Journal of Digital Contents Society Vol. 11 No. 4 Dec. 2010(pp. 453~461)

모바일 애드 혹 분산 시스템에서 선출 알고리즘의 명세 및 설계

박성훈*

요 약

모바일 애드혹 분산 시스템에서 선출 알고리즘을 명세하고 설계하는 것은 매우 어려운 작업이다. 왜냐하면 모바일 애드혹 분산 시스템은 기존의 분산시스템보다 시스템의 실패에 취약하기 때문이다. 본 논문의 목적은 모바일 애드혹 분산 컴퓨팅 환경에 적합한 선출 알고리즘을 명세하고 하나의 설계모형을 제시하는데 있다. 이러한 목적을 위하여 본 논문에서는 하나의 선출 알고리즘을 설계하고 알고리즘의 정확성을 정형적으로 검증 하였다. 이러한 해결방안은 기존의 분산시스템에서 고전적인 알고리즘인 노드 탐지 알고리즘에 기반으로 하고 있다.

Design and Specification of an Election Algorithm in Mobile Ad Hoc Distributed Systems

Sung-Hoon Park*

Abstract

Specifying and designing the election algorithm in mobile ad hoc distributed systems is very difficult task. It is because mobile ad hoc systems are more prone to failures than conventional distributed systems. The aim of this paper is to propose a specification and design of the election algorithm in a specific ad hoc mobile computing environment. For this aim, we specify and design an election algorithm in this paper. In addition, we formally verify it and show that it is correct. This solution is based on the nodes detection algorithm that is a classical one for synchronous distributed systems.

Keywords: Synchronous Distributed Systems, Leader Election, Fault Tolerance, Mobile Ad Hoc Environment.

1. Introduction

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 [2,3,4,5,6,7]. 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. In mobile ad hoc environments, 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

※ 제일저자(First Author) : 박성훈
접수일:2010년 10월 04일, 수정일:2010년 12월 19일, 완료일:2010년 12월 27일
* 충북대학교
spark@cbnu.ac.kr
▣ 이 논문은 2009년도 충북대학교 학술연구지원사업의 연구비지원에 의하여 연구되었음

or data [8]. In particular, a mobile node can fail or disconnect from the rest of the network. Specifying and designing fault tolerant distributed algorithms and applications in such an environment is very complex endeavor.

The aim of this paper is to propose a specification and design of the election algorithm in a specific ad hoc mobile computing environment. This solution is based on the nodes detection algorithm that is a classical one for synchronous distributed systems. The rest of this paper is organized as follows: chapter 2 describes the mobile system model we use. In chapter 3, a specification 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 chapter 4. We conclude in chapter 5.

2. Mobile System Model and Assumptions

We model it as asynchronous computation with failure detection which is the one described in [9,10]. In the following, we only recall some informal definitions and results that are needed in this paper. Before designing 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 $\{1,2,\ldots,n\}$ $(n>1)$ 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 changes 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 identifiers IDs are used to break ties among nodes which have the same value.

Links are bidirectional and First In First Out 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. Specification of Election Algorithm

In this chapter, we specify a leader election algorithm. In later chapters, we will design our election algorithm based on this specification.

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. 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 formally as a following formula Safety.

Safety :

$$\forall\, i, j : 1 \le i, j \le n : (status_i = Norm \wedge status_j = Norm) => (ldr_i = ldrs_j))$$

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:

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

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 enter to the state satisfying *ldrElected*. Let us define this formally as a formula Liveness.

Liveness :

$$\neg ldrElected => eventually\ ldrElected$$

Liveness 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.

4. Design of Election Algorithm

4.1 Leader Election in Static Networks

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) Member detection phase : it operates by first "scattering the election message" and then "gathering the id of each node " that is connected to the static networks. 2) Decision and notifying phase : after gathering the id of all members, the source node decides the most valued node and announces it as a new leader to all other members. 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) Member Detection 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. 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.

2) Decision and Notifying Phase. Once the source node has received *Acks* 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).

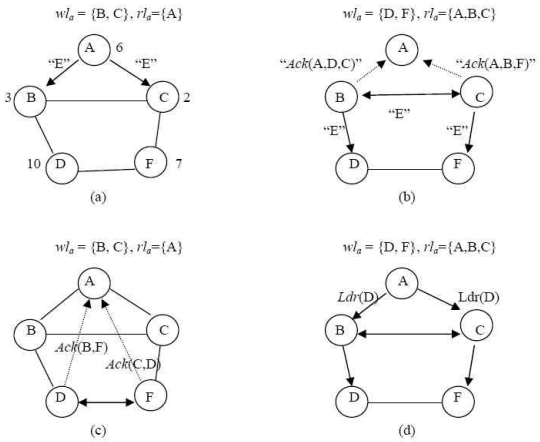

Figure 1: An execution of leader *Election* algorithm based on the group membership detection algorithm. Arrows on the edges indicate transmitted election messages, while dotted arrows parallel to the edges indicate *Ack* messages.

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).

4.2 Leader Election in Ad Hoc Networks

In this chapter, 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 chapter, we described the leader election algorithm LE 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 chapter is inadequate. Furthermore, we assumed in the previous chapter 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.

we describe the exact algorithm performed by an arbitrary node i. The exact algorithm is shown in Figure 2. The 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 figure 2 as specified in the initialization part of the *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 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 chapter 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 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 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 Figure 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.

```
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
32.      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.

4) 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 wli since the node gets disconnected or crashes. It is described at line 23 26 and 34 37 of Figure 2.

4.3 Proof of Correctness

Proof of Safety (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. That is,

$$(Status_i = Norm \wedge Status_j = Norm) \wedge (ldr_i = i \wedge ldr_j = j) \wedge (i \neq j)$$

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 (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

We formally specified the property of our leader election algorithm using temporal logic and designed an asynchronous, distributed leader election algorithm for mobile ad hoc networks and showed it to be correct.

In the ad hoc network, topology is dynamically changing and nodes are frequently connected and disconnected over the networks. Within this environment, the leader election specification states that progress and safety always cannot be guaranteed. In practice, our requirement for progress is that there exists a constant time c such that if connection, disconnection or failure does not occur for a period of at least c, then by end of that period, the system reaches a stable 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 [11,12].

In real distributed 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 financially supported by the Ministry of Education, Science Technology and Korea Industrial Technology Foundation through the Human Resource Training Project for Regional Innovation.

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, 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, March 1996.

[7] David Powell, guest editor. "Special section on group communication." Communications of the ACM, 39(4):50 97, April 1996.

[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 Ad Hoc Networks." In Fourth 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." In Proc. 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.

박 성 훈

1982년 2월 : 고려대학교 정경대학 (경제학사)
1993년 12월 : 인디애나대학교 컴퓨터학과 (공학석사)
2000년 12월 : 고려대학교 컴퓨터공학과 (공학박사)
2010년～현 재 : 충북대학교 전자정보대학 컴퓨터공학부 교수
<관심분야> : 분산/모바일/유비쿼터스 컴퓨팅, 정형기법이론, 계산이론