tree and the *next* queue will be reinitialized i.e., every site will point its *last* to D and will set its *next* to NIL, when receiving the *ELECTED* message. Notice that nodes A and B will have to resend their request to D. Figure 3.c shows the final configuration after failure recovery.

4.3 Relevant algorithm features

Some of the features of our fault tolerant algorithm deserve being emphasized:

Low free failure overhead: The cost of having the predecessors' information mechanism is relatively low in terms of messages. We have just added one message per token request. Thus, the message complexity of the algorithm grows up from $log(n)$ to $log(n) + 1$, remaining $\mathcal{O}(logN)$.

Efficient failure recovery: By minimizing the number of broadcasts and request retransmissions, our algorithm reduces recovery message overhead.

Failure isolation: By periodically checking the liveness of the closest predecessor, our algorithm may detect a fault predecessor node and recover the failure *before* the token arrives at this predecessor. In this case, the execution of the critical section by non-faulty sites and the execution of the failure recovery are completely overlapped.

Fairness: The *next queue* is reconstructed by assembling portions of the queue, minimizing the number of token request retransmissions and preserving the order of requests.

5 Performance Results

This section describes a set of performance evaluation results aimed at comparing the efficiency of both our fault tolerant extension and Naimi-Tréhel's one.

5.1 Environment and Parameters

The experiments were conducted on a 20-nodes cluster. Each node is equipped with two 2.8GHz Xeon processors and 2GB of RAM, running Linux 2.6. Nodes are linked by a 1 Gbit/s Ethernet switch. Using hyperthreading, each processor can run two "virtual nodes". Thus $N = 80$ nodes.

The algorithms were implemented in C using UDP sockets. Each application process that runs on a single virtual node executes 5 critical sections. On each experiment, the following number of simultaneous faults were respectively injected: 1, 3, 5, 8, 20, or 40. Each experiment was executed 20 times. The results represent the average value.

An application is characterized by:

- α: time to execute the critical section.
- β: mean time interval between the release of the CS by a node and its request by this same node.
- ρ: the ratio β/α, which expresses the frequency with which the critical section is requested.

We have developed three types of applications having **low**, **intermediate**, and **high** levels of parallelism. These three levels are respectively expressed by ρ:

- $\rho = 1$: an application where almost all sites wait for the token. The *next* queue is long.
- $\rho = N$: an application where some sites wait for the token. The *next* queue is small.
- $\rho = 2 * N$: a highly parallel application where sites do not compete to get the CS. The *next* queue is often empty, having at most one site.

The choice of timeout values is very important for fault tolerant algorithms' effectiveness. Having measured the average time (*APing*) and the maximum time (*MPing*) of a round-trip ping message between two nodes of our cluster, we have defined the following three timer values:

- Passive: $PassT = N * MPing = 11.85s$
- Intermediate: $InterT = log2(N) * MPing = 3.95s$
- Aggressive: $AggrT = log2(N) * APing = 0.32s$

For each experiment, the same timeout value is set to both *CommitTimer* and *TokenTimer* as well as to all timeouts used by the Naimi-Tréhel algorithm (*NT-Timer*). *ReconnectionTimer* is set to $10 * MPing$. This value is chosen to minimize false suspicions.

We have considered three metrics: **number of sent messages**, **number of received messages**, and the **obtaining time** i.e., the time between the moment a node requests the CS and the moment it gets it.

The difference between the number of sent and received messages expresses the use of broadcast by an algorithm.

Concerning the *obtaining time*, it tends to be high when $\rho = 1$ since there are always many nodes in the *next* queue. However, when $\rho = 2N$, the *obtaining time* is quite short because the *next* queue is empty almost all the time.

Note: In all the figures presented in this section, the abscissas of the curves represent the number of crashes. Thus, when analysing the curves the reader must remember that when the number of crashes increases, the number of nodes that actually execute the algorithm decreases. Consequently, if we do not consider the messages due to failure detection or recovery, the total number of sent or received messages and the *obtaining time* decrease as well.

5.2 Application behavior influence

The current experiments analyze both algorithms when applications with different levels of parallelism are executed. *CommitTimer*, *TokenTimer*, and *NT-Timer* are set to the intermediate $InterT$ timer and the number of predecessors k is fixed to 2.

Number of sent messages: When comparing the number of sent messages with no crash of figures 4.a, 4.b, and 4.c, we can notice the impact of ρ in the failure detection mechanism of both algorithms : the shorter ρ is, the longer the obtaining time, and therefore the higher the number of false suspicions. This explains why the number of sent messages decreases when ρ increases in the absence of failures.

We can also observe for the same figures without crash that our algorithm presents an overhead in terms of sent messages when compared to Naimi-Tréhel's, which is due both to *COMMIT* and failure detection messages (we will come back to this point when discussing the received messages). However, when $\rho = 1$ and there is a small number of crashes, the difference of sent messages between both the algorithms decreases. In our algorithm, as the *next* queue is long, the predecessor's failure recovery mechanism is executed. In Naimi-Tréhel's, a total reinitialization of the structures takes place. Therefore, the mentioned decrease is due to the effectiveness of our predecessors' mechanism. On the other hand, when 50% of the nodes crash, our mechanism of checking the predecessors' liveness becomes more costly. Indeed, if almost all of the predecessors are faulty, it is easier and more efficient to reinitialize (*next* queue and *last* tree), even at the expense of token request fairness.

When ρ increases, the number of sites which are not in a waiting state increases as well. These sites are likely to send a request along a broken path of the *last* tree. We observe that our algorithm presents quite a regular behavior when the number of faults increases, contrary to Naimi-Tréhel's. This happens since in our algorithm, the loss of just one of these requests starts a global reconstruction (*SEARCH_QUEUE* mechanism, see section 4.1). However, in Naimi-Tréhel's, each request loss launches an individual failure recovery (see section 3). The number of these individual recoveries depends on the probability of token loss and the damage in the *last* tree. The greater the number of failures, the greater the *last* tree's damage is and this increases the number of individual failure recoveries. However, beyond a certain number of crashes, the token is eventually lost and therefore a global reinitialization is executed. Such a behavior increases with ρ since the *next* queue is smaller (see the last paragraph of section 3).

Number of received messages: The significant difference in the number of received messages between both algorithms (Figures 4.d, 4.e, and 4.f) proves that our algorithm has reduced the usage of broadcasts when compared to Nami-Tréhel's one.

By comparing the number of received messages in failure-free runs, we can also analyze the failure detection mechanism of the algorithms. For Naimi-Tréhel, when $\rho = 2N$, the number of messages is 3,700. However, when $\rho = 1$, this number goes up to 56,000 messages. This happens because the *obtaining time* for $\rho = 1$ is much higher then for $\rho = 2N$, and consequently the number of false failure suspicions. On the other hand, our algorithm does not present such huge variation of received messages. In case of a failure suspicion, the Naimi-Tréhel algorithm checks the liveness of its closest predecessor by broadcasting a *CONSULT* message while in our algorithm this checking is just resumed to a ping message. We can conclude then that the *obtaining time*, and implicitly the critical section duration, has a great influence on Naimi-Tréhel's failure detection mechanism, contrary to ours.

Obtaining time: When observing the results of Figures 4.g, 4.h, and 4.i, a first global remark is that our algorithm presents a shorter *obtaining time* for almost all experiences when compared to Naimi-Tréhel's in the presence of failure.

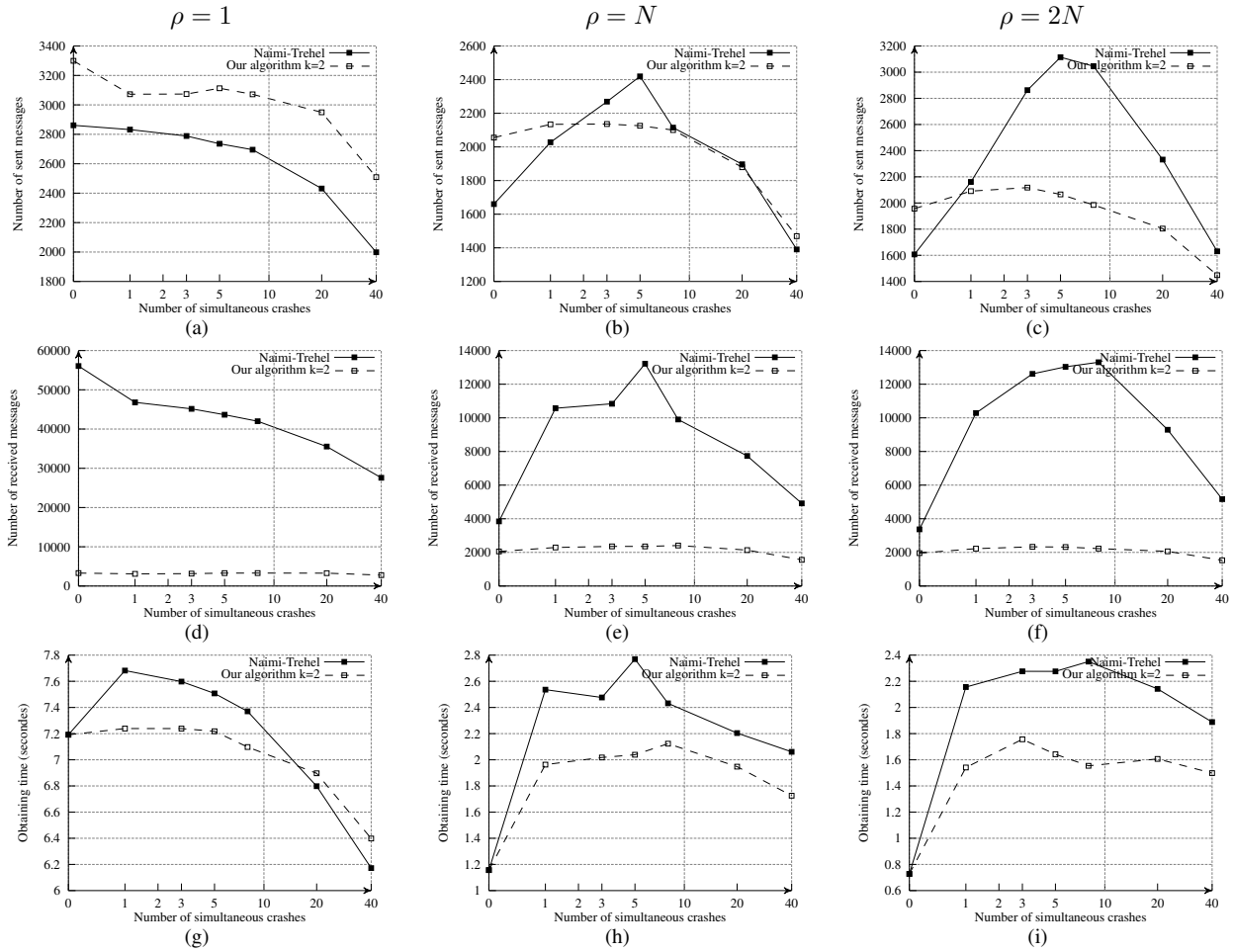


Figure 4. Application Behavior Influence

This gain is due to the fact that both our failure detection and recovery mechanisms are faster than those of Naimi-Tréhel. The only exception is for 50% crashes when $\rho = 1$. Indeed, it is easier and less costly to reinitialize the *next* queue when it is long and extremely broken than to try to recover it.

Another interesting point to discuss is the linear behavior of the *obtaining time*'s curve of our algorithm when $\rho = 1$ and the number of crashes varies from 0 to 5 (Figure 4.g). This behavior is due to the *failure isolation* feature of our algorithm, discussed in section 4.3. Since the *next* queue is long, failures are recovered before the token is sent to the faulty node, leaving then the execution of the algorithm undisturbed. When the *next queue* is smaller ($\rho = N$ or $\rho = 2N$), the failure isolation feature is not observed.

5.3 Failure suspicion timeout influence

We have measured the impact of timeout bounds on message overhead and the *obtaining time*. We have considered the three timer values, $PassT$, $InterT$ and $AggrT$, described in 5.1. *CommitTimer*, *TokenTimer* and *NT-Timer* are set to one of these values for each experiment. The parallelism level of the application is $\rho = N$ and the number of predecessors is $k = 2$.

Number of sent messages: Comparing the number of sent messages of figures 5.a, 5.b, and 5.c in failure-free executions, we can observe that our algorithm sends more messages than Naimi-Tréhel's. In figure 5.a, where the timeout value is high, the difference in the number of sent messages is due only to *COMMIT* messages since there are hardly any false suspicions in this case. By decreasing the timeout, the number of false suspicions to both algorithms increases and therefore so does the number of sent messages.

When there are crashes, we can see in figures 5.b and 5.c that the cost of our failure recovery mechanism does not increase with shorter timeouts ($InterT$ and $AggrT$), contrary to Naimi-Tréhel's. The considerable growth of sent messages overhead in Naimi-Tréhel is a consequence of the concurrent individual failure recoveries performed by the algorithm. On the other hand, the *SEARCH_QUEUE* elec-

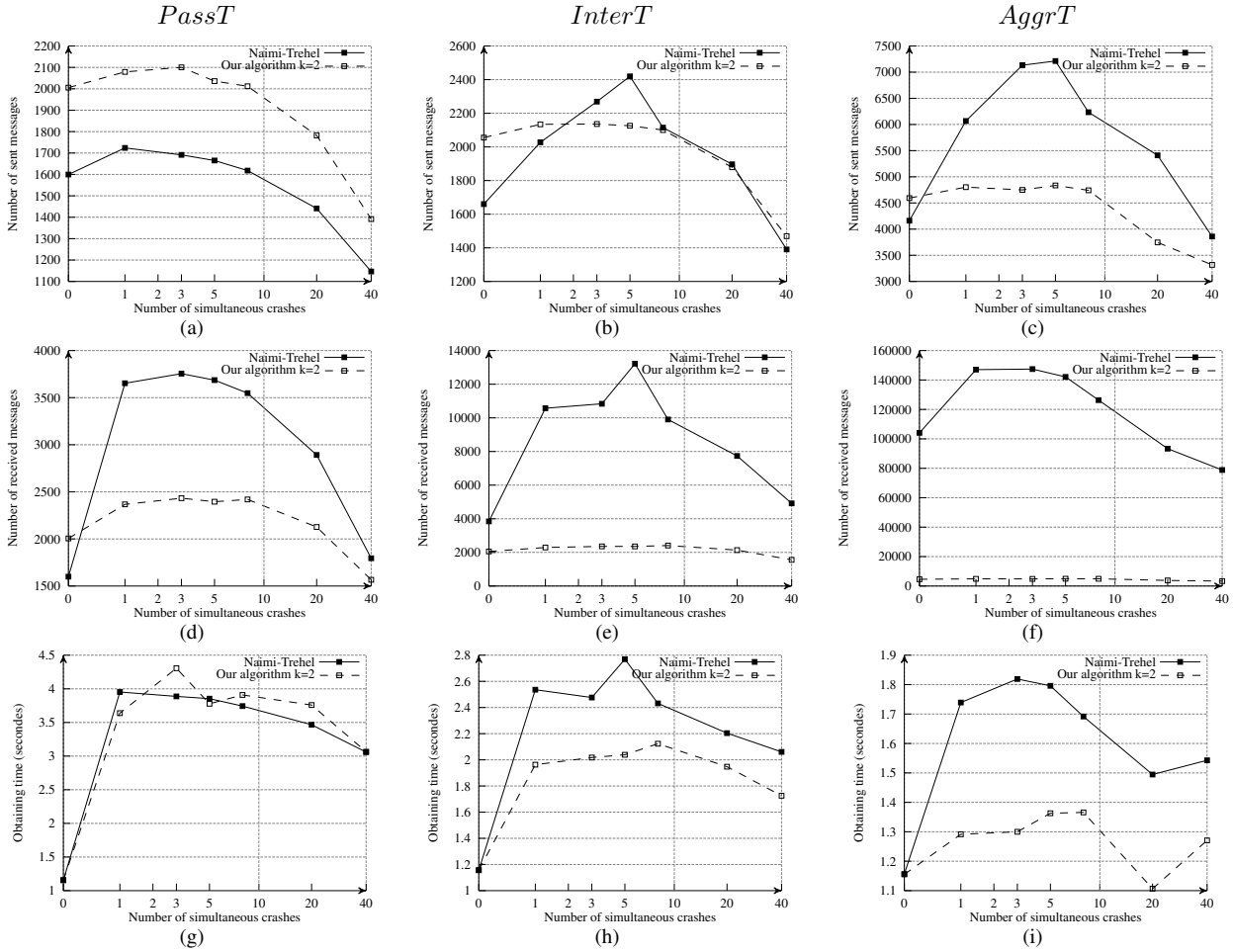


Figure 5. Failure Suspicion timeout influence

tion of our algorithm reduces the effect of concurrent individual's attempts at failure recovery. When the timeout value is high (figure 5.a), the majority of nodes are waiting for the token before the failure is detected. Thus, in the case of Naimi-Tréhel's algorithm, a global reinitialisation, which is less costly than the concurrent individual failure recoveries, takes place (see section 3). Such a reinitialisation has the same cost in terms of sent messages as our algorithm, which explains the similar behavior of both curves (the difference is due to *COMMIT* messages).

Number of received messages: In all curves (Figures 5.d, 5.e, and 5.f), independently of the timeout, we can see the heavy cost of multiple broadcasts usage by the Naimi Tréhel algorithm. This limits its timeout bound. The worst case is achieved with $AggrT$ with no crash: 100,000. Our algorithm is less subject to timeout variation: the number of received messages is around 2,000 for the three timeouts.

Obtaining time: Considering the high cost of the aggressive timeout, one could argue about the benefits of reducing the timeout. To answer this question, we should compare the *obtaining time* of the application for the three different timeouts (Figures 5.g, 5.h, and 5.i). For a small number of faults, the order of magnitude of the *obtaining time* is around 4s when the timer is set to $PassT$. If the latter is reduced to $InterT$, the order of magnitude falls to 2.1s for our solution and 2.4s for Naimi-Tréhel's one. By setting the timer to $AggrT$, the *obtaining time* is still reduced to 1.3s for our algorithm and to 1.8s for Naimi-Tréhel's. There is thus a real benefit in taking an aggressive timeout, despite possible false suspicions and bandwidth cost.

5.4 Predecessors number influence

In order to study the impact of the k parameter on the failure recovery mechanism of our algorithm, we set both *CommitTimer* and *TokenTimer* to the intermediate $InterT$ timer and we considered $\rho = 1$ i.e, an application with a long $next$ queue.

As illustrated in Figure 6, k has no impact on our algorithm in failure-free runs. Indeed, the k-predecessors in-

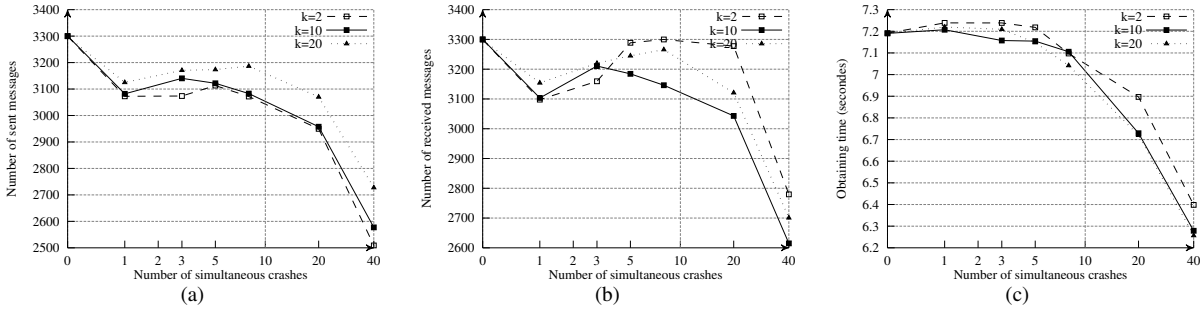


Figure 6. Predecessors number influence

formation is sent in the $COMMIT$ message and the number of predecessors keeps constant during failure detection. However, when there are few failures, the number of sent messages increases with greater values of k (figure 6.a). This happens since the number of sent messages for finding the first no faulty predecessor increases with k.

When the number of failures increases, the probability of having at least 2 successive faulty nodes increases as well. By analyzing the number of received messages (figure 6.b), we can observe that greater values of k ($k = 10$ and $k = 20$) reduce the number of $SEARCH\_POS$ broadcasts when compared to a small value of k ($k = 2$), as the predecessors' mechanism is more efficient with greater values of k.

Finally, we note a light gain in the obtaining time for $k = 10$ or $k = 20$ when there are many failures (figure 6.c) due to the effectiveness of our predecessors' mechanism.

6 Conclusions

We presented and evaluated a fault-tolerant token based mutual exclusion algorithm. Most existing algorithms do not preserve the request queue after failure recovering. Consequently, waiting nodes have to resend their requests, increasing message overhead. Our algorithm recovers the distributed request queue by re-assembling disconnected portions of the queue, as it existed before the failure. This approach minimizes the number of request retransmissions and preserves fairness of the algorithm despite failures.

In failure-free executions, the complexity of our algorithm is $\mathcal{O}(log(N))$. Therefore, our algorithm is scalable.

Experiments on a real environment were conducted in order to compare the performance of our algorithm with a fault-tolerant version of Naimi-Tréhel's algorithm, where the distributed waiting queue is reinitialized in case of failures. In the majority of the experiments, our algorithm has presented a faster and more efficient failure recovery. Performance results show that our fault tolerant approach does not depend on the application behavior, supporting applications with evolutive parallelism grains.

References

[1] 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.

[2] I. Chang, M. Singhal, and M. Liu. An improved O(log N) mutual exclusion algorithm. In *Proc. of the 1990 Int' Conference on Parallel Processing*, 1990.

[3] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7).

[4] 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.

[5] F. Mueller. Fault tolerance for token-based synchronization protocols. *Workshop on Fault-Tolerant Parallel and Distributed Systems, IEEE*, april 2001.

[6] M. Naimi and M. Trehel. How to detect a failure and regenerate the token in the log(n) distributed algorithm for mutual exclusion. *LNCS*, 312:155–166, 1987.

[7] M. Naimi, M. Trehel, and A. Arnold. A log (N) distributed mutual exclusion algorithm based on path reversal. *Journal of Parallel and Distributed Computing*, 34(1):1–13, 1996.

[8] M. L. Neilsen and M. Mizuno. A dag-based algorithm for distributed mutual exclusion. In *Proc. of the 11th Int' Conference on Distributed Computing Systems*, pages 354–360, Washington, DC, 1991.

[9] 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.

[10] K. Raymond. A tree-based algorithm for distributed mutual exclusion. *Trans. on Computer Systems*, 7(1):61–77, 1989.

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

[12] M. Singhal. A dynamic information structure for mutual exclusion algorithm for distributed systems. *IEEE Transactions on Parallel Distributed Systems*, 3(1):121–125, 1992.

[13] J. Sopena, L. Arantes, M. Bertier, and P. Sens. A fault-tolerant token -based mutual exclusion algorithm using a dynamic tree. In *Euro-Par'05*, LCNS, pages 654–663, 2005.

[14] I. Suzuki and T. Kasami. A distributed mutual exclusion algorithm. *Trans. on Computer Systems*, 3(4):344–349, 1985.