한국산학기술학회논문지
Vol. 11, No. 1, pp. 182-189, 2010

Design of Fault-tolerant Mutual Exclusion Protocol in Asynchronous Distributed Systems

Sung-Hoon Park[1*]
[1]**Dept. of Computer Engineering, Chungbuk National University**

비동기적 분산 시스템에서 결함허용 상호 배제 프로토콜의 설계

박성훈[1*]
[1]충북대학교 컴퓨터공학과

Abstract This paper defines the quorum-based fault-tolerant mutual exclusion problem in a message-passing asynchronous system and determines a failure detector to solve the problem. This failure detector, which we call the modal failure detector star, and which we denote by M*, is strictly weaker than the perfect failure detector P but strictly stronger than the eventually perfect failure detector ◇P. The paper shows that at any environment, the problem is solvable with M*.

요 약 본 논문에서는 비동기적 분산시스템에서 고장 추적 장치를 이용한 상호배제의 문제를 서술하고 이러한 문제를 해결하는 가장 약한 고장 추적 장치를 결정하고자 한다. 이를 위해서 M*라고 정의한 modal failure detector star 고장 추적 장치를 정의하고 M*를 이용해서 상호배제 문제는 비동기적 분산 시스템에서 해결 가능함을 보인다. M*는 perfect failure detector P보다 확실히 약하며 eventually perfect failure detector ◇P보다는 강한 고장추적 장치이다. 본 논문에서는 어떤 환경 안에서 이러한 문제가 해결 가능함을 보인다.

Key Words : Mutual Exclusion, Failure Detector, Fault-Tolerant System, Asynchronous Distributed Systems

1. Introduction

1.1 Background

We address the fault-tolerant quorum-based mutual exclusion problem, simply FTQME, in an asynchronous distributed system. In the system, the communication between a pair of processes is by a message-passing primitive, channels are reliable and processes can fail by crashing. In distributed systems, many applications such as replicated data management, directory management and distributed shared memory requires that a shared resource is allocated to a single process at a time. This is called the problem of mutual exclusion [5, 8, 9, 10, 11, 12, 17].

More precisely, the quorum-based mutual exclusion problem specified with two properties: 1) only one process can access a single, indivisible resource at a time, what is called mutual exclusion (ME) 2) The user accessing the resource is said to be in its critical section (CS) and if a correct process (i.e., a process that does not crash) wants to enter its critical section, then it eventually will be in its critical section, what is called progress property (PP), even if some process crashed while in the critical section [2,6].

The problem of mutual exclusion becomes much more complex in distributed systems (as compared to single-computer systems) due to the lack of both a shared memory and a common physical clock and because of unpredictable message delays.

This work was supported by the research grant of the Chungbuk National University in 2008
*Corresponding Author : Sung-Hoon Park(spark@cbnu.ac.kr)
Received December 9, 2009 Revised December 28, 2009 Accepted January 20, 2010

Evidently, the problem cannot be solved deterministically in a crash-prone asynchronous system without any information about failures. There is no way to determine that a process in its CS is crashed or just slow. Clearly, no deterministic algorithm can guarantee fault-tolerant progress and mutual exclusion simultaneously. In this sense, the problem stems from the famous impossibility result that consensus cannot be solved deterministically in an asynchronous system that is subject to even a single crash failure [7].

1.2 Failure Detectors

In this paper, we introduced a modal failure detector M* and showed that the mutual exclusion problem is solvable with it in the environment with majority correct processes. The concept of (unreliable) failure detectors was introduced by Chandra and Toueg [3,4], and they characterized failure detectors by two properties: completeness and accuracy. Based on the properties, they defined several failure detector classes: perfect failure detectors P, weak failure detectors W, eventually weak failure detectors ◇W and so on. In [3] and [4] they studied what is the "weakest" failure detector to solve consensus. They showed that the weakest failure detector to solve consensus with any number of faulty processes is Ω+S and the one with faulty processes bounded by ⌈n/2⌉ (i.e., less than ⌈n/2⌉ faulty processes) is ◇W. After the work of [8], several studies followed. For example, the weakest failure detector for stable leader election is the perfect failure detector P [4], and the one for Terminating Reliable Broadcast is also P [1, 3].

Recently, as the closest one from our work, Guerraoui and Kouznetsov showed a failure detector class for mutual exclusion problems that is different from the above weakest failure detectors. The failure detector called the Trusting failure detector satisfies the three properties, i.e., strong completeness, eventual strong accuracy and trusting accuracy so that it can solve the mutual exclusion problem in asynchronous distributed systems with crash failure. And they used the bakery algorithm to solve the mutual exclusion problem with the trusting failure detector.

1.3 Contributions

How about the quorum-based mutual exclusion problem? More precisely, what is the weakest failure detector to solve the quorum-based mutual exclusion problem? The bakery algorithm is completely different from the quorum-based ME in which the order of getting the critical section is decided based on a ticket order. In contrast to the bakery algorithm, the quorum-based ME algorithm should receive the permissions from all members of a quorum to exclusively use the critical section.

In general, quorum-based mutual exclusion algorithms assume that the system is either a failure-free model [13,14,16,19], or a synchronous model in which (1) if a process crash, it is eventually detected by every correct process and (2) no correct process is suspected before crash [13, 16]: with the conjunction of (1) and (2), the system is assumed to equipped with the capability of the perfect failure detector P [3]. In other words, the perfect failure detector P is sufficient to solve the fault-tolerant quorum-based mutual exclusion problem. But is P necessary? For the answer to the question, we present a modal failure detector star M*, that is a new failure detector we introduce here, which is strictly weaker than P (but strictly stronger than ◇P the eventually perfect failure detector of [3]). We show that the answer is “no” and we can solve the problem using the modal failure detector star M*. Roughly speaking, failure detector M* satisfies (1) eventual strong accuracy and (2) strong completeness together with (3) modal accuracy, i.e., initially, every process is suspected, after that, any process that is once confirmed to be correct is not suspected before crash. If M* suspects the confirmed process again, then the process has crashed. However, M* might suspect temporarily every correct process before confirming it’s alive as well as might not suspect temporarily a crashed process before confirming it’s crash. Intuitively, M* can thus make at least one mistake per every correct process and algorithms using M* are, in terms of a practical distributed system view, more useful than those using P.

We here present the algorithm to show that M* is sufficient to solve fault tolerant quorum-based mutual exclusion and it is inspired by the well-known Grid-based algorithm of Maekawa [11,15-17]: a process that wishes to enter its CS first gets admissions from the one of quorums. M* guarantees that a crash of the process which has been confirmed at least once will be eventually

detected by every correct process in the system.

We show that, in addition to mutual exclusion and progress, our algorithm guarantees also a fairness property, ensuring that any process which wants to get in a CS is eventually granted to access the CS (starvation-freedom property). We do not consider here probabilistic mutual exclusion algorithms [4,7].

1.4 Road Map

The rest of the paper is organized as follows. Section 2 addresses motivations and related works and Section 3 overviews the system model. Section 4 introduces them modal failure detector star M*. Section 5 shows that M* is sufficient to solve the problem, respectively. Section 6 concludes the paper with some practical remarks.

2. Motivations and Related Works

Actually, the main difficulty in solving the mutual exclusion problem in presence of process crashes lies in the detection of crashes. As a way of getting around the impossibility of consensus, Chandra and Toug extended the asynchronous model of computation with unreliable failure detectors and showed in [4] that the FLP impossibility can be circumvented using failure detectors. More precisely, they have shown that consensus can be solved (deterministically) in an asynchronous system augmented with the failure detector $\diamond$S (Eventually Strong) and the assumption of a majority of correct processes. Failure detector $\diamond$S guarantees Strong Completeness, i.e., eventually, every process that crashes is permanently suspected by every process, and Eventual Weak Accuracy, i.e., eventually, some correct process is never suspected. Failure detector $\diamond$S can however make an arbitrary number of mistakes, i.e., false suspicions.

A quorum-base mutual exclusion problem, simply QME, is an agreement problem so that it is impossible to solve in asynchronous distributed systems with crash failures. This stems from the FLP result which mentioning the consensus problem can't be solved in asynchronous systems. Can we also circumvent the impossibility of solving QME using some failure detector? The answer is of course "yes". The Grid-based algorithm of Maekawa [16] solves the QME problem with assuming that it has the capability of the failure detector P (Perfect) in asynchronous distributed systems. This failure detector ensures Strong Completeness (recalled above) and Strong Accuracy, i.e., no process is suspected before it crashes [2]. Failure detector P does never make any mistake and obviously provides more knowledge about failures than $\diamondsuit$S. But it is stated in [7] that Failure detector $\diamondsuit$S cannot solve the ME problem, even if only one process may crash. This means that ME is strictly harder than consensus, i.e., ME requires more knowledge about failures than consensus. An interesting question is then "What is the weakest failure detector for solving the QME problem in asynchronous systems with unreliable failure detectors?" In this paper, as the answer to this question, we show that there is a failure detector that solves QME weaker than the Perfect Failure Detector. This means that the weakest failure detector for QME is not a Perfect Failure Detector P.

3. The Model

We consider in this paper a crash-prone asynchronous message passing system model augmented with the failure detector abstraction [2].

3.1 The Fault-tolerant Quorum-based Mutual Exclusion Problem

We define here the fault-tolerant quorum-based mutual exclusion problem (from now on -FTQME) using the terminology and notations given in [6,13]. Let Π denote a nonempty set of n processes. We associate to every process i $\in \Pi$ a user, ui that can require exclusive access to the critical section. The users can be thought of as application programs. As in [14], every process i $\in \Pi$ and every user ui are modeled as state machines. A process i $\in \Pi$ and the corresponding user ui interact using tryi, criti, exiti and remi actions. The input actions of process i (and outputs of ui) are the tryi action, indicating the wish of ui to enter its CS, and the exiti action, indicating the wish of ui to leave its critical section. The output actions of i (and inputs of ui) are the criti action, granting the access to its critical section, and

the remi action, which tells ui that it can continue its work out of its critical section. A sequence of tryi, criti, exiti and remi actions for the composition (ui, i) is called a well-formed execution if it is a prefix of the cyclically ordered sequence {tryi, criti, exiti, remi}. A user ui is called a well-formed user if it does not violate the cyclic order of actions tryi, criti, exiti, remi, ... A mutual exclusion algorithm defines trying and exit protocols for every process i. We say that the algorithm solves the FTQME problem if, under the assumption that every user is well-formed, any run of the algorithm satisfies the following properties:

Mutual exclusion: No two different processes are in their CS at the same time.

Starvation freedom: Every request for its CS is eventually granted.

Fairness: Different requests must be granted in the order they are made.

Note that Mutual exclusion is a safety property while Starvation freedom is liveness properties. Let Π denote a nonempty set of n processes as defined in the previous section. A coterie C is a set of sets, where each set Q in C is called a quorum. The following conditions hold for quorums in a coterie C under Π [6]:

$\forall Qi \in C : Qi \neq \varnothing \wedge Qi \subseteq U$

Minimality Property : No quorum is a subset of another quorum. $\forall Qi, Qj \in C : Qi \neq Qj : \neg(Qi \subseteq Qj)$

Intersection Property : Every two quorums intersect $\forall Qi,Qj \in C : Qi \cap Qj \neq \varnothing$.

For example, C = {{a, b}, {b, c}} is a coterie under Π = {a, b, c} and Qi = {a, b} is a quorum. The concept of intersecting quorum captures the essence of mutual exclusion in distributed systems. That is, process i executes its CS only after it has locked all the processes in a quorum Qj $\in$C in exclusive mode. To do this, process i sends request messages to all the processes in Qj. On receipt of the request message, the process j of the quorum Qj immediately sends a reply message to i (indicating j has been locked by i) only if j is not locked by some other process at that time. The process i can access the CS only after receiving permission (i.e., reply messages) from all the processes in the quorum P. After having finished the CS execution, i sends release messages to all the processes in the quorum Qj to unlock them. Since any pair of quorums have at least one process in common (by the Intersection Property), mutual exclusion is guaranteed. The Minimality Property is not necessary for correctness, but it is useful for efficiency.

4. The Modal Failure Detector Star M*

Each module of failure detector M* outputs a subset of the range 2^{Π}. Initially, every process is suspected. However, if any process is once confirmed to be correct by any correct process, then the confirmed process id is removed from the failure detector list of M*. If the confirmed process is suspected again, the suspected process id is inserted into the failure detector list of M*. The most important property of M*, denoted by model Accuracy, is that a process that was once confirmed to be correct is not suspected before crash. Let HM be any history of such a failure detector M*. Then HM(i,t) represents the set of processes that process i suspects at time t. For each failure pattern F, M(F) is defined by the set of all failure detector histories HM that satisfy the following properties:

· Strong Completeness: There is a time after which every process that crashes is permanently suspected by every correct process:

- $\forall i,j \in \Pi, \forall i \in$ correct(F), $\forall j \in F(t), \exists t'' : \forall t' > t'', j \in H(i, t')$.

· Eventual Strong Accuracy: There is a time after which every correct process is never suspected by any correct process. More precisely:

- $\forall i,j \in \Pi, \forall i \in$ correct(F), $\exists t : \forall t' > t, \forall j \in$ correct(F), $j \notin H(i, t')$.

· Modal Accuracy: Initially, every process is suspected. After that, any process that is once confirmed to be correct is not suspected before crash. More precisely: - $\forall i,j \in \Pi : j \in H(i,t0), t0 < t < t', j \notin H(i,t) \wedge j \in \Pi - F(t') \Rightarrow j \notin H(i, t')$

Note that model Accuracy does not require that failure detector M* keeps the Strong Accuracy property over every process all the time t. However, it only requires that failure detector M* never makes a mistake before crash about the process that was confirmed at least once to be correct.

If process M* outputs some crashed processes, then M* accurately knows that they have crashed, since they

had already been confirmed to be correct before crash. However, concerning those processes that had never been confirmed, M* does not necessarily know whether they crashed (or which processes crashed).

5. Solving FTQME Problem with M*

We give in Figure 1 an algorithm solving FTQME using M* in any environment where at least one quorum is available. The algorithm uses the fact that eventual strong accuracy property of M*. More precisely, with such a property of M* and the assumption of at least one quorum being available, we can implement our algorithm of Figure 1. Note here that we don't consider the dead lock situation where two or more processes concurrently trying to obtain permissions from each number of quorums but only get in infinitely waiting. In this algorithm, we assume that there is a mechanism to resolve the dead lock.

```
Var status: {rem,try,incs,wait}initially rem
Var my_token: initially true
Var my_token_holder: initially NULL
Var token :initiallyempty list
Var my_quorum_i: initially empty

Periodically(t) do
    request M* for H_M

1.  Upon received (trying,upper_layer)
2.  if not (status = try)then
3.  wait until ∃Q_k : ∀j ∈Q_k : j∉ H_M
4.  status_i:=try
5.  send (ask_permit,i)to∀j ∈ Q_k
6.  my_quorum:= Q_k

7.  Upon received (ok_pemit,j)
8.    token:=token ∪ { j }
9.    If my_quorum=token then
10.       enter CS

11.   OnExitCS
12.     send(return_permit,i)to
          ∀j∈ my_quorum
        status:= rem

13. Upon received (no_permit,j)
14.   send(return_permit,i) to ∀i ∈ token
15.   token: = ∅
16.   goto try_i

17. Upon received (ask_permit,j)
18.   wait until j ∉ H_M
19.   if my_token=true then
20.     send(ok_permit,j)
21.     my_token_holder:=j
22.     my_token:=false
23.         else
24.     send(no_permit,j)

25. Upon received H_M from M_i
26.   if (my_token = false
              ∧my_token_holder∈H_M) then
27.           my_token:= true

28. Upon received (return_permit,j)
29.   my_token:=true
30.   my_token_holder:=NULL
```

[Figure 1] FTQME algorithm using M^* : process i.

We give in Figure 1 an algorithm solving FTQME using M* in any environment E with any number of correct processes (f < n).

Our algorithm of Figure 1 assumes:

- Each process i has access to the output of its modal failure detector module Mi*;
- At least one quorum is available;
- Each process i is well-formed;
- A dead lock resolving mechanism is installed;

In our algorithm of Figure 1, each process i has the following variables:

1. A variable status, initially rem, represents one of the following states {rem,try,incs,wait};
2. A boolean my_tokeni, initially true, indicating whether i has the its token;
3. A variable my_token_holderi, initially NULL, which denotes the token holder when i send its token to other node;

4. A list token_listi, initially empty, keeping the tokens that i has received from each member of a quorum.

Description of [Line 1-6] in Figure 1; the idea of our algorithm is inspired by the well-known Quorum-based ME algorithm of Maekawa [11, 12]. That is, the processes that wish to enter their CS first wait for a quorum whose members are all alive based on the information HM

from its failure detector M*. Those processes eventually find out the quorum by the eventual strong accuracy property of M* in line 3 of Figure 1 and then sets its status to "try", meaning that it is try to get in CS. It sets the variable my_quorum with Qk and send the message "(ask_permit,i)" to all nodes in the quorum.

Description of [Line 7-8] in Figure 1; the candidate asking for a permission to proceed from every process of one quorum does not take steps until the all permissions are received from the quorum. But it eventually received all permissions from a quorum and enter the CS due to the assumption of installed dead lock resolving mechanism in the system and it get in CS.

Description of [11-12] in Figure 1; On exit from the CS, the node sends "return_permit" to the every member of the quorum from which it received permissions. It set its status with "rem" meaning that it is in normal mode. Notice that no candidate i can be served if other process j is accessing the resource. That is because while other process j is serving but not yet releasing the resource, the candidate i can not obtain all permission from the quorum (line 13 in Figure 1).

Description of [13-16] in Figure 1; If the candidate received the message "no_permit" from the a node of quorum, it returns all received permissions from the quorum to every member of the quorum and after that it try again.

Description of [17-24] in Figure 1; The node i, received "ask_permit" from node j, first checks that j is alive and if it is alive then the node i sends its "ok_permit" to the node i when it has its token. But if the node i has no token, it send the message "no_permit" to the node j.

Description of [25-27] in Figure 1; When the node i received the failure detector history HM from M*, if it knows that a node holding its token died, it regenerates its token again.

Description of [28-30] in Figure 1; Upon received "return_permit" from node j, node i sets its "my_token" with true meaning that it has its token.

Now we prove the correctness of the algorithm of Figure 1 in terms of two properties : mutual exclusion and progress . Let R be an arbitrary run of the algorithm for some failure pattern $F \in E$ ($f < n$). Therefore we prove Lemma 1 and 2 for R respectively.

Lemma 1. (mutual exclusion property) No two different processes are in their CSs at the same time.

Proof: By contradiction, assume that i and j ($i \neq j$) are in their CSs at time t'. According to the line 7-9 of the algorithm 1, no process enters its CS before receiving permissions from a quorum. Thus i must have received all permissions from each member of a quorum and j must have received all permissions from each member of a quorum before t'.

Without loss of generality, assume the event that i received all permission from a quorum precedes the event that j received all permission from other quorum. That is, at some time t'' < t', j received all permissions from a quorum while i is entering CS but before exits from CS. That means that at some time t'' < t', j passed the (my_quorum = token) clause in line 9 while i is still in CS. Thus, one of the following events occurred before t'' at every member of a quorum:

(1) Every member of quorum j has a token and sends (Ok_Permit, j): by the algorithm of Figure 1. But by intersection property of quorum, i is in the CS at t' > t'' and at least one member of the quorum does not have a token: a contradiction.

(2) Every member of quorum j received HM from Mj and $i \in$ HM by the algorithm of Figure 1, at some time t'' < t'. Thus, we can assume that the following is true: $i \notin$ HM at time t'and $i \in$ HM at time t''. By the model accuracy property of M, i is crashed at t''. But it is in the CS at t' > t'' : a contradiction. Hence, mutual exclusion is guaranteed.

Lemma 2. If a correct process request for the CS, then at some time later the process eventually enters in its CS.

Proof: Assume that a correct process i volunteers at time t', and no correct process is ever in its CS after t'. According to the algorithm, after t', process i never reaches line 9 of the algorithm. In other words, i is blocked at some wait clause. The first wait clause (line 3

in Figure 1) is not able to block the process, due to the modal accuracy (1) property of M* and the fact that (n>f) processes are correct. Thus, eventually, statusi = try, and wait clause in line 5-6 in Figure 1 cannot block the process neither. Thus, i issues send (ask_permit, i). The second received clause (more precisely, the statement in line 17 in Figure 1) is not blocking neither, because of the guarantee that any send message is eventually delivered by every correct process. Thus, i is blocked in the third clause (line 7-8 in Figure 1) while processing some token := token ∪ { j }. We show that if a correct process i is blocked while processing some token from j, then process j is blocked and it never sends (ok_permit, j) nor (no_permit, j). But j is never blocked since it is always in one of two states, i.e., my_token is true or not. So contradiction.

Theorem 1: The algorithm of Figure 1 solves FTQME using M*, in any environment E with f < n/2, combining with two lemmas 1 and 2.

6. Concluding Remark

Is it beneficial in practice to use a mutual exclusion algorithm based on M*, instead of a traditional algorithm assuming P? The answer is “yes”. Indeed, if we translate the very fact of not trusting a correct process into a mistake, then M* clearly tolerates mistakes whereas P does not. More precisely, M* is allowed to make up to n2 mistakes (up to n mistakes for each module Mi, i∈ Π). As a result, M*’s implementation has certain advantages comparing to P’s (given synchrony assumptions). For example, in a possible implementation of M*, every process i can gradually increase the timeout corresponding to a heart-beat message sent to a process j until a response from j is received. Thus, every such timeout can be flexibly adapted to the current network conditions. In contrast, P does not allow this kind of “fine-tuning” of timeout: there exists a maximal possible timeout, such that i starts suspecting j as soon as timeout exceeds. In order to minimize the probability of mistakes, it is normally chosen sufficiently large, and the choice is based on some a priori assumptions about current network conditions. This might exclude some remote sites from the group and violate the properties of the failure detector. Thus, we can implement M* in a more effective manner, and an algorithm that solves FTQME using M* exhibits a smaller probability to violate the requirements of the problem, than one using P, i.e., the use of M* provides more resilience.

Acknowledgment

This research was financially supported by the Ministry of Education, Science Technology (MEST) and Korea Industrial Technology Foundation (KOTEF) through the Human Resource Training Project for Regional Innovation.

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

[2] D. Agrawal and A. E. Abbadi. An e.cient and fault-tolerant solution for distributed mutual exclusion. ACM Transactions on Computer Systems, 9(1):1 . 20, February 1991.

[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.

[4] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. Journal of the ACM, 43(2):225.267, March 1996.

[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.

[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] E. Gafni and M. Mitzenmacher. Analysis of timing-based mutual exclusion with random times. SIAM Journal on Computing, 31(3):816.837, 2001.

[9] V. Hadzilacos. A note on group mutual exclusion. In 20th ACM SIGACTSIGOPS Symposium on Principles of Distributed Computing, August 2001.

[10] Y.-J. Joung. Asynchronous group mutual exclusion. In 17th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing, pages 51.60, June 1998.

[11] 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.

[12] L. Lamport. A new solution of Dijkstra's concurrent programming problem. Communications of the ACM, 17(8):453.455, August 1974.

[13] L. Lamport. The mutual exclusion problem. Parts I&II. Journal of the ACM, 33(2):313.348, April 1986.

[14] 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

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

[16] M. Maekawa. A√N algorithm for mutual exclusion in decentralized systems. ACM Transactions on Computer Systems, 3(2):145.159, May 1985.

[17] D. Manivannan and M. Singhal. An e.cient 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 university
- Sep. 2004. ~ current : Professor in Chungbuk National University, Korea.

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