A Fault-Tolerant Token-Based Mutual Exclusion Algorithm Using a Dynamic Tree

Julien Sopena[1], Luciana Arantes[1], Marin Bertier[2], and Pierre Sens[1]

[1] LIP6 – Université Paris 6, INRIA, CNRS
{Julien.Sopena,Luciana.Arantes,Pierre.Sens}@lip6.fr
[2] LRI – Université Paris 11, CNRS
Marin.Bertier@lri.fr

Abstract. This article presents a fault tolerant extension for the Naimi-Trehel token-based mutual exclusion algorithm. Contrary to the extension proposed by Naimi-Trehel, our approach minimizes the use of broadcast support by exploiting the distributed queue of token requests kept by the original algorithm. It also provides good fairness since, during failure recovery, it tries to preserve the order in which token requests would have been satisfied had the failure not occurred.

1 Introduction

Mutual exclusion is a fundamental concept in distributed systems. Several algorithms have been proposed to solve the problem of mutual exclusion, serializing concurrent accesses to a shared resource. They can essentially be divided into two groups: *permission-based* (e.g. Lamport [2], Ricart-Agrawala [8], Maekawa [3]) and *token-based* (e.g. Suzuki-Kazami [9], Raymond [7], Naimi-Trehel [5]). Algorithms of the first group are based on the principle that a node may enter critical section only after having received permission from all the other nodes (or a majority of them [3]). The drawback of these algorithms is the high communication overhead. In the second group of algorithms, a system-wide unique token is shared among all nodes, and its possession gives a node the exclusive right to enter into the critical section, thus ensuring the safety property.

Some token-based algorithms, such as Raymond [7] and Naimi-Trehel [5], consider that nodes are organized in a logical tree and that a node always sends a token request to its father in the tree. Tree-based algorithms have an average lower message cost, and many of them result in a logarithmic message complexity $\mathcal{O}(logN)$ with regard to the number of nodes. Presenting better scalabilty, they can be more easily adapted to large scale configurations like grid and peer-to-peer environments [1]. Another advantage of these algorithms is the simplicity of their local data structures. However, a tree-based algorithm is very sensitive to node failure since it cannot tolerate even a single failure of one of the nodes in a token request path.

In this paper we propose a fault tolerant extension for the Naimi-Trehel token-based mutual exclusion algorithm. Naimi-Trehel's algorithm maintains

J.C. Cunha and P.D. Medeiros (Eds.): Euro-Par 2005, LNCS 3648, pp. 654–663, 2005.

two main data structures: a dynamic logical tree such that the root of the tree is always the last node that will have the token among the current requesting ones nodes, and a distributed queue that keeps token requests that have not been satisfied yet. The dynamic property of the request tree is strongly exploited in our solution. Let N be the number of nodes in the system. The new algorithm can tolerate at most $N-1$ node failures and the message overhead for failure recovery is relatively low.

Naimi and Trehel have proposed a fault tolerant extension of their own algorithm in [6]. 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, causing a high message overhead. Furthermore, the distributed queue of token requests has to be completely rebuilt.

We have modified the original Naimi-Trehel algorithm, introducing one additional message per token request. This modification has minimal impact on the original protocol in the absence of failures. On the other hand, our algorithm presents a lower cost in terms of messages in the presence of failures since it broadcasts at most one message when compared to the multiple broadcast messages of Naimi-Trehel's algorithm [6]. In contrast to the latter, the basic idea of our algorithm, in case of failure recovery, is to reconstruct the distributed queue of token requests by assembling disconnected portions of the previous queue. In addition to the low recovery cost, this approach exhibits the fairness property since it preserves the order in which token requests were previously queued.

We should mention that Mueller also presents in [4] a fault-tolerant extension of the Naimi-Trehel algorithm without broadcast support. However in his solution a ring communication structure, which includes all nodes of the system, is used for detecting a node failure as a message circulates constantly on it. Even if his solution does not use broadcast support, it also presents a lack of scalability, since the ring exhibits a message overhead that grows linearly with the number of nodes. Furthermore, his approach tolerates only one node failure.

The organization of this paper is as follows. Section 2 presents our considered system model. Section 3 briefly describes Naimi-Trehel's algorithm and outlines the problems for making it fault tolerant. Their fault tolerant version of this algorithm is described in section 4. In Section 5, we describe our fault-tolerant extension for the original Naimi-Trehel algorithm. A performance comparison of both fault-tolerant algorithms is presented in section 6, whilst the last section concludes our work.

2 General Model

We consider a distributed system consisting of a finite set of N sites $\Pi = \{S_1, S_2, \ldots, S_N\}$ that are spread throughout a network. Sites communicate only by sending and receiving messages. Every pair of sites is assumed to be connected by means of a reliable communication channel. However, messages may be delivered in a different order than the one they were sent in. The words site and node are interchangeable.

We consider a synchronous fully-connected network where process speeds and message transmission times are bounded. T_{msg} is the maximum latency for sending a message between two sites. Contrary to Naimi-Trehel, which considers that a critical section execution takes T_{cs} in average, our algorithm makes no assumption on the time for executing critical sections.

Sites can fail by crashing only, and this crash is permanent. $N - 1$ node failures are tolerated.

3 Naimi-Trehel's Algorithm

Naimi-Trehel's algorithm [5] is a token-based algorithm. It keeps two data-structures:

1. A logical dynamic tree structure such that the root of the tree is always the last site that will get the token among the current requesting ones. Requesting sites then form a logical tree pointing by probable token owners towards the root. Initially, the root is the token holder, elected among all sites. We call this tree the *last tree*, since each site keeps a local variable called *last* that points to the probable owner of the token.

2. A distributed queue which keeps critical section (CS) requests that have not yet been satisfied. We call this queue the *next queue*, since each site S_i keeps a local variable called *next* that points to the next site to whom the token will be granted after S_i leaves the critical section.

One **invariant of Naimi-Trehel algorithm** is that the root node of the *last* tree is always the tail node of the *next* queue.

When a site S_i wants to enter the critical section, it sends a request to its *last*. S_i then sets its *last* variable to itself and waits for the token. Site S_i becomes the new root of the tree.

Receiving S_i's token request message, site S_j can take one of the following actions: (1) S_j is not the root of the tree. It forwards the request to its *last* and then updates its *last* variable to S_i. Notice that the *last tree* is modified dynamically; (2) S_j is the root of the tree. If S_j is holding an idle token, it sends it back to S_i directly. On the other hand, if S_j holds the token but is in the critical section or is waiting for the token, S_j sets its *next* variable to S_i. At the end of the execution of critical section, S_j sends the token to its *next*.

An example of Naimi-Trehel's algorithm execution with four nodes is shown in Figure 1. Initially (a), site A is the root which holds the token. The local *last* variable of all nodes points to A. In (b), node B asks for the token by sending a request to its *last* ($last_B = A$). B becomes the new root ($last_B = B$). Then, A updates its *next* and *last* variables to point to B. In (c), C asks A for the token. The request is forwarded to B which updates its *next* to C ($next_B = C$). Both A and B update their *last* to C, since the latter is the last requester of the token (C becomes the new root of the tree). When A releases the critical section, the token will be sent to B as $next_A = B$.

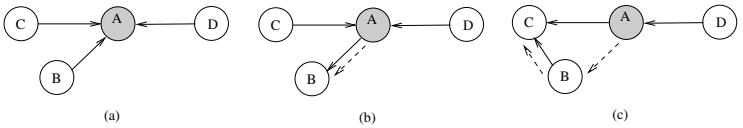

Fig. 1. Example of Naimi-Trehel's algorithm execution

The major challenges for making Naimi-Trehel algorithm fault-tolerant are:

1. *The faulty node is an intermediate node of the last tree.* In this case, if the faulty node were used for forwarding the token request before the failure, the request should be resent. Furthermore, we must be sure that the state of *last tree* is consistent before re-sending the request. However, in the Naimi-Trehel algorithm, while a request message is in transit, the tree is temporarily broken into several smaller rooted trees. Thus, finding a right path to the root may be impossible if the failure has occurred when the tree was in an unstable state.

2. *The faulty node belongs to the next queue.* In this case, it is not possible to known the path for token transmission anymore. Therefore, *next queue* must either be rebuilt from the beginning or by gathering disconnected portions of the queue which existed before the failure.

3. *The faulty node had the token.* In this case, the token must be regenerated and the uniqueness of the token must be guaranteed.

4 Naimi-Trehel Fault Tolerant Extension

In [6], Naimi and Trehel propose a fault-tolerant version of their algorithm. The original algorithm is not modified but some extensions are included in the algorithm to detect site failures, recover from failures, and regenerate the token.

To detect a site failure, site S_i, which requests the critical section, arms a timer T_{wait}. This timer depends on latency communication time (T_{msg}) and the average time (T_{cs}) for executing the critical section. If S_i does not receive the token after the expiration of T_{wait}, it suspects that a failure has occurred. Therefore, S_i broadcasts a *CONSULT* message to ask for the state of the other sites and arms a new timer, T_{elec}. When a site S_j receives this message, it answers to S_i only if the latter is its *next*. At the expiration of T_{elec}, if S_i does not receive any response, it is sure that a failure has occurred. S_i then broadcasts a *FAILURE* message to detect the presence of the token in one of the sites. A site replies to S_i if it owns the token.

If after a new T_{elec} delay, S_i has not received any answer to its failure message, it considers that the token is lost and it becomes a candidate to regenerate the token. It then broadcasts an *ELECTION* message. In case of concurrent election messages, the site with the smallest identifier is chosen. At the end, an *ELECTED* message is broadcasted to inform all sites of the new token owner. Finally, the identification of the *last* of each site is set to the new token owner. Notice that each node having requested the token before the failure has to re-send its token request.

5 Our Fault-Tolerant Algorithm

Contrary to the fault tolerant extension proposed by Naimi and Trehel, we have modified the original algorithm in order to provide the same guarantees in terms of fault tolerance and to optimise efficiency and complexity in the occurence of failures.

5.1 Principle of the Algorithm

The guiding principle of our algorithm is to reconstruct the *next queue* by gathering intact portions of the previous *next queue* which existed just before the failure. The aim of this reconstruction is to preserve the initial order of token requests as much as possible and to avoid request retransmitions, as is the case in Naimi-Trehel's solution. On the other hand, if the reconstruction is not possible, a new *next queue* will be created as well as a new *last tree*. The latter needs to be consistent with the former guaranteeing the invariant mentioned in section 3.

Considering the original algorithm, a site always knows, through its *next* variable, which site will receive the token after it, i.e. its successor in *next queue*. However, it is not aware of which site will grant the token to it, neither which sites will get the token before it. In other words, it is not aware of its predecessors in *next queue*. Thus, in order to inform a node of its predecessor in *next queue*, we have added a confirmation mechanism to the original algorithm for each token request. Whenever a site S_j updates its *next* variable, i.e. S_j is in the last element of the *next* queue and received a token request, it sends a *COMMIT* message to the requester in order to confirm the reception of the request and to communicate the identification of its predecessors. The *next queue* then keeps a ordering where the smallest position corresponds to the site which has the token. A site loses its position when it leaves the queue. Initially the token holder has position zero. A *COMMIT* message sent to the requester S_i, by site S_j, contains the two following informations:

– *The k predecessors of* S_i: k is a configurable parameter, indicating how many failures the algorithm can recover by using mechanism **M1**, described below.
– S_i*'s position in the queue*: equals to S_i's closest predecessor's position + 1.

The cost of having a predecessor information mechanism is low in terms of messages. We have added just one message per token request. Thus, the message complexity of the algorithm only grows from $log(N)$ to $log(N) + 1$ and thus remains $\boldsymbol{\mathcal{O}(log(N))}$. However, this mechanism enables the detection of failures more effectively than Naimi-Trehel's fault tolerant extension. In their approach, the reception of the token is controlled by a timer T_{wait}, which depends both on latency (T_{msg}) and the time (T_{cs}) for executing the critical section. In our approach, the same timer depends only on latency (T_{msg}). After receiving a *COMMIT* message, S_i periodically checks the liveness of its closest predecessor.

After detecting a failure, site S_i will start a failure recovery by executing a different mecanism for each of the three following cases:

– **Mechanism 1 (M1).** Site S_i has received a *COMMIT* message and there are less than k consecutive faulty sites in *next queue*.

– **Mechanism 2 (M2).** Site S_i received a *COMMIT* message, but there are more than k consecutive faulty sites in the *next queue*.
– **Mechanism 3 (M3).** The site did not receive any *COMMIT* message.

We now detail how to recover from failures in the three cases. **M1.** When site S_i detects a failure of its closest predecessor, it sends an *ARE_YOU_ALIVE* message to each of its predecessors from the closest to the farthest, so as to check if they are still alive. It stops querying when it obtains an I_AM_ALIVE message from one of its predecessors. The latter then takes S_i as its new successor, i.e. it sets its *next* variable to S_i. The *next queue* is then reconstructed and the order is preserved. Furthermore, the *last tree* remains consistent with the *next queue* and the invariant mentioned in section 3 is asserted.

M2. If no predecessor responded to the *ARE_YOU_ALIVE* message, S_i will try to reconnect itself to *next queue* by diffusing a *SEARCH_PREV* message which contains S_i's position. S_i then arms a timer $(2 * T_{msg})$, waiting for the answer messages. All sites having a smaller position then S_i's will answer to it. After waiting $2 * T_{msg}$, S_i will choose among these sites, the one which has the greatest position to become its closest predecessor. Then, S_i reconnects itself to this chosen site by sending a *CONNECTION* message to it. If S_i does not receive any answer at all after $2 * T_{msg}$, it concludes that it has no predecessors and consequently the token has been lost. S_i should then regenerate the token, initializing its position to zero.

Observe that in both mechanisms **M1** and **M2**, due to our predecessor information approach, the order of *next queue* is preserved.

M3. We must consider now the case where the site which detects the failure has not received the *COMMIT* message yet, and therefore has no position in *next queue*. Moreover, in the absence of such information, several sites can detect the same failure simultaneously.

M3.a We initially consider the situation when just one site S_i detects the failure. In order to reconnect itself to *next queue*, S_i will search for the site which has the greatest position. This search is initiated by the diffusion of a *SEARCH_QUEUE* message. S_i then arms a timer $(2 * T_{msg})$, waiting for the answer messages. A site that has a position in *next queue* answers to S_i with an *ACK_SEARCH_QUEUE* message which contains its position in the *next queue*, as well as whether or not it has a *next*. Among all the received answers within $2 * T_{msg}$, S_i will select the site S_j with the greatest position. S_i then considers three possibilities:

(i): S_j has informed that it has no next. S_i then resends a token request to S_j. Notice that, since this request is sent directly to a node at the tail of *next queue*, S_i does not use *last tree* to send a token request. Thus, we avoid the problem mentioned in section 3 concerning the instability of *last tree* when token requests are in transit.

(ii): S_j has informed that it has a next. S_i can conclude that S_j's *next* has failed. S_i then sends a *CONNECTION* message to S_j in order to force S_j to reconnect itself to S_i; i.e. S_j will set its *next* to S_i.

(iii): If site S_i has not received any answer, it concludes that it has no more predecessors and that the token has been lost. S_i can then regenerate the token, initializing its position to 0. It is sure to be the only site to regenerate the token.

M3.b We now discuss the situation when several sites detect the node failure concurrently. They will start tracking the *next queue*, and will even generate a new token, which may bring *next queue* to an inconsistent state or the loss of the token uniqueness property. An election mechanism then is necessary. We consider that a site is elected if it is always candidate after a time of $2 * T_{msg}$.

Having sent a *SEARCH_QUEUE* message to the other sites as described above, site S_i is a candidate to reconnect to *next queue*. However, if S_i receives a *SEARCH_QUEUE* message from node S_j, it knows that another site S_j is also a candidate for reconnection. Thus, if S_j has made fewer accesses to the critical section than S_i (this information is included in the *SEARCH_QUEUE* message) or S_i's access number is equal to S_j's but S_j has a greater identifier than S_i, the latter loses the election, sending a token request to S_j. In turn, S_j will be responsible for reconnecting itself to *next queue*. If S_j later loses the election, it will behave like S_i. However, if it wins the election, it finds itself in the situation of mechanism **M3.a**. *Next queue* is thus repaired[1].

Contrary to the first two mechanisms, the order of previous token requests is not preserved in mechanism **M3**. Thus, *last tree* must be reconstructed to be consistent with the new *next queue*. However, this reconstruction is done dynamically, without any additional overhead in terms of message and latency, since all the information a site needs has been transmitted to it in the *SEARCH_QUEUE* message. Considering that S_i is the single site that suspected the fault (**M3.a**), or the one that wins the election (**M3.b**), *last tree* is reconstructed as follows:
I: all sites which do not wait for the token set their *last* variable to S_i.
II: all sites that have a position in *next queue* set their *last* variable to S_i.
III: all sites without a position, but in wait for the token, set their *next* variable to the same value as their *last* variable.

An example of failure recovery based on mechanism **M3** is shown in figure 2. We consider that there are two faulty sites, as shown in figure 2.a. The *next queue* is broken into two portions (G,H and A,B,C). Sites G and H have already obtained a position in *next queue*, but sites A, B and C have not. The token is held by site G, first site of the *next queue*. We also consider that site D had sent a token request to one of the two faulty sites and while waiting for the *COMMIT* message, it accepted a token request from E. Thus, there is a second queue of sites waiting for the token, but it is not connected to *next queue* yet. Notice that in such configuration *last tree* is also broken (the *last* variable values of sites F, G, H and D have become useless).

Suppose that sites A and D detect a node failure concurrently. Both of them broadcast a *SEARCH_QUEUE* message. We then say that A wins the election, i.e. A is the elected node. In figure 2.b, we can see how some of the *last* variables are updated. Having a position, sites G and H update their *last* to A (see II), as

[1] To ensure that two sites do not get the same position, message ordering is controlled by using Lamport's timestamp.

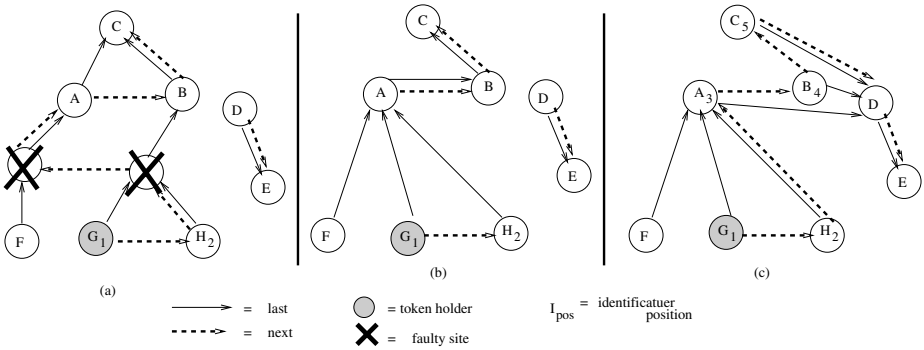

Fig. 2. Example of our fault-tolerant algorithm execution

well as site F, which was not waiting for the token (see I). However, A, B and E, which are waiting for the token but do not have a position, update their *last* variables to the same value as their respective *next* variables (see III).

When receiving the *ACK_SEARCH_QUEUE* message from H, site A can conclude that the *next* of H is a faulty site (see *ii*). A then sends a *CONNECTION* message to H. When the latter receives such a message, it sets its *next* variable to A. On the other hand, since site D lost the election, it sends a token request to the elected node (A). When receiving D's request, A forwards this request to its *last* ($last_A = B$). The request travels along *last tree*, arriving at C, the root of the tree. All sites belonging to *last path* which received the request update their *last* variable to D. Site C sets its *next* variable to D, sending a *COMMIT* message to it. Figure 2.c shows the final configuration, considering that sites D et E have not received a *COMMIT* message yet.

5.2 Sketch of Proof

We are just going to give the outline of the correctness proof of our algorithm. For this purpose, we should prove its *safety* and *liveness* properties:
– **Safety:** there is always at most one token in the system, which guarantees that at most one site can execute the critical section at any time.
– **Liveness:** A site requesting entry to the critical section will eventually succeed within a bounded time.

Proof of liveness: This proof comprises two parts. Firstly, we should prove the *liveness* for a site which has a position, and then that a site eventually obtains a position within a bounded time.

In the absence of failures, we can identify the following four invariants, which are easily proven by induction: **I1** - the site with the smallest position has the token; **I2** - the position ordering respects the order of *next queue*; **I3** - after S_i got its position, no site can get a new position which is smaller than S_i's; **I4** - two sites cannot have the same position.

In the absence of failures, these invariants ensure that a site S_i holding a position will receive the token within a finite time. Indeed, **I1** and **I2** ensure

site S_i that the owner of the token is one of its predecessors while **I2** and **I3** guarantee that a site is never inserted before S_i's predecessors. On the other hand, when a failure occurs, invariant **I1** is not true anymore if the token is lost. Thus, to ensure *liveness*, it is just enough to prove that mechanism **M2** is able to make invariant **I1** true again within a bounded time, i.e. that the site with the smallest position eventually detects the loss of the token and regenerates it within a bounded time. It is not necessary to prove that the three recovery mechanisms need to reconstruct the *next queue*. However, we must prove that they do not change the other invariants.

To prove the second part, i.e. that a site obtains a position within a bounded time, we must prove that:
– *A site whose token request is lost retransmits it within a bounded time using a set of data structures which is consistent with the original algorithm.* The failure detection of mechanism **M3** takes place within at most $N * Tmsg$. Moreover, it is also possible to show a scenario in the absence of failure which is similar to the one resulting from the execution **M3** that rebuilt the *last tree* and the *next queue*.
– *A request can be lost a finite number of times.* This is ensured by our model, i.e. there can be at most $N-1$ permanent crashes. However, if there is an infinite number of failures, this property keeps true if and only if the system has periods of stability of at least $N * T_{msg}$.

Proof of safety: in the original Naimi-Trehel algorithm there is always only one token. However, in our approach, **M2** and **M3** may regenerate a token. Thus we need to prove that in these mechanisms:
– *A site regenerates a token only when the latter has been lost:* in mechanism **M2** (resp. **M3**), a site regenerates a token, if and only if, it did not receive an answer *ACK_SEARCH_PREV* (resp. *ACK_SEARCH_QUEUE*). To prove that "no answer implies no more token", its contrapositive can easily be proven. This can be done by using invariant **I1**, which implies that if there is at least one token, then the site with the smallest position has one of these tokens.
– *Only one site regenerates the token:* we can prove by contradiction that **M2** and **M3** are not compatible. In the same way, we can prove that two sites cannot regenerate the token by using both mechanism **M2** (resp. **M3**).

6 Performance Issues

Considering failure detection and recovery, we can compare our algorithm to Naimi-Trehel's fault-tolerant one (see section 4).
– **Message complexity:** in the worst case of failure recovery, Naimi-Trehel's fault tolerant algorithm broadcasts four messages. Our solution sends one *COMMIT* message per token request in order to keep control of *next queue* and a broadcast *SEARCH_PREV* message, if necessary. It is also worth mentioning that in the former all the successors of the faulty node must resend their token request, while in our algorithm only lost requests are resent.
– **Time:** so as to detect faulty sites, Naimi-Trehel's algorithm controls the reception of the token. In the worst case, it waits $(N-1) * (T_{msg} + T_{cs})$. Our

algorithm controls the arrival of the token request at the tail node of *next queue* as well as the reception of the *COMMIT* message by the requester. It then waits at most $((N-1)+1)*T_{msg}$ for suspecting a faulty node, which does not depend on T_{cs}. Another important point is that our algorithm has fewer phases than Naimi-Trehel's during failure recovery, reducing recovery time. Moreover, it may happen that such a recovery is done while the algorithm goes on executing normally as if no failure had occurred; i.e it does not need to wait for a stable *last tree* as in the Naimi-Trehel approach. The failure recovery time is then covered up by the time that a site waits for a token.

– **Fairness:** in Naimi-Trehel's approach, *next queue* is rebuilt from the beginning at each failure recovery and the original ordering is not preserved. In our approach, after receiving the *COMMIT* message, a site has its position p in *next queue* which ensures that it will access the critical section after at most $p-1$ other critical section accesses.

7 Conclusion

We presented in this paper a new fault-tolerant algorithm for mutual exclusion. This algorithm is an extension of the Naimi-Trehel token-based algorithm. Compared to the solution proposed in [6], our algorithm has two main properties: a short recovery delay and a resilient fairness of requests. In case of failure, we reconstruct the distributed request queue by assembling portions of the previous queue. Our algorithm requires at most one broadcast and the order of critical section requests is preserved as much as possible despite failures.

References

1. M. Bertier, L. Arantes, and P. Sens. Hierarchical token based mutual exclusion algorithms. In *4th IEEE/ACM CCGrid04*, 10 April 2004.
2. L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–564, July 1978.
3. M. Maekawa. A $\sqrt{N}$ algorithm for mutual exclusion in decentralized systems. *ACM Transactions on Computer Systems*, 3(2):145–159, May 1985.
4. Frank Mueller. Fault tolerance for token-based synchronization protocols. *Workshop on Fault-Tolerant Parallel and Distributed Systems, IEEE*, april 2001.
5. 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, 10 April 1996.
6. Mohamed Naimi and Michel Trehel. How to detect a failure and regenerate the token in the log(n) distributed algorithm for mutual exclusion. *Lecture Notes In Computer Science LNCS*, 312:155–166, 1987.
7. K. Raymond. A tree-based algorithm for distributed mutual exclusion. *ACM Transactions on Computer Systems (TOCS)*, 7(1):61–77, 1989.
8. G. Ricart and A. Agrawala. An optimal algorithm for mutual exclusion in computer networks. *CACM: Communications of the ACM*, 24, 1981.
9. I. Suzuki and T. Kasami. A distributed mutual exclusion algorithm. *ACM Transactions on Computer Systems (TOCS)*, 3(4):344–349, 1985.