Journal of the Korea Academia-Industrial cooperation Society
Vol. 14, No. 5 pp. 2457-2464, 2013

http://dx.doi.org/10.5762/KAIS.2013.14.5.2457

A Mutual Exclusion Algorithm in Asynchronous Distributed System with Failure Detectors

Sung-Hoon Park[1*]

[1]**Dept. of Computer Engineering, Chungbuk National University**

비동기적 분산시스템에서 고장 추적 장치를 이용한 상호배제 알고리즘 설계

박성훈[1*]

[1]충북대학교 컴퓨터공학과

Abstract In this paper, we design and analyze a mutual exclusion algorithm, based on the Token and Failure detector, in asynchronous distributed systems. A Failure Detector is an independent module that detects and reports crashes of other processes. There are some of advantages in rewriting the Token-based ME algorithm using a Failure Detector. We show that the Token-based ME algorithm, when using Failure Detector, is more effectively implemented than the classic Token-based ME algorithm for synchronous distributed systems.

요 약 본 논문에서는 토큰과 고장추적 장치를 이용한 상호배제 알고리즘을 비동기적인 분산시스템 환경에서 설계하고 분석하고자 한다. 고장추적 장치란 독립된 프로그램으로 다른 프로세스의 크래시 여부를 알려주는 기능을 한다. 이러한 고장추적 장치를 이용하여 토큰기반 ME 알고리즘을 재 작성하는 경우 많은 유익함이 있다. 기존에 동기적인 분산 시스템에서 제안 되었던 토큰기반의 상호배제 (Mutual Exclusion)은 고장 추적 장치를 이용하여 재설계 되는 경우 전통적인 토큰기반 상호배제 알고리즘 보다 훨씬 효율적으로 구현 될 수 있음을 보인다.

Key Words : Synchronous Distributed Systems, Mutual exclusion, Fault Tolerance, Failure Detector.

1. Introduction

Mutual exclusion, simply ME, is an important problem to construct fault-tolerant distributed systems. Depending on a network topology, many kinds of mutual exclusion algorithms have been presented so far. Some algorithms are based on complete topology and others based on ring topology [1-3] or tree topology [5-7]. Among those, as a classic paper, there is the Token-based ME algorithm for synchronous systems based on complete topology specified by Suzuki and Kasami [8]. The mutual exclusion algorithm is used usefully in those systems where a critical section is needed, such as replicated data management, atomic commitment, process monitoring and recovery.

In this paper, we show the Token-based ME algorithm, when using failure detector (FD) [4], is more effectively implemented than the classic Token-based ME algorithm in synchronous distributed systems with crash failures [8]. As the original classic paper, Suzuki and Kasami's Token-based ME algorithm detects a crashed process by time-out intervals, but the modified our Token-based ME algorithm presented in this paper uses a failure-detector instead of the explicit time-out.

This work was supported by the research grant of the Chungbuk National University in 2011
*Corresponding Author : Sung-Hoon Park(Chungbuk National Univ.)
Tel: +82-10-7444-3200 email: spark@cbnu.ac.kr
Received April 15, 2013 Revised (1st May 2, 2013, 2nd May 8, 2013) Accepted May 9, 2013

A failure detector is an independent module that detects and reports crashes of other processes. There are some of advantages in rewriting the Token-based ME algorithm in this way. First, the modularity facilitates use of different failure detection mechanism in different systems. Therefore, implementation of mutual exclusion algorithm is efficient under synchronous distributed systems composed of heterogeneous processes. Second, by checking crashes of other processes concurrently rather than sequentially, execution time of the modified Token-based ME algorithm using failure detector is faster than the classic the Token-based ME algorithm. Especially, in the distributed system where many processes are connected and crash failures occur at many processes repeatedly, the execution time of the modified Token-based ME algorithm with failure detector is much more efficient than the classic Token-based ME algorithm. The rest of this paper is organized as follows: in Section 2, we define a model and definition in a conventional synchronous system. Section 3 describes a solution for the mutual exclusion algorithm with failure detector and analyzes the protocol in terms of the number of messages and times. We conclude in Section 4.

2. Mutual exclusion in a Static System

Our model of asynchronous computation with failure detection is the one described in [9,10]. In the following, we only recall some informal definitions and results that are needed in this paper.

2.1 Suzuki and Kasami's Token-based ME Algorithm

The Token-based ME algorithm, which is Suzuki and Kasami's mutual exclusion algorithm for synchronous distributed system, shows that all the processes in a group can get in a critical section in which every not crashed process agrees to use the critical section.

The Token-based ME Algorithm works as follows. When a process wants to get in the CS (Critical Section), it requests the unique token. A process, *P*, holds a Token as follows:

1. P sends a requesting message for token to the process with a token.
2. If the process with the token has received the message, then it puts the message into its own request queue.
3. If the process does not want to get in CS, it sends its token to the top process of its request queue.
4. The process received the token can get in CS.

At any moment, a process can get a token from one of its colleagues. When a request message arrives, the receiver sends an *OK* message back to the sender to indicate that he has the token and will eventually transfer it to some process who already requested it. The receiver who is the token holder then puts the message into its own request queue. Eventually, every process that wants to get in CS will receive a token and use CS. The token holder announces that it has token continuously sending ok message to other processes.

If a process that holds a token goes down, other process knows that by its failure detector FD. One of process among them begins to generate new token and puts the necessary information into the new token. The process generated new token notifies all process about the new token by sending the ok message.

If it happens to be the process currently waiting a token, it will get the token from the new coordinator and can use the CS.

Thus every process that requested the token always goes in CS even though the lost of a token, hence the name "token-based ME algorithm with Failure Detector".

This algorithm does exactly the function of mutual exclusion in the distributed systems where small number of processes are connected as a group and the frequency of each process's crash failures is relatively low. But in the system where large number of processes are connected as a group and each process's crash failures are frequent, the execution speed for the algorithm will be slow. At the worst case, the system may not generate a token for a long time.

Repetitions of crash failures about processes with token continue to have the system be in the state of the absence of token since a lot of time is spent to detect whether a process with token is crashed. The modified Token-based ME algorithm using a failure detector, which we call Token-based ME_FD algorithm, is faster than the Suzuki and Kasami's one in terms of execution speed. Because it

greatly reduces the time taken for the detection of crashed processes.

2.2 Failure Detector

The failure detector (FD) is an independent module with a function that detects crash and recovery of a process in a system. Whenever the client needs this information, the FD reports this to a client. The FD has an input Request_FD(*i*) which asks the monitoring the process *i*. We illustrate the meaning and usefulness of this with an example. Suppose process *I* crashes and a client asks the FD on process *j* of monitoring process *i* by sending a signal Request_FD(*i*). In this case, the FD on process *j* accepts Request_FD(*i*) as an input from the client and it refers to the down_list which is a list of crashed processes to check whether process *i* is down or not.

If the process *i* is in the down_list, FD informs the client that the process *i* is down by raising the signal <DownSig, *i*>. If it is not in the down_list, FD monitors the process *i* for a few seconds. After that, if FD detects that the process is dead, it adds it to the down_list and informs the client that the process *i* is down by sending the signal <DownSig, *i* >. If it knows that the process is alive, the FD informs the client that the process *i* is alive by sending the signal <UpSig, *i*>. Note that the FD never sends a signal more than once whenever the FD receives the signal Request_FD(*i*). More precisely, after an invocation of Request_FD(*i*), if process *i* is down, then the FD is required to raised <DownSig, *i*> only once regardless of whether the process *i* recovers again after raising <DownSig, *i*> before the most recent invocation of Request_FD(*i*). Furthermore, to ensure that the FD reports up-to-date information, we require that the client receives <DownSig, *i*> only if process *i* is down after the most recent invocation of Request_FD(*i*).

By managing the information about crashed processes as a form of the down process list, the FD can send the information about crashed processes to the client more promptly than the Suzuki and Kasami's one can. When the crashed process recovers again, it sends to the FD on each process immediately the message informing that it has recovered. After receiving the message, if the name of the recovered process exists in the down_list, the FD removes the process's name from the down_list. There are many other methods to implement failure detector. For example, the simplest implementation of failure detector is to send the "Are You Alive?" message to each process being monitored periodically. If a reply is not received in the expected time, FD raises <DownSig, *i*> for the process. A more slightly complicated approach is for each process *i*, when it starts monitoring process *j*, to tell process *j* to periodically send "I'm alive" message to process *i*. This uses fewer messages and reduces the latency of the FD. A more complicate approach, based on an attendance list [11,12], is to a construct logical ring and periodically circulates a token around it. If a process does not see the token within the expected time, then one or more processes are failed, and "Are you alive?" messages can be used to pinpoint the failed processes. With this approach, fewer messages will be used if multiple processes are being monitored by multiple processes, though at the expense of increased detection latency.

3. Token-based ME Algorithm Using FD

Token-based ME algorithm is designed for the system with a few of following properties. As a system environment the synchronous system is assumed, where transmission and processing time of the messages occurring between processes is predicted and information exchanges between processes is done within the given time. A system is based on the fully connected communication networks in which fixed number of processes is inter-connected through them. Processes crash and recover. We do not assume any other kinds of failures such as Byzantine failures. Each process has access to a small amount of stable storage for relevant information that is used for recovering right after the failures. Communications between processes are done as sending messages. Communication is executed as FIFO. We assume also that communications under the synchronous system is reliable.

3.1 Description and algorithm of Token-based ME_FD

We use integers to identify the processes connected on the system and specify the set of processes as formula

(3.1).

$$ID = \{ 1,2,.....,n \} \quad (3.1)$$

where *n* means total number of processes connected on the system and integers identifying processes means parameter which decides priority of them. For simplicity, we use process identifiers as priorities: lower numbers correspond to higher priorities, as in UNIX. That is, the priority of process 1 has the highest and the priority of process 2 is second high and so on. Basic idea of Token-based ME algorithm is that only the process with a unique Token has a right to use the critical section (CS) among all of processes. Each process *i* has a *status* variable, initially having Norm value.

Following is the scenario of the mutual exclusion using FD. [Line 7-9 of Figure 3.1]: The process trying CS periodically ask its failure detector FD about the crash of the token holder process by the module of Request_FD(*token-holder*). [Line 9-12 of Figure 3.1]: When process *i* wants to get in CS, the process sets its *status* variable to Try and indicates that it is in the stage 1 of organizing a trying for CS. In stage 1, process *i* sends request message <Req, i> to the token holder process. [Line 13-20 of Fig. 3]: When a process received a request message such as <Req, j> from the process *j*,if it is in the state of using CS then it put the message <Req, j> into its queue *Rqueus* and sends ok message to the process *j*. If it is not in the state of using CS but only in HaveToken states, then it sends its token to the process *j*. [Line 21-22 of Figure 3.1]: When received a message, <Ok, j> from process *j*, the process i set its status with try and denote the token holder as process *j*. [Line23-25 of Figure 3.1]: When process *i* received a Token and Rqueue from process *j*, it can gets in CS with setting its status with Have-Token.

[Line 26-33 of Figure 3.1]:When process *i* exits from CS, if its request queue is not empty, it sends his token to the first process in request queue and delete the process from the Rqueue setting its status with Norm. If there is no element in the Rqueue, it just sets its status with HaveToken and wait a process to request token. [Line 34-42 of Figure 3.1]:When process *i* received a signal such as <downSig,*j*> from its failure detector FD, if process j is the token holder and its status is in Try, then it sends a message <Ltoken,i> to all process to notify the crash of the token holder process *j*. After that it generates new token and sets its status with Wait to receive the request information of other processes. If process *i* itself is a token holder and process *j* is the token waiting process, it deletes the element of process *j* from the request queue Rqueue.

[Line 43-47 of Figure 3.1]: When process *i* received the token lost message <Ltoken, j> from process *j*, if process *i* is in state of trying CS, it sends <Req2,i> to *j*and sets its token holder process with *j*. [Line 48-50 of Figure 3.1]: When process *i*received a message <Req2,j> from *j* announcing that *j* is in state of trying CS, it puts the message into its request queue Rqueue. [Line 51-58 of Figure 3.1]: On time out with time interval , process *i* checks its request queue and if it is not empty then it sends his token to the first process in request queue and delete the process from the Rqueue setting its status with Norm. If there is no element in the Rqueue, it just sets its status with HaveToken and waits a process to request token.

If the process with token holder is operational, process *i* stays on the wait state in order to give those processes to go in CS a chance to get a token from the token holder. If a process is recovered from the dead state, it waits for the message <Norm?, t> asking the state of recovered process from the token holder. If none of processes waiting a token are operational (i.e., if process *i* receives the message <downSig, *j*> for those processes from FD), then it stays in wait state of waiting a token and sets it s *status* variable to Try.

On wait state, process *i* prepares the token for the processes waiting token and generates a new token by sending them <Ltoken, j> message. When a waiting process receives a OK message from the token holder process, it sends an Ack message and switches its *status* variable from Norm to Try state indicating that it is waiting for the token for the mutual exclusion. If a process on wait state detects the failure of the process which waiting a token by receiving <DownSig, *i*> from FD, it deletes the process from the Rqueue. When the process *i* on wait state which organized the mutual exclusion has received an acknowledgement signal from a process, then it saves the acknowledge into its Rqueue andnotifies the fact that it has saved the request message

by sending OK message. The process received OK message from process *i* accepts process *i* as their a new token holder, switching their status from Wait to Try.

Periodically, the token holder sends the message <Norm?,t> checking status of process to the processes waiting the token in order to find out whether recovered processes exist. The process which has received the message <Norm?,t> sends the message <NotNorm,t> if it is not on Norm state. The token holder which has received message <NotNorm,t> switches its state to Try, and then it does reset the mutual exclusion process again.

The messages described above have a mutual exclusion identifier. We can identify which mutual exclusion the message is part of. An identification tag is a tuple which contains the identifier of the process which starts the mutual exclusion, the process's incarnation number which is kept on stable storage and incremented on each recovery after failure, and a sequence number of mutual exclusion which is incremented for each mutual exclusion. If the Ack or Ok which doesn't contain the expected identifier arrives, the message is ignored. Figure 3.1 depicts re-written Token-based ME Algorithm using FD. It is designed in forms of reactive style, using Upon statement to specify the codes to execute when message or signal is received.

It is specified as the codes executed at intervals using Periodically() statement. Each process is initiated by executing Upon Recovery statement. The variable declaration statement means a variable is stored on the stable storage. The statement send *m* to *j* means message *m* is sent to *j*. Send *m* to *s*, where *s* is set of processes, means message *m*is sent to each process which belongs to the set *s* repeatedly. In the same way, Request_FD(*s*) denotes repeated execution of Request_FD.

```
1.Var status : {Norm, Try, inCS, HaveToken, Wait}
2.Var token-holder : ID
3.Var Rqueue : Queue
4.Var Token: Boolean value
5.Var next_turn : ID
6.Periodically() do
7. if status = Try  token-holder  i then
8.        Request_FD(token-holder) fi od
9.To request (CS) do
10. if status = Norm  token-holder  i then
11.       send <Req, i> to token-holder
12.       status := Try fi od
13. Upon receive <Req, j> from j do
14.   if status = inCS then
15.        put <Req, j> into Rqueue
16.      send <Ok, i> to j
17.   else if status = HaveToken then
18.      send<Token, Rqueue> to j
19.      status := Norm
20.   fi od
21.  Upon receive <Ok, j>  from j do
22.      if status = Try then  token-holder := j fi od
23.  Upon receive <Token, Rqueue> from j do
24.    status := HaveToken
25.    get in (CS)  od
26.Upon exit (CS) do
27.    if Rqueue = not empty then
28.    status := Norm
29.       next_turn:= First (Rqueue)
30.      delete First_element from Rqueue
31.      send <Token, Rqueue> to next_turn
32.       else  status = HaveToken
33.    fi od
34. Upon receive <downSig,j> from FD do
35.   if  j = token-holder  status = Try then
36.     send<Ltoken,i> to all  processes
37.     generate New(token)
38.     status := Wait
39.     wait ()
40.     else if  i = token-holder then
41.   if j Rqueue then delete j from Rqueue fi
42.     od
43. Upon receive <Ltoken,j> from j do
44.   if status = Try then
45.     send <Req2, i> to j
46.     token-holder := j
47.   fi od
48. Upon receive <Req2, j> from j do
49.   if status = Wait then
50.    put <Req2, j> into Rqueue fi od
51.  On time-out() do
52.   if Rqueue = not empty  status = Wait  then
53.     status := Norm
54.     next_turn := First (Rqueue)
55.     delete first_element from Rqueue
```

56. **send**<Token, Rqueue> to *next_turn*
57. **else** *status* = HaveToken
58. **fi od**

[Fig. 3] Bully_FD Algorithm

The significant differences between existing Token-based ME algorithm and Token-based ME_FD algorithm is as follows.

Token-based ME_FD algorithm uses a failure detector rather than explicit time-outs to track failed processes. In the original Token-based ME algorithm, process *i* waits a reply from process *j* to confirm a process's failure. But in Token-based ME_FD algorithm, process *j* is being monitored by process *i*'s FD and process *i* receives either <DownSig,*i*> or <UpSig, *i*> from the failure detector. Note that procedure time-out in the Token-based ME algorithm is, in effect, integrated into the codes of handling <DownSig,*i*> in the Token-based ME_FD algorithm.

In stage 1 of the mutual exclusion, each process checks concurrently rather than sequentially whether the processes with lower priorities is operational. This optimization is independent of the use of a failure detector, but we can take advantage of such techniques using FD but would be awkward to express using Suzuki and Kasami's RPC-style communication primitive.

Each message has an mutual exclusion identifier that identifies the mutual exclusions, so we can avoid confusions incurred from the deferred messages on the network.

We omit re-distribution round on the application level used in existing Token-based ME algorithm. Implementing of Re-distribution round in the Token-based ME_FD algorithm straightforward.

3.2 Analysis of efficiency of the processing time

Let's compare the processing time of Token-based ME_FD algorithm proposed on this chapter with the Suzuki and Kasami's Token-based ME algorithm.

In general, when there is no crash failure among processes there is no difference about the processing time between the original Token-base ME and the modified Token-based ME_FD. Therefore, we consider the case that a crash failure has occurred in the process with token holder. We define the elements that affect the processing time as follows.

N: Total number of process on the system

N_f: The number of failed processes

T_m: Average propagation time per message of a process

T_p: Average message handling time of a process

T_o: Time-out ($T_o > T_e$)

T_e : Average response time from a process

The message delivery subsystem delivers all messages within T_m seconds of the sending of message. A process responses to all messages within T_p seconds of their delivery. Thus, formula (3.2) describes the average response time from a process.

$$T_e = 2\ T_m + T_p \tag{3.2}$$

When the token-holder process fails, a process in Try state, whose identification is *k*, recognizes it and generate new token and the mutual exclusion would be started again. Following formula describes the total processing time in Token-based ME algorithm of Suzuki and Kasami.

$$\begin{aligned} T_{\text{TOKEN-BASED\_ME}} &= (k-1)T_o + [(N_f - k + 1)T_o + (N - N_f)T_e] \\ &= N_f\ T_o + (N - N_f)\ T_e \end{aligned} \tag{3.3}$$

The term ((k-1)* T_o) in formula (3.3) describes the time taken when process *k* detects that the process with token-holder is crash failed. It is the time required on transiting from the state Try to Wait. The term $[(N_f - k + 1)T_o + (N - N_f)T_e]$ describes the time taken on the process *k*'s checking whether the processes with Try state are crashed or not. It is the time taken on transiting from the state Try to Norm. In the same way, total time taken from start mutual exclusion to finish it on executing the Token-based ME_FD algorithm is formulated in formula (3.4).

$$T_{\text{TOKEN-BASED ME\_FD}} = 2NT_s + (pN_fT_p + (1-p)N_fT_o) + (N - N_f)T_e \tag{3.4}$$

In formula (3.4), T_s denotes the time required for a process to send one message to a FD. Consequently, the

term $2*N*T_s$ is the total time taken to transmit messages between process *k* and the FD as one of signal forms. Let's assume that p is the ratio of all failed processes to the failed processes which FD has already known as it is written in its down process list. For instance, if 10 processes have been failed, and FD has written 7 processes in down process list, then the value of p is 0.7. The term $(pN_fT_p+(1-p)\ N_fT_o)$ means the time taken for FD to detect the failure of each process, and the term $(N - N_f)T_e$ signifies the time for FD to confirm the liveliness of the normal processes.

$$T_d=(3.3) - (3.4)=pN_f(T_o - T_p) - 2NT_s$$
$$pN_f(\ T_o - T_p\ ) \qquad (3.5)$$

By using the formula (3.3) and (3.4), the formula (3.5) is induced as below which describes the difference of processing time between Suzuki and Kasami Token-based ME algorithm and Token-based ME_FD algorithm. In the formula (3.5), the value of $2*N*T_s$ is small enough to be negligible. As I mentioned before, it is the time required for the message exchanges to detect the failed processes between the process and the FD. The message exchanges between them are executed almost concurrently rather than sequentially. Definitely $(T_0 - T_p\ ) > 0$ is true and $p*N_f*(T_0 - T_p\ ) > 0$ is also true. Thus, we can make sure that our Token-based ME_FD algorithm is faster than Suzuki and Kasami Token-based ME algorithm in processing time.

4. Concluding Remarks

So far, many algorithms related with mutual exclusion on distributed system are proposed [13,14,15,16,17,18]. Many of them have concentrated on the solution to the problem of self-stabilizing construction of system using timeout interval. The mutual exclusion algorithms based upon timeout interval are clear and simple in terms with semantics in the system where the small number of processes are connected and the frequency of each process's crash and failure is relatively low. But in the distributed system where many heterogeneous processes are connected and the frequency of each process's relatively high, there are some of problems such as prolongation of executing time. The Token-based ME_FD algorithm is same as the classic Token-based ME algorithm in terms with using timeout interval to detect the crashed processes. The difference between two algorithms is that the Token-based ME_FD algorithm uses the FD but the classic Token-based ME algorithm uses timeout interval directly to detect the crashed processes. By doing this, Token-based ME_FD algorithm can detects the crashed processes concurrently rather than sequentially and thus the speed of processing time in the Token-based ME_FD is more enhanced than the classic one. As another advantage, FD is a module so that modularity facilitates use of different failure detection mechanism in different systems.

References

[1] Carole Delporte-Gallet and Hugues Fauconnier: The weakest Failure Detector to Solve certain Fundamental Problems in Distributed computing. In: Proceedings of the ACM Symposium on Principles of Distributed Computing, New York: ACM Press 2004
DOI: http://dx.doi.org/10.1145/1011767.1011818

[2] D. Agrawal and A. E. Abbadi. An efficient and fault-tolerant solution for distributed mutual exclusion. ACM Transactions on Computer Systems, 9(1):1 . 20, February 1991.
DOI: http://dx.doi.org/10.1145/103727.103728

[3] T. D. Chandra, V. Hadzilacos, and S. Toueg. The weakest failure detector for solving consensus. Journal of the ACM, 43(4):685.722, March 1996.
DOI: http://dx.doi.org/10.1145/234533.234549

[4] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. Journal of the ACM, 43(2):225.267, March 1996.
DOI: http://dx.doi.org/10.1145/226643.226647

[5] G. Chockler, D. Malkhi, and M. K. Reiter. Backo. protocols for distributed mutual exclusion and ordering. In Proceedings of the 21st International Conference on Distributed Computing Systems (ICDCS-21), April 2001.

[6] E. W. Dijkstra. Solution of a problem in concurrent programming control. Communications of the ACM, 8(9):569, September 1965.
DOI: http://dx.doi.org/10.1145/365559.365617

[7] M. J. Fischer, N. A. Lynch, and M. S. Paterson.

Impossibility of distributed consensus with one faulty process. Journal of the ACM, 32(3):374.382, April 1985.

[8] I. Suzuki and T. Kasami. A distributed mutual exclusion algorithm. ACM transaction on Computer Systems, 3(4):344-349, November 1985.
DOI: http://dx.doi.org/10.1145/6110.214406

[9] E. Gafni and M. Mitzenmacher. Analysis of timing-based mutual exclusion with random times. SIAM Journal on Computing, 31(3):816.837, 2001.
DOI: http://dx.doi.org/10.1137/S0097539799364912

[10] V. Hadzilacos. A note on group mutual exclusion. In 20th ACM SIGACTSIGOPS Symposium on Principles of Distributed Computing, August 2001.
DOI: http://dx.doi.org/10.1145/383962.383997

[11] Y.-J. Joung. Asynchronous group mutual exclusion. In 17th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing, pages 51.60, June 1998.
DOI: http://dx.doi.org/10.1145/277697.277706

[12] P. Keane and M. Moir. A simple local-spin group mutual exclusion algorithm. IEEE Transactions on Parallel and Distributed Systems, 12(7):673. 685, July 2001.
DOI: http://dx.doi.org/10.1109/71.940743

[13] L. Lamport. A new solution of Dijkstra's concurrent programming problem. Communications of the ACM, 17(8):453.455, August 1974.
DOI: http://dx.doi.org/10.1145/361082.361093

[14] L. Lamport. The mutual exclusion problem. Parts I&II. Journal of the ACM, 33(2):313.348, April 1986.
DOI: http://dx.doi.org/10.1145/5383.5385

[15] S. Lodha and A. D. Kshemkalyan. A fair distributed mutual exclusion algorithm. IEEE Transactions on Parallel and Distributed Systems, 11(6):537. 549, June 2000. 24
DOI: http://dx.doi.org/10.1109/71.862205

[16] N. A. Lynch. Distributed Algorithms. Morgan Kaufmann Publishers, 1996.

[17] M. Maekawa. A√N algorithm for mutual exclusion in decentralized systems. ACM Transactions on Computer Systems, 3(2):145.159, May 1985.
DOI: http://dx.doi.org/10.1145/214438.214445

[18] D. Manivannan and M. Singhal. An efficient fault-tolerant mutual exclusion algorithm for distributed systems. In Proceedings of the ISCA International Conference on Parallel and Distributed Computing Systems, pages 525.530, October 1994.

Sung-Hoon Park [Regular member]

- Feb. 1982 : B.S in economics and statistics from Korea university
- Dec. 1993 : M.S in Computer science from Indiana University, USA
- Dec. 2000 : Ph.D. in computer science and engineering from Korea Univ.
- Sep. 2004 ~ current : Professor in Chungbuk National University, Korea.

<Research Interests>
Distributed System, Mobile Computing and Theory of Computation