Journal of The Korea Society of Computer and Information
Vol. 21 No. 8, pp. 41-48, August 2016

www.ksci.re.kr
http://dx.doi.org/10.9708/jksci.2016.21.8.041

The Design of an Election Protocol based on Mobile Ad-hoc Network Environment

Sung-Hoon Park*, Yeong-Mok Kim**, Su-Chang Yoo***

Abstract

In this paper, we propose an election protocol based on mobile ad-hoc network. In distributed systems, a group of computer should continue to do cooperation in order to finish some jobs. In such a system, an election protocol is especially practical and important elements to provide processes in a group with a consistent common knowledge about the membership of the group. Whenever a membership change occurs, processes should agree on which of them should do to accomplish an unfinished job or begins a new job. The problem of electing a leader is very same with the agreeing common predicate in a distributed system such as the consensus problem. Based on the termination detection protocol that is traditional one in asynchronous distributed systems, we present the new election protocol in distributed systems that are based on MANET, i.e. mobile ad hoc network.

▸ Keyword : Synchronous Distributed Systems, Leader Election, Fault Tolerance, Mobile Ad Hoc Network.

I. Introduction

In distributed systems, a group of computer should continue to do cooperation in order to finish some jobs. The election protocol is especially helpful tools since it is closely related to group communication [1], which (among other uses) provides a powerful basis for implementing active replications. Whenever a membership change occurs, processes can consent to which of them should do to finish a waiting job or begin a new job. The problem of constructing a stable election protocol is very same with the one of getting common knowledge in a synchronous distributed system such as the consensus problem [1].

The Election problem [1] is defined as that a unique coordinator be elected from a given group of processes. Many research has been widely done so far based on only wired network environment in the research community [2,3,4,5,6,7].

Because many distributed protocols need an election protocol to build fault tolerant distributed systems. However, to our knowledge, there is a few work that has been devoted to this problem in a mobile ad hoc network environment.

When nodes are moving, based on topology change nodes would dynamically join and leave the mobile ad hoc network. In such mobile ad hoc networks, leader election can arise more frequently and having it a particularly critical element of fault tolerant distributed 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

• First Author Sung-Hoon Park, Corresponding Author Sung-Hoon Park
*Sung-Hoon Park (spark@cbnu.ac.kr), Dept. of Computer Engineering, Chungbuk National Univ.
**Yeong-Mok Kim (yeongmokkim@gmail.com), Dept. of Computer Engineering, Chungbuk National Univ.
***Su-Chang Yoo (izibt@nate.com), Dept. of Computer Engineering, Chungbuk National Univ.
• Received: 2016. 06. 20, Revised: 2016. 07. 06, Accepted: 2015. 07. 21.
• This paper was supported by the research grant of the Chungbuk National University in 2014.

or disconnect from the rest of the network. Designing fault tolerant distributed systems in such an MANET environment is a difficult and complex effort. Leader election protocols for MANET have been proposed in [9,10]. In this paper, we are focused on an extrema-finding algorithm, because we believe 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 algorithm and it 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 therefore they are not well-suited to the applications discussed earlier.

Several clustering algorithms have been proposed for mobile networks (e.g. [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.

II. System Model, Constraints and Assumptions

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.

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,…,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 Ni, which indicates the neighboring nodes, with that i can directly communicate the neighboring nodes. We assume that every communication channel is bidirectional; j∈ Ni iff i∈ Nj. More precisely, in the network G = (V, E), we can define E such that for all i∈ V, (i, j) ∈ E if and only if i∈ Nj. The graph can become disconnected if the network is partitioned due to node movement. Because the nodes may changes their location, Ni 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 Wi 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.

III. 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 the 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.

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).

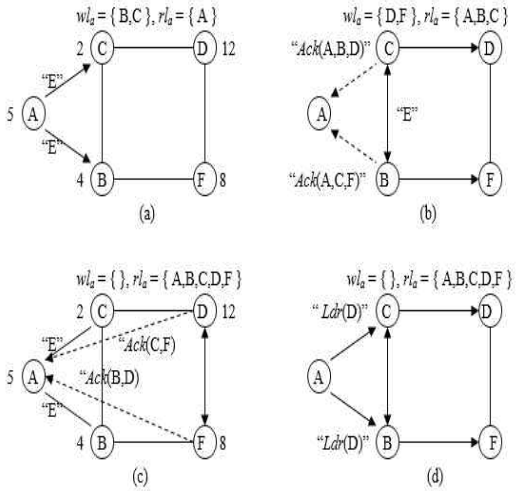

Fig. 1. An execution of leader election algorithm based on the group membership detection algorithm. Arrows on the edges indicate transmitted election messages, while dotted arrows parallel to the edges indicate Ack messages.

IV. 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.

```
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 of n_i except j:
15.        send ack(n_i) to processes j;
16.  On status_i = Elect :
17.    Upon received ack (q) from process j :
18.      wl_i : = wl_i – { j }
19.      cl_i := cl_i ∪ { j };
20.      wl_i := wl_i ∪ { q - { q ∩ cl_i } }
21.  if wl_i = empty then checklist(); end-if
22.    Upon received election(r) from process j :
23.      if { (mum_i, i) < (r, j) } then
24.        send election(r) to each process of n_i;
25.        send ack(n_i) to processes j;
26.        e_num : = r; k : = j;
27.        status_i := Wait; end-if
28.  On status = Wait :
29.    Upon received leader(t) from process j :
30.       ldr_i := t ;
31.    send leader(ldr_i) to each process of n_i except j;
32.       status_i := Norm;
33.    Upon received election(r) from process j :
34.      if { (e_mum, k) < (r, j) } then
35.        send election(r) to each process of n_i ;
36.        send ack(n_i) to processes j;
37.        e_num : = r; k : = j;
38.      end-if
39.  Checklist() :
40.    ldr_i := max (cl_i)
41.    send leader (ldr_i ) to each process of n_i ;
42.    status_i := Norm;
```

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

4.1 Algorithm Performed By the Nodes

In this section, we describe the exact algorithm performed by an arbitrary node i. The exact algorithm is shown in Figure 2. The Leader Election 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 Leader Election module.

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 does not 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 "Election" 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 16-20 of Figure 2.

2) Detecting all Nodes connected Networks: Node j, upon receiving an election message from i, sends an Ack message piggy backed 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 piggy backed neighbors which are in ni. Therefore, node i knows that all nodes connected to network are detected when the cl_i is empty. It is denoted in line 22-27 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 20 and 28-30 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: $\langle num_1, id_1\rangle > \langle num_2, id_2\rangle \Longleftrightarrow ((num_1 > num_2) \vee ((num_1 = num_2) \wedge (id_1 > id_2)))$

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 stop participating any further its 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 Fig. 2.

V. 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 statusi 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 $ldr_i = ldrj_i$. Let's specify the safety property formally as a following formula Safety_LE1.

Safety_LE1: $(\forall i,j : 1 \leq i,j \leq n : (status_i = Norm \wedge status_j = Norm) \Rightarrow (ldr_i = ldr_j))$

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 $\equiv (\forall i : 1 \leq i \leq n : ldr_i = j \wedge (status_i=Norm)))$

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 ($\Diamond$) enter to the state satisfying ldrElected. Let us define this formally as a formula Liveness_LE2.

Liveness_LE2: $\neg$ldrElected => $\Diamond$ldrElected

Liveness_LE2 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 reaches a state satisfying ldrElected. Furthermore, the system remains in that state as long as no failures or disconnections occur.

Proof of Safety_LE1 (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.

$(status_i=Norm \wedge status_j=Norm) \wedge (ldr_i=i \wedge ldr_j=j) \wedge (i \neq j)$

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 Liveness_LE2 (Proof by contra diction) 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.

VI. Conclusions

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.

REFERENCES

[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, April 1982.

[3] N. Mohammed, H. Otrok, W. Lingyu, M. Debbabi, and P. Bhattacharya, Mechanism Design-Based Secure Leader Election Model for Intrusion Detection in MANET, IEEE Transactions on Dependable and Secure Computing, vol.8, no.1, pp.89-103, March 2011

[4] R. Ali, S. Lor, R. Benouaer, M. Rio, Cooperative Leader Election Algorithm for Master/Slave Mobile Ad Hoc Network, 2nd IFIP Wireless Days (WD), Paris, pp. 1-5, 15-17 December 2009.

[5] Masum, S. M., Ali, A. A., Bhuiyan, M. T. I.: Asynchronous Leader Election in Mobile Ad Hoc Networks. AINA 06, 827-831, 2006.

[6] S. Lee, M. Rahman, and C. Kim, A Leader Election Algorithm Within Candidates on Ad Hoc Mobile Networks, Embedded Software and Systems, Lecture Notes in Computer Science, Vol. 4523, pp: 728-738, Springer Berlin / Heidelberg 2007.

[7] M. Lima, A. dos Santos, and G. Pujolle, A Survey of Survivability in Mobile Ad Hoc Networks, IEEE Communications Surveys & Tutorials, 11 (1): 66-75, First Quarter 2009.

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

[9] N. Malpani, J. Welch and N. Vaidya. Leader Election Algorithms for Mobile AdHoc Networks. InFourth International Workshop on Discrete Algorithms and Methods for Mobile Computing and Communications, Boston, MA, August 2000.

[10]K. Hatzis, G. Pentaris, P. Spirakis, V. Tampakas and R. Tan. Fundamental Control Algorithms in Mobile Networks. InProc. of 11th ACM SPAA, pages 251-260, March 1999.

[11]C. Lin and M. Gerla. Adaptive Clustering for Mobile Wireless Networks. In IEEE Journal on Selected Areas in Communications, 15(7):1265-75, 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, April 2001.

Authors

Sung-Hoon Park received the B.S degree in Dept. of Statics and Economics from Korea University, Korea, in 1982 and Ph.D degree at Computer Science from Indiana University, USA in 1996. He is currently full professor in Dept. of Computer Engineering, Chungbuk National University. He is interested in Theory of Computation, Distributed and Mobile Computing.

Young-Mok Kim received B.S degree in Dept. of Management from Korea University, Korea, in 1983 and M.S degree in Dept. of Computer Engineering, Chungbuk National University, Korea, in 2012. He is currently Ph.D student in Dept. of Computer Engineering, Chungbuk National University. He is interested in Theory of Computation, Distributed and Mobile Computing.

Su-Chang Yoo received B.S degree in Dept. of Computer Engineering from Chungbuk National University, Korea, in 2012 and M.S degree in Dept. of Computer Engineering, Chungbuk National University, Korea, in 2014. He is currently Ph.D student in Dept. of Computer Engineering, Chungbuk National University. He is interested in N tiered architecture, Distributed and health-care Programming.