http://dx.doi.org/10.5392/IJoC.2012.8.4.001

A Fault-tolerant Mutual Exclusion Algorithm in Asynchronous Distributed Systems

Yoon Kim
Dept. of Computer Information and Security
Korea National College of Welfare, Pyeongtaek, Korea

ABSTRACT

Mutual Exclusion is one of the most studied topics in distributed systems where processes communicate by asynchronous message passing. It is often necessary for multiple processes at different sites to access a shared resource or data called a critical section (CS) in distributed systems. A number of algorithms have been proposed to solve the mutual exclusion problem in distributed systems. In this paper, we propose the new algorithm which is modified from Garg's algorithm[1] thus works properly in a fault-tolerant system. In our algorithm, after electing the token generator, the elected process generates a new token based on the information of the myreqlist which is kept by every process and the reqdone which is received during election. Consequently, proposed algorithm tolerates any number of process failures and also does even when only one process is alive.

***Keyword**: Mutual Exclusion, Critical Section, Fault-Tolerant, Fault Detection, timeout, safety, liveness, fairness*

1. INTRODUCTION

Mutual Exclusion has been one of the fundamental issues in distributed systems. It is often necessary for multiple processes at different sites to access a shared resource or data called a critical section (CS) in distributed systems. The problem is to ensure that only one process is allowed to enter the critical section at a time. A number of algorithms have been proposed to solve the mutual exclusion problem in distributed systems. These algorithms are mainly divided into two groups: permission-based and token-based.

In permission-based algorithms, a process may enter the critical section after receiving permissions from all other processes. In token-based algorithms, a unique token (also known as a PRIVILEGE message) is shared among all processes and only one process can enter its critical section if it possesses the token. Examples of permission-based algorithms are Agrawal-Abbadi's algorithm[2], Ricart- Agrawala's algorithm[3], Singhal's algorithm[4] and Maekawa's algorithm[5]. Examples of token-based algorithms are Susuki-Kasami's broadcast algorithm[6], Singhal's heuristic algorithm[7], Mamum's Group algorithm[8] and Raymond's tree based algorithm[9] .

Few fault-tolerant token-based algorithms [10,11,12,13] have been proposed to handle node failures and message loss. Mishra and Srimani algorithm[11] uses the concept of circulating a privilege message (token) among the sites. Nodes use timeout and probe messages to detect failure of the current token holder.

In this paper, we present a new fault-tolerant distributed mutual exclusion extension for Garg's decentralized token-based mutual exclusion algorithm in distributed system which assumes a fault-free system. Garg's Algorithm works properly in fault-free systems. However, it does not tolerate failure of a process, especially a token holding process or a process that has requested the token, which results in the token loss in the entire system.

In this case, a system is in deadlock because any request does not arrive from a process holding the token and thus any process can not enter the critical section. This does not satisfy liveness property of mutual exclusion, that every request for the critical section is eventually granted. We also have implemented the solution of this problem in this paper. Consequently, it is designed to perform properly even when any of the cooperating processes in the system are malfunctioned. That is, the algorithm designed in this paper tolerates any number of process failures even in case that only one process is alive in the system.

2. GARG'S ALGORITHM

To request the critical section, a process increases its component in *v* vector and sends the request with its *v* vector to all the processes. If a process has the token, then it checks for eligible requests and sends the token to one of the eligible processes. On receiving the token, a process is eligible to enter the critical section. To release the critical section, it sends the

* *Corresponding author, Email : ykim@hanrw.ac.kr*
Manuscript received Aug. 10, 2012; revised Sep 17, 2012; accepted Sep 27, 2012

token to the next eligible process or if there's no eligible process, it holds the token until it receives a request. To ensure fairness, all the processes piggyback *v* vector on all outgoing messages.

When a process receives a program message, it updates its old value in *v* vector. To solve the problem that a request broadcast may never reach the token if it was made when the token was in transit between two processes, each process maintains *myreqlist*. Whenever the token is received, it is updated with all the entries in the *myreqlist*. This algorithm requires, at most, N messages per critical section invocation: (N-1) request messages and one token message.

3. PROPOSED ISSUES

We have modified the Garg's algorithm in order to make it also work properly in a fault-tolerant system. In this section, we discuss the problems of Garg's algorithm in terms of process failures and present the changes needed to make the Garg's algorithm fault tolerant to these failures.

3.1 Process Failures

Processes may fail by crashing. Crashed processes nay stop their execution permanently or recover from failures restarting execution. In this paper, we focus on only the former case called fail-stop failures. Since we assume that the communication link is reliable, we do not consider the link failures at this time. In addition, we distinguish crashed processes into the following two cases.

- A process with the token (token holder) fails.
- A process that has requested the token fails.

3.2 Fault Detection

To detect process failures, we use Acknowledgements and timeout-based re- transmission of messages which are commonly used techniques for fault detection in a distributed system. In our algorithm, a process detects process failures in the following cases.

First case is that process does not receive message that it is using the token now from the token holder while waiting for the token periodically. Second case is that process does not receive an acknowledgement from the process that will receive the token after it asks aliveness before sending the token.

3.3 Fault-Tolerance

Our algorithm is fault-tolerant because a process detects above two cases of process failures and takes an action as follows:

1) Process Failure Case 1: When a process detects the failure of the token holder, it starts an election algorithm to elect a token generator. The token generator sends the token after checking the next eligible process.

2) Process Failure Case 2: Before a process sends the token, it asks the receiving process' aliveness. If it does not receive an acknowledgment within a timeout period, it detects the process failure and checks again for the next eligible process to send the token.

4. SPECIFICATION OF FAULT-TOLERANT ALGORITHM

In this section, we propose our own fault tolerant extension to the original Garg's algorithm[1]. We consider a distributed system consisting of N processes denoted by $\{P_1, P_2, \ldots, P_n\}$, which are fully connected, communicating only by message passing. The system is synchronous i.e. both process speed and message transmission times are bounded. Processes can fail only by crashing, and crashes are permanent.

4.1 Basic Idea

The rules of our algorithm are as follows:

To request the critical section, a process increments its component in *v* vector and sends the request with its *v* vector to all the processes.

On receiving a request message, a process updates its old value in *v* vector and appends the request in the list (*myreqlist*). If it has the token and does not use the critical section, then it checks for eligible requests and sends the token to one of the eligible processes. But if it is in the critical section, it sends "token_using" message to the requesting process.

On receiving the token, a process is eligible to enter the critical section and sends back an acknowledgment. To release the critical section, a process checks for eligible requests in the list (*myreqlist*) and sends the token to one of the eligible processes. If there is no eligible process to send the token, the process holds the token.

When a process receives a request, it sends a "token_using" message to the requesting process if it has the token and is in the critical section. The token holding process periodically sends the "token_using" message to the processes in the *myreqlest*.

After a process sends a request, if it does not receive the token and the "token_using" message within a timeout, it suspects token loss due to the token holding process failure. Then, it sends a "token_loss_detect' message to all other processes to initiate an election process. After a process sends the token to the next eligible process, if it does not receive an acknowledgement within a timeout period, it suspects the process failure. Then, it checks the eligible process and sends a newly generated token.

In our algorithm, after electing the token generator, the elected process generates a new token based on the information of the *myreqlist* which is kept by every process and the *reqdone* which is received during election. To ensure fairness, all the processes piggyback *v* vector on all outgoing messages. When a process receives a program message, it updates its old value in *v* vector.

4.2 System Model

The model of our algorithm is an asynchronous system composed of a set of N processes denoted by $P_1, P_2, \ldots, P_n$. The processes are fully connected without any shared memory or a global clock and communicate only through message passing. Further assumptions on the message delay and system failures are as follows:

(1) **Message Delay :** Message delay is unbounded since the algorithm is based on an asynchronous model.
(2) **Communication Links** : Communication links are bidirectional and reliable, that is, no message is lost and changed.
(3) **System Failures** : There are no faults in the system, that is, processes are reliable.
(4)**Token Loss** : Since the processes and communication links are reliable, the token loss does not happen.
(5) **Timeout** : We use timeout as a timer to detect the token loss.

4.3 Data Structures

Each process Pi has variables described in Fig. 1.

v	An array of integers. *v*[i] represents the number of request of Pi. Initially, *v*[i]=0 for Pi (1 = i = N).
myreqlist	A list of (*pid*, *reqvector*). It denotes some request that has not been fulfilled yet. When Pi receives a request, it appends the request to the list. The list is used to update the token.
inCS	A boolean variable which is set to true when Pi is in the critical section.
havetoken	A boolean variable which is set to true when Pi receives the token and is set to false when Pi releases the token. The token has the following variables. Any process holding the token does the computation using these two variables
reqdone	An array of integers. *reqdone*[i] equals the number of requests made by Pi that has been fulfilled. Initially, *reqdone*[i]=0 for Pi (1 = i = N).
reqlist	A list of (*pid*, *reqvector*). It denotes some request that has not been fulfilled yet. For the fault-tolerance, two more variables are used.
timeout	A timer for the detection of token loss.
flag	A string variable which is set to "request/token_using/new_token"

Fig. 1. Data Structures

5. DESCRIPTION OF FAULT-TOLERANT AlGORITHM

In this section, we describe the exact algorithm performed by an arbitrary node *i*. The exact algorithm is shown in Fig. 1.

5.1 Requesting Critical Section

To request the critical section, a process increments its component in *v* vector and sends the request with its *v* vector to all the processes including itself. After then, it sets timeout and flag to "request". If it does not receive the token and "token_using" message within a timeout, it suspects the token loss as denoted by line 1-5 in algorithm of Fig. 2.

5.2 Receiving a request of Critical Section

On receiving a request, a process updates its *v* vector with the maximum. After then, it appends the request to the *myreqlist*. If it has the token and does not use the critical section, then it checks for eligible requests and sends the token to one of the eligible processes. But if it is in the critical section, it sends "token_using" message to the requesting process as denoted by line 6-11

5.3 Receiving a token

On receiving the token, a process sends back an acknowledgment, which is efficient in case of the process failure. It receives the *reqdone* vector with the token. Processes keep *reqdone* updated through token passing. It sets *inCS* and *havetoken* to true. Then, it calls *checkdone*()procedure to delete the fulfilled request from the *myreqlist*. After then, it sends 'token_using" message to all the processes in *myreqlist* as denoted by line 12-17 in algorithm of Fig. 2.

5.4. deleting the fulfilled requests from *myreqlist*

To delete the fulfilled requests from the *myreqlist*, a process uses the *reqdone* vector. A request *w* is considered fulfilled if *w.v* is at most *reqdone* as denoted by line 18-21 in algorithm

5.5 releasing critical section

To release the critical section, a process sets *inCS* to false. Then, it calls *checkreq()* procedure in order to check for eligible requests in the *myreqlist* to send the token as denoted by line 22-24 in algorithm of Fig. 2.

5.6 Checking Eligible Requests in *myreqlist*

To send the token, a process checks for eligible requests in the list (*myreqlist*). A request *w* is eligible if *w.v* is at most *reqdone*, that is, there is no request that happened before *w* and has not been fulfilled yet. Then, it deletes the eligible process from the *myreqlest* and the increments *reqdone* vector of the eligible process. It sends the token to one of the eligible processes and sets *havetoken* to false. If there is no eligible process to send the token, the process holds the token denoted by line 27-37

5.7 Receiving message that token has been occupied

On receiving a "token_using" message, a process keeps flag to "request"and sets timeout. If it does not receive the token and "token_using" message within a timeout, it suspects the token loss. In our algorithm, if a process uses token, it sends a "token_using" message to the requesting process when it receives a request or the token. In addition, the token holding process periodically sends the "token_using" message to the processes in the *myreqlist*. As denoted by line 38-40 in algorithm of Fig 2.

```
Pi::
var
υ: array[1…N] of integer initially ∀ j : υ[j] = 0;
inCS: boolean initially false;
havetoken: boolean initially false except for P0;
myreqlist: list of(pid, reqvector) initially null;
reqdone: array[1…N] of integer initially 0;
timeout: timer initially t;
flag: request/random/new_token of string initially null;
1. To request:
2. υ[i] := υ[i] + 1;
3. send (request, υ) to all processes (including itself);
4. set timeout;
5. flag=”request”;
6. Upon receive(request, u):
7. υ := max(υ, u.υ);
8. append(myrequlist, u);
9. if (havetoken) then
10. if not inCS then checkreq();
11. else send(token_using, u) to the requesting processes;
12. receive(u, token, reqdone):
13. send an acknowledgement to the process which
has sent the token
14. inCS := true;
15. havetoken=true;
16. checkdone();
17. send token_using to all processes in myreqlist
18. checdone():
19. done := {x ∈ myrequlist such that
∀ j: myreqlist[j] ≤ reqdone[j]};
20. if done ≠ {} then
21. delete(myreqlist, x);
22. release:
23. inCS := false;
24. checkreq();
25. receive(u): //program message
26. υ := max(υ, u.υ);
27. checkreq():
28. eligible := {w ∈ myrequlist such that
∀ j: j ≠ w.p : w.υ[j] ≤reqdone[j]};
29. if eligible ≠ {} then
30 w := first(eligible);
31. delete(myreqlist, w);
32. reqdone[w.p] := reqdone[w.p] + 1;
34. send (alive_ask_msg) to Pw.p;
35. set timeout;
36. flag=”alive_ask”;
37. endif
38. Upon receive token_using message:
39. flag = “token_using”;
40. set timeout;;
```

```
41. Upon expire_timeout();
42. if (havetoken==false && flag==”request”) then
43. flag=”start_election”;
44. set timeout;; //wait for “new_token”
45. election();
46. if (flag==”start_election”);
set timeout;;
47. election();
48. if (flag==”alive_ask”);
49. checkreq();
50. Upon receive(election_result, reqdon):
51. Checkreq();
52. send new_token to all processes;
```

Fig 2. Proposed Algorithm

5.8 Actions to be made upon timeout

When timeout period has been elapsed, actions are made according to the value of flag. After sending a request (flag="request"), if Pi does not receive the token within a timeout period and "token_using" message periodically, it suspects the failure of the process with token. Then it starts election process.

After finishing the election process, Pi informs the elected process that it is the token generator. Then, it generates a new token and sends it to the next eligible process and "new_token"message to all processes to inform that a new token has been generated.

After setting flag to "start_election", if Pi does not receive "new_token" message within a timeout period, it suspects the failure again (during election , after election or after generating token) and restarts the election process Those procedures are denoted by line 41-49 in algorithm of Fig. 2.

5.9 Sending newly generated token

If Pi is informed that it is the token generator, then it generates a new token and checks for the eligible process to send the token. In addition, it sends "new_token" message to all processes to inform that a new token has been generated as denoted by line 50-52 in algorithm of Fig 2.

6. PROOF OF CORRECTNESS

In this section we prove that the algorithm ensures these three properties of mutual exclusion.

Theorem 1: At any instant, only one process has permission to use the critical section (safety property).

Proof: Because there is only one token in the system, no two processes hold the token simultaneously. Thus, the only one process holding the token can executes the critical section. □

Theorem 2: Every requesting process eventually executes the critical section (liveness property).

Proof: A token request message of a process reaches other processes in finite time since communication links are reliable and thus message loss does not occur. Any request will be placed in the token *reqlist* in finite time since one of the

processes will eventually have the token and every process maintains the records of all requests. Since there can be at most N-1 requests made before the request of a process, the requesting process will eventually have the opportunity to execute the critical section after all the requests of other processes made before its request have granted.

Theorem 3: Different requests must be granted in the order they are made (fairness property).

Proof: Let R_i be the request from a process P_i and R_j be the request from a process P_j. When Ri is made before R_j, we will prove that for these two requests, P_j enters CS after P_i finishes CS.

In the algorithm (Fig. 1.), to send the token, a token holding process checks for the next eligible request. A request *w* is eligible if there is no request that happened before *w* and has not been fulfilled yet. If every component except P_j's in υ vector of R_j is at most *reqdone*, R_j is the next eligible request. If R_i is not fulfilled yet, P_i's component in υ vector of R_j is not at most *reqdone*.

Therefore, since R_j can not be the next eligible request unless R_i is not fulfilled, P_j can not enter CS before P_i. OnlyP_j can enter CS after P_i finishes CS.

7. SIMULATION AND PERFORMANCE ANALYSYS

We, here, analyse the performance of proposed algorithm and simulate it in order to confirm if the algorithm works well on the real world. For performance analysis, we use simulating model as used in [7]. We use a distributed system with 10-50 processes(nodes). Each process makes request at random time and to make our simulation rather simple, we assume that there are no simultaneous requests from processes in the system at a time. Request arrival pattern at a process(node) is Poisson distribution. We assume that three different types of timeout such as "request/token_using/new_token" has the same value and we set it as 30ms. We also assume arbitrary process failures. For performance analysis, we consider the parameter LTS(latency to stable status) which means the latency from time that a process detect failed process to the time that system reach to the stable status. As showed in Table 1 and Fig.2, the result of the simulation shows us that the system does not be stopped by any kind of failures during execution. According to the type of timeout, latency time to the stable status varies and there is no formal pattern showed.

8. CONCLUSIONS

Table 1. Simulation Results

No. of processes	Type of timeout (30 ms)	Latency to stable status(sec)
10	request	0.91
	token_using	0.84
	new_token	1.27
20	request	1.24
	token_using	1.19
	new_token	1.38
30	request	1.85
	token_using	1.94
	new_token	2.38
40	request	2.18
	token_using	2.03
	new_token	2.35
50	request	3.49
	token_using	3.52
	new_token	3.94

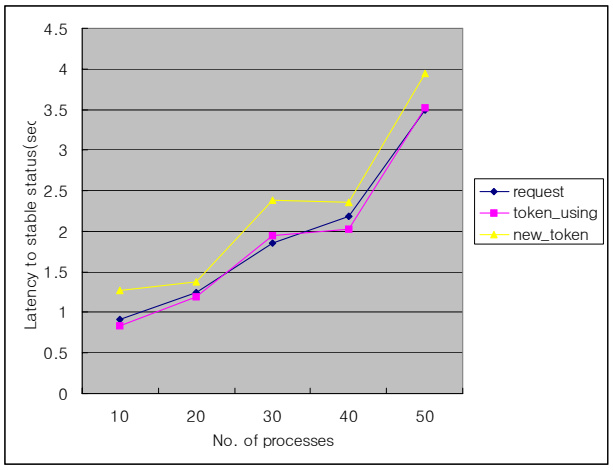

Fig. 2. Simulation results

In this paper, we proposed a new fault-tolerant distributed mutual exclusion algorithm in distributed system which assumes a fault-free system. To detect process failures, we used acknowledgements and timeout based re-transmission of messages which are commonly used techniques for fault detection in a distributed system.

As proved in section 6, proposed algorithm works properly against both node and network failures satisfying three properties of mutual exclusion. We can assume by intuition that proper timeout value in our algorithm will vary according to the system environments, that is, the number of processes, and reliability of nodes and network etc. Future works include simulation and performance analysis which will be expected to give us reference to deciding what are the proper values of timeout in corresponding system.

REFERENCES

[1] Vijay K. Garg, Elements of Distributed Computing, A John Wiley & Sons Inc., Publication, New York, 2002

[2] D. Agrawal and A. El Abbadi, An efficient solution to the distributed mutual exclusion problem, in Proc. 8th ACM Symposium on Principles of Distributed Computing, pp. 193-200, 1989

[3] G.Ricart and A.K.Agrawala, An optimal algorithm for Mutual Exclusion in Computer Networks, Commu nications of the ACM, 1981

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

[5] M. Maekawa, A Square Root N Algorithm for Mutual Exclusion in Decentralized Systems, ACM Transactions on Computer Systems(TOCS), 324):145-159, May 1985

[6] I. Suzuki and T. Kasami,A distributed mutual exclusion algorithm, ACM Transactions on Computer Systems (TOCS), 3(4):344-349, 1985

[7] M. Singhal, A Heuristically Aided Algorithm for Mutual Exclusion in Distributed Systems, IEEE Transactions on Computers, vol. 38, no. 5, pp. 651-662, May 1989

[8] Quazi Ehsanul Kabir Mamun and Hidenori Nakazato, A New Token Based Protocol for Group Mutual Exclusion in Distributed Systems, IEEE Proceedings of the fifth International Symposium on Parallel and Distributed Computing, 2006

[9] K. Raymond, A tree-based algorithm for distributed mutual exclusion, ACM Transactions on Computer Systems (TOCS), 7(1):61-77, 1989

[10] J. Sopena, L Arantes, M Bertier and P Sens. A Fault-Tolerant Token-Based Mutual Exclusion Algorithm Using a Dynamic Tree**.** In *EuroPar 2005, Lisboa, Portugal, September 2005*. *LNCS*, 2005.

[11] S. Mishra and P.K. Srimani Fault-tolerant mutual exclusion algorithms. Journal of Systems Software, 11(2):111-129, February 1990.

[12] M. Singhal, A heuristically-aided algorithm for mutual exclusion in distributed systems, IEEE Transactions on Computers, vol. 38, No.5, May 1989, pp. 651-662.

[13] Chang, M. Singhal, and M. Liu, A fault tolerant algorithm for distributed mutual exclusion, in Proc. of 9th IEEE Syrup. On Reliable Dist. Systems, 1990 pp. 146-154.

Yoon Kim [Regular member]

He received the B.S.,in mechanical engineering from Hanyang university, Korea in 1982 and received M.S. in computer science from Stevens Institute of Technology, USA and received PhD. in computer science from Chungbuk National University. Currently he is a professor of Korea National College of Welfare.