분산 이동 시스템에서 선출 프로토콜의 설계

Design of an Leader Election Protocol in Mobile Ad Hoc Distributed Systems

박성훈
충북대학교 전기전자컴퓨터공학부

Sung-Hoon Park(spark@cbnu.ac.kr)

요약

선출 프로토콜은 프로세스들의 그룹 통신, 데이터베이스의 원자성 완료와 복제된 데이터의 관리 등의 조정자(coordinator)가 이용 될 수 있는 많은 실질적인 문제를 해결하는데 하나의 기본적인 구성 요소로 이용 될 수 있다. 이 문제는 여러 연구단체에서 포괄적으로 연구 되어왔던 바, 이렇게 주된 연구 관심 영역이 된 하나의 이유는 많은 분산 프로토콜들이 하나의 선출 프로토콜을 필요로 하기 때문이다. 그러나 이러한 유용성에도 불구하고, 우리가 알기에는 이동 분산 컴퓨팅 환경에서 이러한 문제를 다룬 연구는 아직 없었다. 이동 분산 시스템은 기존의 분산 시스템 보다 훨씬 더 실패(failure)의 가능성이 높다. 그러한 환경에서 다수의 움직이는 노드들(nodes)로부터 선출의 문제를 해결하는 것은 움직이는 노드의 많은 실패(failure)에도 불구하고 하나의 모바일 노드가 우선순위에 의하여 리더로 선출 될 수 있도록 하는 것이다. 본 논문에서는 이동 분산 컴퓨팅 시스템에서 선출 문제에 대한 하나의 해결 방안을 제시 한다. 이 해결 방안은 Group Membership Detection 알고리즘에 바탕을 두고 있다.

■ 중심어 : | 이동 분산 컴퓨팅 | 동기적 분산 시스템 | 리더 선출 | 고장 감내 시스템 |

Abstract

The Election paradigm can be used as a building block in many practical problems such as group communication, atomic commit and replicated data management where a protocol coordinator might be useful. The problem has been widely studied in the research community since one reason for this wide interest is that many distributed protocols need an election protocol. However, despite its usefulness, to our knowledge there is no work that has been devoted to this problem in a mobile ad hoc computing environment. Mobile ad hoc systems are more prone to failures than conventional distributed systems. Solving election in such an environment requires from a set of mobile nodes to choose a unique node as a leader based on its priority despite failures or disconnections of mobile nodes. In this paper, we describe a solution to the election problem from mobile ad hoc computing systems. This solution is based on the Group Membership Detection algorithm.

■ keyword : | Election Algorithm | Failure Detector | Concurrency Control | Distributed Systems |

* 본 논문은 2007학년도 충북대학교 학술연구지원 사업에 연구비지원에 의하여 연구되었습니다.
접수번호 : #081105-001 심사완료일 : 2008년 12월 04일
접수일자 : 2008년 11월 05일 교신저자 : 박성훈, e-mail : spark@cbnu.ac.kr

1. Introduction

In recent years, several paradigms have been identified to simplify the design of fault-tolerant distributed applications in a conventional static system. Election is among the most noticeable, particularly since it is closely related to group communication [7], which (among other uses) provides a powerful basis for implementing active replications. The Election problem [1] requires that a unique coordinator be elected from a given set of processes.

The problem has been widely studied in the research community[2-6] since one reason for this wide interest is that many distributed protocols need an election protocol. However, despite its usefulness, to our knowledge there is no work that has been devoted to this problem in a mobile ad hoc computing environment. When nodes are mobile, topologies can change and nodes may dynamically join/leave a network. In such networks, leader election can occur frequently, making it a particularly critical component of system operation.

Mobile ad hoc systems are more often subject to environmental adversities which can cause loss of messages or data [8]. In particular, a mobile node can fail or disconnect from the rest of the network. Designing fault-tolerant distributed applications in such an environment is a complex endeavor. Leader election algorithms for mobile ad hoc networks have been proposed in [9][10]. As noted earlier, we are interested in an extrema-finding algorithm, because it is desirable to elect a leader with some system-related attributes such as maximum battery life or maximum computation power. The algorithms in [9] are not extrema-finding and cannot be extended to perform extrema finding. Although, extrema-finding leader election algorithms for mobile ad hoc networks have been proposed in [10], these algorithms are unrealistic as they require nodes to meet and exchange information in order to elect a leader and are not well-suited to the applications discussed earlier. Several clustering algorithms have been proposed for mobile networks [11][12], but these algorithms elect cluster-heads only within their single hop neighborhood.

The aim of this paper is to propose a solution to the election problem in a specific ad hoc mobile computing environment. This solution is based on the group membership detection algorithm that is a classical one for synchronous distributed systems. The rest of this paper is organized as follows: Section 2 describes the mobile system model we use. In Section 3, a solution to the election problem in a conventional synchronous system is presented. A protocol to solve the election problem in a mobile ad hoc computing system is presented in Section 4. We conclude in Section 5.

2. Model and Definitions

Before developing a leader election algorithm for ad-hoc computing environments, we first define our system model based upon assumptions and goals. We model an ad hoc network as an undirected graph, i.e., $G = (V, E)$, where vertices V correspond to set of mobile nodes $\{1, 2, \cdots, n\}$ ($n > 1$) with unique identifiers and edges E between a pair of nodes represent the fact that the two nodes are within each other's transmission radii and, hence, can directly communicate with one another that changes over time as nodes move. Each process i has a variable N_i, which indicates the neighboring nodes, with that i can *directly* communicate the neighboring nodes. We assume that every communication channel is bidirectional; $j \in N_i$ iff $i \in N_j$. More precisely, in the network $G = (V, E)$, we can define E such that for all $i \in V$, $(i, j) \in E$ if and only if $i \in N_j$. The graph can become disconnected if the network is partitioned due to node movement. Because the nodes may change their location, N_i may be dynamically changed and so may G accordingly. We make the following assumptions about the nodes and system architecture:

- Each node has a weight value W_i associated with it. The value of a node indicates its "priority" as a leader of the system and can be calculated upon some criteria such as the node's battery power, the position where the node's distance from other nodes is minimal, computational capabilities etc.
- All nodes have unique identifiers. They are used to identify participants during the election process. Node IDs are used to break ties among nodes which have the same value.
- Links are bidirectional and FIFO, i.e. messages are delivered in order over a link between two neighbors.
- Node mobility may result in arbitrary topology changes including network partitioning and merging. Furthermore, nodes can crash arbitrarily at any time and can come back up again at any time.
- A message delivery is guaranteed only when the sender and the receiver remain connected (not partitioned) for the entire duration of message transfer. Each node has a sufficiently large receive buffer to avoid buffer overflow at any point in its lifetime.

The objective of our leader election algorithm is to ensure that after a finite number of topology changes, *eventually* each node i has a leader which is the most-valued-node from among all nodes in the connected component to which i belongs.

3. Leader Election Algorithm in a Static Network

In this section, we describe a leader election algorithm based on group membership detection algorithm, simply GMDA, by diffusing computations. In later sections, we will discuss in detail how this algorithm can be adapted to a mobile setting.

3.1 Leader Election in a Static Network

We first describe our election algorithm in the environment of a static network, where we assume that nodes and links never fail. The algorithm consists of two phases operated at the node that initiates the election algorithm. 1) Scattering phase - it operates by first "scattering the election message" and 2) Gathering phase - it operates by then "gathering the id of each node" that is connected to the static networks. We refer to this computation-initiating node as the *source node.* As we will see, after gathering all nodes' ids completely, the source node will have the information enough to determine the most-valued-node and will then broadcast its identity to the rest of the nodes in the network. The algorithm uses three messages, i.e., *Election*, *Ack* and *Leader*.

1) Scattering phase. *Election* messages are used to initiate the election by "scattering" the election message. When election is triggered at a source node *s* (for instance, upon crash or departure of its current leader), the node makes a waiting list *wl* and a received list *rl* and begins a *diffusing computation* by sending an *Election* message to all of its immediate neighbors. Initially the waiting list consists of only its immediate neighboring node's ids and the received list consists of an empty list. Every node i other than the source propagates the *Election* message to all of its neighboring nodes except the node from which it first received an *Election* message.

When node i receives an *Election* message from the neighboring node for the first time, it immediately sends the *Ack* message to the source node. The *Ack* message sent by node *i* to the source node contains the ids of all its neighboring nodes that is needed for the source node to elect a leader.

2) Gathering phase. When the source node receives the *Ack* message from the node *j*, it removes *j* from the waiting list and puts *j* into the received list and immediately checks one by one the every node id contained in the *Ack* message. If there is any id in the *Ack* which has already been acknowledged, i.e. that means it is in the received list, it is discarded. Otherwise, it is put into the waiting list of source node and the source node waits the *Ack* message from it. The waiting list is growing and shrinking repeatedly based on the received *Ack* messages, but the received list steadily growing by receiving the *Ack* messages. But eventually the waiting list could be empty and the received list could include all ids of nodes connected to the networks when the source node received the *Ack* messages from all other nodes. Hence the source node eventually has sufficient information to determine the most-valued -node in the received list, because the waiting list could be eventually empty and it means that the source node has received the *Ack* messages from all the nodes.

3) Completing Phase. Once the source node has received *Ack*s from all other nodes, it determines the most-valued-node as a leader among the received list and broadcasts a *Leader* message to all other nodes announcing the identity of the leader.

We illustrate a sample execution of the algorithm. We describe the algorithm in a somewhat synchronous manner even though all the activities are in fact asynchronous. Consider the network shown in [Figure 1(a)]. In this figure, and for the rest of the paper, thin arrows indicate the direction of flow of *Election* messages and dotted arrows indicate the direction of flow of *Ack* messages to the source node. The number adjacent to each node in [Figure 1(a)] represents its value. As shown in [Figure 1], node A is a source node that initializes wl_a and rl_b with {B,C} and {A} respectively and starts a diffusing computation by sending out *Election* messages (denoted as "E" in the figure) to its immediate neighbors, viz. nodes B and C, shown in [Figure 1(a)].

As indicated in [Figure 1(b)], nodes *B* and *C* in turn propagate the *Election* message to its immediate neighbors only except the source node and send the *Ack* message with neighboring node list to the source node *A*. Hence *B* and *C* also send *Election* messages to one another. But the *Election* messages are not acknowledged to the source node since nodes *B* and *C* have already received *Election* messages from the source node respectively. The information about neighboring node is piggybacked upon the *Ack* message sent by each node. Upon received Ack messages from B and C, node A updates wl_a = { B,C }, rl_b = { A } with the neighboring node information piggybacked on the Ack messages.

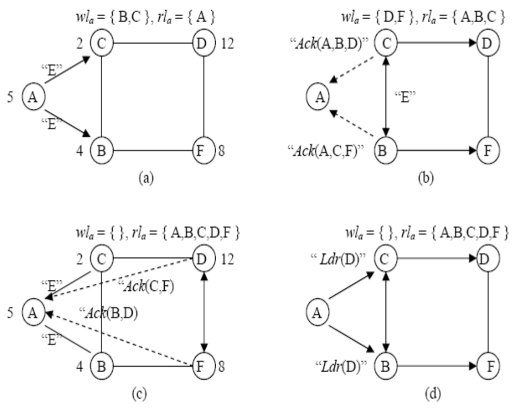


Figure 1. An execution of leader election algorithm based on the group membership detection algorithm. Arrows on the edges indicate transmitted election messages, while dotted arrows parallel to the edges indicate Ack messages.

In [Figure 1(c)], the node D and F also send the *Ack* messages to the sources node when they received the Election messages from the B and C respectively. Each of these *Ack* messages contains the identities of the neighbor and its actual value. Eventually, the source *A* hears all acknowledgments from all of other nodes except itself in [Figure 1(d)] and then decides the most-valued node among them and broadcasts the identity of the leader, D, via the *Ldr* message shown in [Figure 1(d)].

4. Leader Election in a Mobile, Ad Hoc Network

In this section, we redesign the leader election algorithm presented above and describe the operation of the leader election algorithm in the context of a mobile, ad hoc network. In the previous section, we described the leader election algorithm in a static network. But with the node mobility, node crashes, link failures, network partitions and merging of partitions, the simple LE algorithm presented in the previous section is inadequate. Furthermore, we assumed in the previous section that only single node knows as an external input the leader crash or failure, departure and it initiates the election protocol. In reality, such an assumption is inadequate, because many nodes concurrently can receive such inputs and each of them starts a leader election protocol independently. It results from the lack of knowledge of other computations that have been started by other nodes. We assume that the value of the node is the same as its identifier. This assumption has been made only for simplicity of presentation without loss of generality.

Before we formally specify our algorithm and describe it in detail, we briefly introduce notation used in our algorithm specification and the execution model.

4.1 Algorithm Performed By the Nodes

In this clause, we describe the exact algorithm performed by an arbitrary node *i*. The exact algorithm is shown in [Figure 2]. The *LeaderElection* module on every node loops forever and on each iteration checks if any of the actions in the algorithm specification are enabled, executing at least one enabled action on every loop iteration. The bootstrapping of election module involves assigning values variables in line 1-5 of [fig. 2] as specified in the initialization part of the *LeaderElection* module.

```
1. $num_i$:= 0; $ldr_i$ := null;
2. $status_i$ : = Norm; one of states in {Norm, Elect,
 Wait}

3. $n_i$ := {set of all neighboring processes};
4. $cl_i$ := { $i$ };  $wl_i$ := { };
5. $e\_num$ : = null;  $k$ : = null;

6. On $status_i$ = Norm:
7.   if no_signal from $ldr_i$ then
8.      $status_i$ := Elect;
9.      $mum_i$ := $mum_i$ +1;
10.     send $election(mum_i)$ to each process
             of $n_i$ end-if
11. Upon received $election(m)$ from process $j$:
12.       $status_i$ = Wait;
13.       $e\_num$ : = $m$; $k$ : = $j$;
14.       send $election(m)$ to each process
15.              of $n_i$ except $j$:
16.       send $ack(n_i)$ to processes $j$

17. On $status_i$ = Elect :
```

```
18.   Upon received ack (q) from process j :
19.     $wl_i$ : = $wl_i$ - { j }
20.     $cl_i$ := $cl_i \cup$ { j }
21.     $wl_i$ := $wl_i \cup$ { q - { q $\cap$ $cl_i$ } }
22.     if $wl_i$ = empty then checklist();
            end-if
23.  Upon received election(r) from process j:
24.     if { ($mum_i$, i) < (r, j) } then
25.       send election(r) to each process of $n_i$
26.       send ack($n_i$) to processes j
27.       e_num : = r; k : = j
28.       $status_i$ := Wait; end-if

29. On status = Wait :
30.   Upon received leader(t) from process j:
31.       $ldr_i$ := t ;
32.       send leader($ldr_i$) to each process of
              $n_i$ except j ;
33.       $status_i$ := Norm;
34.  Upon received election(r) from process j:
35.    if { (e_mum, k) < (r, j) } then
36.      send election(r) to each process of $n_i$;
37.      send ack($n_i$) to processes j
38.      e_num : = r k : = j
39.      end-if

40. Checklist() :
41.   $ldr_i$ := max ($cl_i$);
42.   send leader ($ldr_i$) to each process of $n_i$;
43.   $status_i$:= Norm;
```

Figure 2. A leader election algorithm in mobile ad hoc computing environments based on the group membership detection algorithm.

1) Initiate Election

The leader of a connected component periodically sends a heartbeat messages to other nodes. The election process is triggered in node i when it doesn't receive the messages from the leader due to its departure or crash, as denoted by line 7-8 in the algorithm of [Figure 2]. As described in section 3, node i starts the process of scattering an election message. That is it begins a *diffusing computation* by sending an *Election* message to all of its immediate neighbors, informing them the starting of an election process for a new leader. At triggering a new election, node i sets its variable *status* to "Elect" to indicate that it is in the mode of an election. In the election mode, node i waits until it hears the *Ack* messages from all the connected nodes to which it sends an election message. The list wl_i is, therefore initialized to N_i, i's current neighbors. It is denoted in line 8-10 and 17 of [Figure 2].

2) Detecting all Nodes connected Networks

Node j, upon receiving an election message from i, sends an *Ack* message piggybacked with its neighbors id and weight to the node i and propagates *Election* messages to its own neighbors in the set n_j. Node i, upon receiving an *Ack* message from node j, puts it into the set of confirmed node list cl_i and inserts into the waiting list wl_i the piggybacked neighbors which are in n_i. Therefore, node i knows that all nodes connected to network are detected when the $cl_i$is empty. It is denoted in line 18-22 of [Figure 2].

3) Decide New Leader

When the waiting list wl_i is empty, node i knows that it received the *Ack* messages from all connected nodes and it decides a new leader based on the nodes weight among the set of confirmed node list cl_i that consists of the acknowledged nodes. The exact process to decide new leader is described in line 22 and 40-43 of [Figure 2]. As described in line 17-18 of [Fig. 2], after hearing all *Ack* messages from the nodes in the waiting list *wl_i*, node *i* announce the new leader to other nodes and other nodes received the *leader* messages from node *i* set its variable *ldr* to the new leader's id by which they know who the current leader is.

4) Handling Multiple, Concurrent Computations

It is obvious that more than one node can concurrently detect leader's departure and each of them can initiate diffusing computations independently leading to concurrent diffusing computations. Since each of these computations has the same goal, i.e. to elect a new maximum identity leader, we need to minimize this duplication of effort. Furthermore, the outcome of election is not affected by the identity of the node that initiated the computation and a node has to unnecessarily maintain a large amount of state if it participates in multiple diffusing computations at the same time. We, therefore, handle multiple, concurrent diffusing computations by requiring that each node participate in only a single diffusing computation at any given time. In order to achieve this, each diffusing computation is assigned, what we call, a *computation-index*. This computation-index is a pair, viz. <*num*, *id*>, where *id* represents the identifier of the node which initiated that computation and *num*is integer, which is described below.

Definition: $<num_1, id_1> \; > \; <num_2, id_2> \; <==> \; ((num_1 > num_2) \lor ((num_1 = num_2) \land (id_1 > id_2)))$

A diffusing computation *A* is said to have higher priority than another diffusing computation *B* iff : $computation\text{-}index_A > computation\text{-}index_B$.

When a node participating in a diffusing computation hears another computation with a higher priority, then the node stops participating any furtherits current computation in favor of the higher priority computation. It is described at line 23-26 and 34-37 of [Figure 2].

5) Handling Node Partitions

Once node *j* receives an Election messages from node *i*, it must sends the *Ack* message to the node. But because of node mobility, it may happen that node *j*, which should yet report an *Ack* message to node *i*, gets disconnected from it. Node *i* must detect this event, since otherwise it will never report an *Ack* message to node *i*and therefore, no leader will be elected. In this case, node *i* send an *Election* message to the node *j* again and wait an *Ack* message for a certain timeout period. If node *i* does not received *Ack* message from the node for those period, then it removes the node from the list *wl_i* since the node gets disconnected or crashes. It is described at line 23-26 and 34-37 of [Figure 2].

4.2 Proof of Correctness

The specification for leader election is consisted of two parts. One is *safety* and the other is *liveness*. To verify the correctness of leader election algorithm, the algorithm should be satisfied with both of safety and liveness properties. The safety requirement asserts that all the nodes connected the system never disagree on the leader when the nodes are in a state of normal operation. The liveness requirement asserts that all the nodes should eventually progress to be in a sate of normal operation in which all nodes connected to the system agree to the only one leader. As described in [Fig 2], each node of system has a local variable *ldr* indicating its leader. Since it is impossible to make all nodes change their local variable *ldr* simultaneously, each node uses a variable *status* to reserve the status of system during the process changing their leader. If status equals Norm, the node is normal mode of operation and the value of *ldr* is significant; if *status* has any other value, the node is in a process of a new leader's being elected. We require those nodes to agree to a leader only among nodes whose *status* is Norm. We use subscripts to distinguish local variables of different nodes; for example, ldr_i and $status_i$ are local variables for node *i*.

The safety property of the system with *n* nodes is specified using those local variables. At all times, for all operational nodes *i* and *j*, if $status_i$ = Norm and $status_j$ = Norm, then $ldr_i = ldr_j$. Let's specify the safety property form ally as a following formula SLE1.

SLE1: $(\forall i,j : 1 \le i,j \le n : (status_i=\text{Norm} \land status_j=\text{Norm}) \Rightarrow (ldr_i = ldrs_j))$

The liveness requires that the system eventually progress to a stable state in which the leader is operational and all operational nodes are in the normal state in which they have its status variable with Norm. Such a state is characterized by using the predicate *ldrElected*, defined as below.

Definition: $ldrElected \equiv (\forall i : 1 \le i \le n : ldr_i = j \land (status_i = \text{Norm})))$

Repeated failure and disconnection of nodes will prevent the system from entering the stable state. If there is a period such that there are no more failure and disconnection, the liveness property with *ldrElected* means that a state unsatisfied with *ldrElected* eventually(◇) enter to the state satisfying *ldrElected*. Let us define this formally as a formula SLE2.

SLE2: $ldrElected \Rightarrow \Diamond ldrElected$

SLE2 means that for a given system, there exists a constant *c* such that if no failures or disconnections occur for a period of at least c, then by end of that period, the system eventually(◇) reaches a state satisfying *ldrElected*. Furthermore, the system remains in that state as long as no failures or disconnections occur.

Proof of SLE1(Proof by contradiction). Let's assume following formula, which is the case that there exist two nodes *i*, *j* on the system whose states are Norm and have different leaders.

$(Status_i = \text{Norm} \land Status_j = \text{Norm})\ (ldr_i = i \land ldr_j = j) \land (i \neq j)$

This formula is to be true, at least two nodes in the systems, node *i* and *j*, should have detected the leader's failure or disconnection and entered into the "Elect" mode respectively when the leader had been crashed or disconnected. Each of nodes *i* and *j* should choose itself as a most-valued node respectively in order to declare itself as a leader. But in each election round, only one node has the most value and it would be selected as a leader. Thus it is contradiction. □

Proof of SLE2 (By contradiction) a non-progress means that the new leader is not elected forever even though there is no leader therefore, no *leader* messages must be sent to all nodes. Let us assume that the leader has failed. Because the number of nodes is finite and at least one node is alive, there must be at least one process that detected the leader's disconnection and started the election procedure. Eventually the node receives the *Ack* messages from all other nodes and decides most-valued node as a new leader. Therefore, it is contradiction. □

5. Concluding Remarks

In this paper, we proposed an asynchronous, distributed leader election algorithm for mobile, ad hoc networks and showed it to be correct. We formally specified the property of our leader election algorithm using temporal logic. We have assumed the ad-hoc network topology is dynamically changing and nodes are frequently connected and disconnected over the networks. With this approach, the leader election specification states explicitly that progress and safety cannot always be guaranteed. In practice, our requirement for progress is that there exists a constant *c* such that if connection or disconnections occur for a period of at least c, then by end of that period, the system reaches a state satisfying a leader elected. Furthermore, the system remains in that state as long as no failures or disconnections occur.

In fact, if the rate of perceived a leader failures in the system is lower than the time it takes the protocol to make progress and accept a new leader, then it is possible for the algorithm to make progress every time there is a leader failure in the system.

In real world systems, where process crashes actually lead a connected cluster of processes to share the same connectivity view of the network, convergence on a new leader can be easily reached in practice. However, the algorithm should work correctly even in the case of unidirectional links, provided that there is symmetric connectivity between nodes. We are currently working on the proof of correctness in the case of unidirectional links. We are also investigating on how our election algorithm can be adapted to perform clustering in wireless, ad hoc networks. Acknowledgment : This research was supported by the Ministry of Education, Science Technology and Korea Industrial Technology Foundation through the Human Resource Training Project for Regional Innovation

참 고 문 헌

[1] G. LeLann, "Distributed systems - towards a formal approach," *in Information Processing 77*, B. Gilchrist, Ed. North - Holland, 1977.

[2] H. Garcia-Molian, "Elections in a distributed computing system," *IEEE Transactions on Computers*, Vol.C-31, No.1, pp.49-59, Han 1982.

[3] H. Abu-Amara and J. Lokre, "Election in asynchronous complete networks with intermittent link failures," *IEEE Transactions on Computers*, Vol. 43, No.7, pp.778-788, 1994.

[4] H. M. Sayeed, M. Abu-Amara, and H. Abu-Avara, "Optimal asynchronous agreement and leader election algorithm for complete networks with byzantine faulty links," *Distributed Computing*, Vol.9, No.3, pp.147-156, 1995.

[5] J. Brunekreef, J. P. Katoen, R. Koymans, and S. Mauw, "Design and analysis of dynamic leader election protocols in broadcast networks," *Distributed Computing,* Vol.9, No.4, pp.157-171, 1996.

[6] G. Singh, "Leader election in the presence of link failures," *IEEE Transactions on Parallel and Distributed Systems,* Vol.7, No.3, pp.231-236, 1996(3).

[7] P. David, guest editor, Special section on group communication. *Communications of the ACM*, Vol.39, No.4, pp.50-97, 1996(4).

[8] D. K. Pradhan, P. Krichna, and N. H. Vaidya, Recoverable mobile environments: Design and tradeoff analysis. FTCS-26, 1996(6).

[9] N. Malpani, J. Welch, and N. Vaidya, Leader Election Algorithms for Mobile Ad Hoc Networks. In *Fourth International Workshop on Discrete Algorithms and Methods for Mobile Computing and Communications, Boston, MA*, 2000(8).

[10] K. Hatzis, G. Pentaris, P. Spirakis, V. Tampakas, and R. Tan, Fundamental Control Algorithms in Mobile Networks. In *Proc. of 11th ACM SPAA*, pp.251-260, 1999(3).

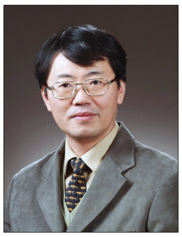
[11] C. Lin and M. Gerla, Adaptive Clustering for Mobile Wireless Networks. In *IEEE Journal on Selected Areas in Communications*, Vol.15,

No.7, pp.1265-1275, 1997.

[12] P. Basu, N. Khan, and T. Little, A Mobility based metric for clustering in mobile ad hoc networks. In *International Workshop on Wireless Networks and Mobile Computing*, 2001(4).

저 자 소 개

박 성 훈(Sung-Hoon Park) 종신회원



- 1982년 2월 : 고려대학교 정경대학(경제학사)
- 1993년 12월 : 인디애나대학교 컴퓨터학과(공학석사)
- 2000년 12월 : 고려대학교 컴퓨터공학과(공학박사)
- 2004년 9월 ~ 현재 : 충북대학교 전자정보대학 컴퓨터공학과 교수

<관심분야> : 분산/모바일/유비쿼터스 컴퓨팅, 정형기법이론, 계산이론