분산 시스템에서 고장 추적 장치를 이용한 선출 알고리즘

An Election Algorithm with Failure Detectors in Distributed Systems

박성훈
충북대학교 전기전자컴퓨터공학부

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

요약

본 논문에서는 동기적인 분산시스템에서 불리 알고리즘에 기초하여 하나의 새로운 선출(election) 알고리즘을 설계하고 이를 분석하고자 한다. 동기적인 분산시스템에서 기존의 불리 알고리즘은 고장 추적 장치를 이용하여 더욱 효율적으로 설계되고 구현 될 수 있음을 보인다.

■ 중심어 : | 선출 알고리즘 | 고장 추적자 | 동시성 제어 | 분산 시스템 |

Abstract

In this paper, we design and analyze an election algorithm, based on the Bully algorithm, in synchronous distributed systems. We show that the Bully algorithm, when using Failure Detector, is more effectively implemented than the classic Bully algorithm for synchronous distributed systems.

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

1. Introduction

Leader election, simply election, is an important problem to construct fault-tolerant distributed systems. Depending on a network topology, many kinds of leader election algorithms to elect a high-priority leader have been presented so far. Some algorithms are based on complete topology and others based on ring topology [1-3] or tree topology [4-7]. Among those, as a classic paper, there is the Bully algorithm for synchronous systems based on complete topology specified by Garcia-Molina [8].

The leader election algorithm is used usefully in those systems where a coordinator of the protocol is needed, such as replicated data management, atomic commitment, process monitoring and recovery. In this paper, we show the Bully algorithm, when using failure detector (FD)[9], is more effectively implemented than the classic Bully algorithm in synchronous distributed systems with crash failures. Garcia-Molina's Bully algorithm detects a crashed node by time-out intervals, but the modified Bully algorithm presented in this paper uses a failure-detector instead of the explicit time-out.

A failure detector is an independent module that detects and reports crashes of other nodes. There are

* 본 논문은 2006학년도 충북대학교 학술연구지원사업의 연구비 지원에 의하여 연구되었습니다.
접수번호 : #071018-002
접수일자 : 2007년 10월 18일
심사완료일 : 2008년 2월 11일
교신저자 : 박성훈, e-mail : spark@cbnu.ac.kr

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

2. Model and Failure Detector

2.1 Model

Our model of asynchronous computation with failure detection is the one described in [9][10]. In the following, we only recall some informal definitions and results that are needed in this paper. We use integers to identify the nodes connected on the system and specify the set of nodes as formula (3.1).

$$ID = \{ 1, 2, \ldots\ldots, n \} \qquad (3.1)$$

where n means total number of nodes connected on the system and integers identifying nodes means parameter which decides priority of them. For simplicity, we use node identifiers as priorities: lower numbers correspond to higher priorities, as in UNIX. That is, the priority of node 1 has the highest and the priority of node 2 is second high and so on. Bully algorithm is designed for the system with a few of following properties. As a system environments the synchronous system is assumed, where transmission and processing time of the messages occurring between nodes is predicted and information exchanges between nodes is done within the given time. A system is based on the fully connected communication networks in which fixed number of nodes is inter-connected through them. Nodes crash and recover. We do not assume any other kinds of failures such as Byzantine failures. Each node has access to a small amount of stable storage for relevant information that is used for recovering right after the failures. Communications between nodes are done as sending messages. Communication is executed as FIFO. We assume also that communications under the synchronous system is reliable.

2.2 Failure Detector

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

If the node i is in the down_list, FD informs the client that the node i is down by raising the signal

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

By managing the information about crashed nodes as a form of the down node list, the FD can send the information about crashed nodes to the client more promptly than the Garcia-Molina's one can. When the crashed node recovers again, it sends to the FD on each node immediately the message informing that it has recovered. After receiving the message, if the name of the recovered node exists in the down_list, the FD removes the node's name from the down_list.

There are many other methods to implement failure detector. For example, the simplest implementation of failure detector is to send the "Are You Alive?" message to each node being monitored periodically. If a reply is not received in the expected time, FD raises <DownSig, *i*> for the node. A more slightly complicated approach is for each node *i*, when it starts monitoring node *j*, to tell node *j* to periodically send "I'm alive" message to node *i*. This uses fewer messages and reduces the latency of the FD. A more complicate approach, based on an attendance list [1][2], is to a construct logical ring and periodically circulates a token around it. If a node does not see the token within the expected time, then one or more nodes are failed, and "Are you alive?," messages can be used to pinpoint the failed nodes. With this approach, fewer messages will be used if multiple nodes are being monitored by multiple nodes, though at the expense of increased detection latency.

3. Bully Algorithm Using FD

3.1 Garcia-Molina' S Bully Algorithm

The Bully algorithm, which is Garcia-Molina's leader election algorithm for synchronous distributed system, shows that all the nodes in a group can reach a stable state in which every not crashed node agrees to only one leader.

The Bully Algorithm works as follows. When a process notices that the coordinator is no longer responding to requests, it initiates an election. A process, *P*, holds an election as follows:

1. *P* sends an *ELECTION* message to all processes with higher numbers.
2. If no one responds, *P* wins the election and becomes coordinator.
3. If one of the higher-ups answers, it takes over. *P*'s job is done.

At any moment, a process can get an *ELECTION* message from one of its lower-number colleagues. When such a message arrives, the receiver sends an *OK* message back to the sender to indicate that he is alive and will take over. The receiver then holds an election, unless it is already holding one. Eventually, all processes give up but one, and that one is the new coordinator. It announces its victory by sending all processes a message telling them that it is the new coordinator.

If a process that was previously down comes back up, it holds an election. If it happens to be the highest-numbered process currently running, it will win the election and take over the coordinator's job. Thus the biggest guy in town always wins, hence the name "bully algorithm".

This algorithm does exactly the function of electing a leader in the distributed systems where small number of nodes are connected as a group and the frequency of each node's crash failures is relatively low. But in the system where large number of nodes are connected as a group and each node's crash failures are frequent, the execution speed for the algorithm will be slow. At the worst case, the system may not reach for a long time to the stable state in which every not crashed node agrees only one leader.

Repetitions of crash failures of nodes with high priority continue to have the system be in the state of the leader election since a lot of time is spent to detect whether a node is crashed. The modified Bully algorithm using a failure detector, which we call Bully_FD algorithm, is faster than the Garcia-Molina's one in terms of execution speed. Because it greatly reduces the time taken for the detection of crashed nodes.

3.2 Description and algorithm of Bully_FD

Basic idea of Bully algorithm is that the operational node with the highest priority is elected as a leader among all of nodes. Each node i has a *status* variable, initially having Norm value. Following is the scenario of the leader election using FD.

When node i detects that its leader is crashed or when the crashed node is recovered, the node sets its *status* variable to $Elec_1$ and indicates that it is in the stage 1 of organizing a leader election. In stage 1, node i checks whether the nodes of less(i), whose priority is higher than itself, is operational or not.

If some of them are operational, node i stays on the Wait state in order to give those higher-priority nodes a chance to become the leader. If it is recovered from the Dead state, it waits for the message <Norm?, t> asking the state of recovered node from the leader. If none of nodes with higher-priority are operational (i.e., if node i receives the message <downSig, j> for all $j \in$ less(j) from FD), then it stays in stage 2 of organizing a leader election and sets its *status* variable to $Elec_2$.

On stage 2, node i prepares the lower-priority nodes in greater(i) for a new leader by sending them Halt message. When a node receives a Halt message, it sends an Ack message and switches its *status* variable from Norm to Wait state indicating that it is waiting for the outcome of the election. If a node on Wait state detects the failure of the node which halted it by receiving <DownSig, i> from FD, it switches its status from Wait to $Elec_1$ and restarts the election. When the node on stage 2 which organized the election has received an acknowledgement signal from each node in greater(i), then it becomes a leader to set its status to Norm and notifies the fact that it is elected as a new leader to all nodes in greater(i) by sending *ldr* message. All nodes received *ldr* message from node i accepts node i as their a new leader, switching their status from Wait to Norm.

Periodically, the leader sends the message <Norm?,t> checking status of node to all nodes with lower-priorities in order to find out whether recovered nodes exist. The node which has received the message <Norm?,t> sends the message <NotNorm,t> if it is not on Norm state.

The leader which has received message <NotNorm,t> switches its state to $Elec_1$, and then it does the leader election process again.

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

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

```
Var status : {Norm, Elec1,Elec2,Wait }
Var ldr : ID
Var elid : ID × Natural Number
Var down : set (less(i))
Var dw_gter,acks : set(greater(i))
Var nextel : Natural Number
Var stable incarn : Natural Number

Periodically(τ) do
if status = Norm ∧ ldr ≠ i then
  Request_FD(ldr) fi od
procedure Stage1()
status ← Elec1; down ← ∅
elid ← <i,incar, nextel>
if  i = 1 then Stage2()
else Request_FD(less(i)) fi

Upon receive <Halt,t> from j do
down ← down - { j }; elid ← t; status ← Wait
Send <Ack, t> to j  od

Periodically(τ) do
if status = Wait then Request_FD(j)  od

Upon receive <downSig,j> from FD do
if  j ∈ less(i) then down ← down ∪ { j }
   if ( status = Norm ∧ j = ldr )
      ∨ (status = Wait ∧ j = head(elid)) then
         Stage1()
   else if status = Elec1 ∧ down ⊇ less(i) then
         Stage2() fi  fi
else  (*j ∈ greater(i) * )
  dw_gter ← dw_gter ∪ { j } fi  od

Upon receive <UpSig, j> from FD do
if j ∈ greater(i) ∧status = Elec2 then
     send <Halt, elid> to j fi od

procedure Stage2()
status ← Elec2; acks ,dw_gter ← ∅
Request_FD(greater(i))

Upon receive <Ack,t> from j do
acks ← acks ∪ { j }
if status = Elec2 ∧((acks∪dw_gter) ⊇ greater(i))
  then ldr ← i status ← Norm
       send <Ldr,t> to acks fi od

Upon receive <ldr,t> from j do
if status = Wait ∧ t = elid then
   ldr ← j; status ← Norm fi od

Periodically (τ) do
if status = Norn ∧ ldr = i then
   Send <Norm?, elid> to greater(i)  fi od
```

```
Upon recrive <Norm?,t> from j do
if $status \neq$ Norm then
   Send <NotNorm,t> to j fi od

Upon receive <NotNorm, t> from j do
if $status = \text{Norm} \land ldr = i \land t = elid$ then
   Stage1() fi od

Upon recovery do
$incarn \leftarrow incarn + 1$; Stage1() od
```

Figure 3.1 Bully_FD Algorithm

The expressions $S \cup x$ and S-x denote the element x is added to and removed from set S respectively. The expressions $S \leftarrow (Sx)$ and $S \leftarrow (S\text{-}x)$ mean that element x is added to and removed from set S, thus set S is updated. The operator *head* returns the first element of a tuple.

The significant differences between existing Bully algorithm and Bully_FD algorithm is as follows.

1. Bully_FD algorithm uses a failure detector rather than explicit time-outs to track failed nodes. In the original Bully algorithm, node i waits a reply from node j to confirm a node's failure. But in Bully_FD algorithm, node j is being monitored by node i's FD and node i receives either <DownSig,i> or <UpSig, i> from the failure detector. Note that procedure time-out in the Bully algorithm is, in effect, integrated into the codes of handling <DownSig,i> in the Bully_FD algorithm.
2. In stage 1 of an election, each node checks concurrently rather than sequentially whether the nodes with lower priorities is operational. This optimization is independent of the use of a failure detector, but we can take advantage of such techniques using FD but would be awkward to express using Garcia-Molina's RPC-style communication primitive.
3. Each message has an election identifier that identifies the elections, so we can avoid confusions incurred from the deferred messages on the network.

3.2 Efficiency Analysis of the Processing Time

Let's compare the processing time of Bully_FD algorithm proposed on this chapter with the Garcia-Molina's Bully algorithm. We define the elements that affect the processing time as follows.

N:	Total number of nodes on the system
N_f:	The number of failed nodes
T_m:	Average propagation time per message of a node
T_p:	Average message handling time of a node
T_o:	Time-out ($T_o > T_e$)
T_e :	Average response time from a node

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

$$T_e = 2\,T_m + T_p \qquad (3.2)$$

When the leader fails, a node with highest priority, whose identification is k, recognizes it and starts the leader election and finishes it. Following formula describes the total processing time in Bully algorithm of Garcia-Molina.

$$\begin{aligned} T_{GM} &= (k-1)*T_o + [(N_f - k+1)*T_o + (N - N_f)*T_e] \\ &= N_f * T_o + (N - N_f) * T_e \qquad (3.3) \end{aligned}$$

The term $((k-1)*T_o)$ in formula (3.3) describes the time taken when node k detects that all nodes with higher-priority are crash failed. It is the time required on transiting from the state $Elec_1$ to $Elec_2$. The term $[(N_f-k+1)*T_o+(N-N_f)*T_e]$ describes the time taken on the node k's checking whether the nodes with lower-priority are crashed or not. It is the time taken on transiting from the state $Elec_1$ to Norm.

In the same way, total time taken from start leader election to finish it on executing the Bully_FD algorithm is formulated in formula (3.4).

$$T_{BULLY\_FD} = 2*N*T_s + (p*N_f*T_p + (1-p)*N_f*T_o) + (N-N_f)*T_c \qquad (3.4)$$

In formula (3.4), T_s denotes the time required for a node to send one message to a FD. Consequently, the term $2*N*T_s$ is the total time taken to transmit messages between node k and the FD as one of signal forms. Let's assume that p is the ratio of all failed nodes to the failed nodes which FD has already known as it is written in its down node list. For instance, if 10 nodes have been failed, and FD has written 7 nodes in down node list, then the value of p is 0.7. The term $(p*N_f*T_p + (1-p)*N_f*T_o)$ means the time taken for FD to detect the failure of each node, and the term $(N-N_f)*T_c$ signifies the time for FD to confirm the liveliness of the normal nodes.

$$T_d = (3.3) - (3.4) = p*N_f*(T_o-T_p) - 2*N*T_s \approx p*N_f*(T_o-T_p) \qquad (3.5)$$

By using the formula (3.3) and (3.4), the formula (3.5) is induced as below which describes the difference of processing time between Garcia-Molina Bully algorithm and Bully_FD algorithm. In the formula (3.5), the value of $2*N*T_s$ is small enough to be negligible. As I mentioned before, it is the time required for the message exchanges to detect the failed nodes between the node leading the leader election and the FD. The message exchanges between them are executed almost concurrently rather than sequentially. Definitely $(T_0-T_p) > 0$ is true and $p*N_f*(T_0-T_p) > 0$ is also true. Thus, we can make sure that Bully_FD algorithm is faster than Garcia-Molina Bully algorithm in processing time.

4. Concluding Remarks

So far, many algorithms related with leader election on distributed system are proposed [11-14]. Many of them have concentrated on the solution to the problem of self-stabilizing construction of system using timeout interval. The leader election algorithms based upon timeout interval are clear and simple in terms with semantics in the system where the small number of nodes are connected and the frequency of each node's crash and failure is relatively low. But in the distributed system where many heterogeneous nodes are connected and the frequency of each node's relatively high, there are some of problems such as prolongation of executing time. The Bully_FD algorithm is same as the classic Bully algorithm in terms with using timeout interval to detect the crashed nodes. The difference between two algorithms is that the Bully_FD algorithm uses the FD but the classic Bully algorithm uses timeout interval directly to detect the crashed nodes. By doing this, Bully_FD algorithm can detects the crashed nodes concurrently rather than sequentially and thus the speed of processing time in the Bully_FD is more enhanced than the classic one. As another advantage, FD is a module so that modularity facilitates use of different failure detection mechanism in different systems.

참 고 문 헌

[1] E. J. Chang and R. Roberts, "An improved algorithm for the decentralized extrima-finding in circular configurations of processes," *Communication of ACM*, Vol.22, No.5, pp.281-283, 1979.

[2] G. L. Peterson, "An O(nlogn) unidirectional algorithm for circular extrima problem," *ACM Trans. Programming language and systems*, Vol.4, pp.758 - 762, 1982.

[3] D. S. Hirshberg and J. B. Sinclair, "Decentralized extrima finding in circular configurations of Processors," *Communications of the ACM*, Vol.23, No.11, pp.627-628, 1980.

[4] R. Gallager, P. Humblet, and P. Spira. "A distributed algorithm for minimum weighted spanning trees," *ACM trans. On Programming language and Systems*, Vol.5, No.1, pp.66-77, 1983.

[5] G. Gafini, "Improvement in time complexity of two message optimal algorithm," *Proc. Principles of Distributed Computing Conf.*, pp.175-185, 1985.

[6] F. Chin and H. F. Ting, "An almost linear time and O(nlogn+e) message distributed algorithm for minimum weighted spanning trees," *Proc. Foundations of Computer Science Conf.*, pp.257-166, 1985.

[7] R. Chow, K. Luo, and R. N. Wolfe, "An optimal distributed algorithm for failure driven leader election in bounded-degree networks," *Proc. IEEE Workshop on Future Trends of Distributed Computing Systems*, pp.136-141, 1992.

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

[9] D. C. Tushar and T. Sam, "Unreliable failure detectors for reliable distributed systems," *Journal of ACM*, Vol.43, No.2, pp.225-267, 1996.

[10] V. Hadzilacos and S. Toueg, "B. Reliable and Related Problems," *In Distributed Systems (Second Edition)*, ACM Press, New York, pp.97-145, 1993.

[11] D. Shlomi, I. Amos, and M. Shlomo, "Uniform dynamic self-stablizing leader election," In Sam Toueg, Paul G. Spirakis, and K. Lefteris, *Proc. 5th International Workshop on distributed Algorithms*(WDAG '91), of Lecture Notes in Computer Science, Vol.579, pp.167-180, 1991.

[12] I. Gene, L. Chengdian, and S. Janos, "Deterministic, constant space, self-stabilizing leader election on uniform rings," In Jean-Michel Helary and Michel Raynal, editors, *Proc. 9th International Workshop on Distributed Algorithms*(WDAG '95), Vol.972, pp.288-302, 1995.

[13] G. Rachid, "On the hardness of failure-sensitive agreement problems," *Information Processing Letter*, Vol.79, No.2, pp.99-104, 2001.

[14] A. Mostefaoui, E. Mourgaya, and M. Raynal, "Asynchronous Implementation of Failure Detectors," *Proc. Int. IEEE Conference on Dependable Systems and Networks(DSN'03)*, IEEE Computer Press, San Francisco(CA), pp.351-360, 2003.

저 자 소 개

박 성 훈(Sung-Hoon Park) 종신회원

- 1982년 2월 : 고려대학교 정경대학 통계학과(경제학사)
- 1993년 2월 : 인디애나대학교 대학원 컴퓨터학과(공학석사)
- 1997년 2월 : 고려대학교 컴퓨터공학과(공학박사)
- 2004년 9월 ~ 현재 : 충북대학교 전기전자컴퓨터공학부 교수

<관심분야> : 분산/모바일/유비쿼터스 시스템 및 알고리즘