http://dx.doi.org/10.5392/IJoC.2011.7.4.030

On Relationship between Safety and Liveness of Election Problem in Asynchronous Distributed Systems

Sung-Hoon Park
Dept. of Computer Engineering
Chungbuk National University, Chung-Buk 330-800, Korea

ABSTRACT

A Leader is a Coordinator that supports a set of processes to cooperate a given task. This concept is used in several domains such as distributed systems, parallelism and cooperative support for cooperative work. In completely asynchronous systems, there is no solution for the election problem satisfying both of safety and liveness properties in asynchronous distributed systems. Therefore, to solve the election problem in those systems, one property should be weaker than the other property. If an election algorithm strengthens the safety property in sacrifice of liveness property, it would not nearly progress. But on the contrary, an election algorithm strengthening the liveness property in sacrifice of the safety property would have the high probability of violating the safety property. In this paper, we presents a safety strengthened Leader Election protocol with an unreliable failure detector and analyses it in terms of safety and liveness properties in asynchronous distributed systems.

Keywords: *Distributed Computing, Leader Election, Asynchronous Distributed Systems, Failure Detectors*

1. INTRODUCTION

Distributed systems consist of groups of processes that cooperate in order to complete specific tasks. A *Leader* is a Coordinator that supports a set of processes to cooperate a given task. This concept is used in several domains such as distributed systems, parallelism and cooperative support for cooperative work.

To elect a *Leader* (or Coordinator) in a distributed system, an *agreement* problem must be solved among a set of participating processes. This problem, called the *Election* problem, requires the participants to agree on only one leader in the system [1]. The problem has been widely studied in the research community [2]-[6]. One reason for this wide interest is that many distributed protocols need an election protocol.

The Election problem is described as follows. At any time, there is at most one process that considers itself a *leader* and all other processes consider it as to be their only leader. If there is no leader, a leader is eventually elected.

The so-called FLP impossibility result proved by Peterson and Lynch, which states that it is impossible to solve any non-trivial agreement in an asynchronous system even with a single crash failure, also applies to the election problem [7],[8]. That means that there is no solution for the election problem satisfying both of safety and liveness properties in completely asynchronous distributed systems.

It must be pointed out, however, that the impossibility result really means “not always possible,” as opposed to “never possible.” As a matter of fact, any algorithm that tries to solve the Election Problem cannot always make progress without violating safety; there exist cases in which the algorithm violating safety, although it is very unlikely.

Therefore, to solve the election problem in those systems, one property should be weaker than the other property. If an election algorithm strengthens the safety property in sacrifice of liveness property, it would be difficult to progress. But on the contrary, an election algorithm strengthening the liveness property in sacrifice of the safety property would have the high probability of violating the safety property. There exists a trade-off between safety property and liveness property.

A stable election protocol, which implies the safety strengthened election protocol, is needed in a practical distributed computing environment. Consider a mission critical distributed system such as an electronic commerce system that runs multiple servers in which one of them roles a master (leader) and others are slaves.

To have data consistency among the servers in the system, this system should not violate safety property, which means that all processes connected the system never disagree on a *leader*. In those systems the safety property is more important property than the liveness property.

As a classic paper, there is Garcia-Molina’s Invitation algorithm to solve election problem in asynchronous distributed systems. The algorithm strengthens the progress property rather than

* *Corresponding author. E-mail : spark@chungbuk.ac.kr*
Manuscript received Sep 30, 2011 ; accepted Dce.13, 2011

*This work was supported by the research grant of the Chungbuk National University in 2010

safety and it allows more than two leaders in the systems.

Our idea is based upon the Garcia-Molina's Invitation algorithm for solving the election problem in asynchronous distributed systems [2]. He redesigns the Bully algorithm for synchronous distributed systems into the Invitation algorithm for asynchronous distributed systems by using a specification that is weak enough to be solvable, allowing the algorithm to progress even in completely asynchronous distributed systems.

His specification uses a strong progress requirement, allowing executions in which even a single process suspicion of the current leader's crash and its attempted leader election from the members may lead a progress to elect a new leader from all processes.

We propose an election algorithm that requires processes to elect a new leader only when they agree with the current leader's crash. This requirement is strong because, if no set of processes agrees on the current leader's crash, no progress is made. The requirement is, however, much more stronger than the one proposed by Garcia-Molina's Invitation algorithm in that it implicitly states that the leader election of any process be allowed only on the basis of only it's own knowledge.

In this paper, we presents a safety strengthened Leader Election protocol with an unreliable failure detector and analyses it in terms of safety and liveness properties in asynchronous distributed systems.

Our algorithm, based on a standard three phases commit protocol, is fully distributed. It does not extend the asynchronous model of concurrent computation to include global failure detectors. Progress of the algorithm can be guaranteed only in case of minimal violating a safety property.

The rest of the paper is organized as follows. In Section 2, we describe our system model and definitions. In Section 3, this paper relates the election specification to other ways to solve the election problem. In Section 4, this paper provides a stable algorithm that solves the Leader Election problem. In Section 5, we ensure the correctness of the algorithm by proving that it satisfies the two properties of the specification given in Section 4. Finally, Section 6 summarizes the main contributions of this paper and discusses related and future works.

2. MODEL AND DEFINITIONS

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.

2.1 Processes

We consider a distributed system composed of a finite set of processes $\Omega=\{p_1,p_2,..,p_n\}$ where processes are identified by unique id's. Communication is by message passing, *asynchronous* and *reliable*. Processes fail by crashing; Byzantine failures are not considered.

Every pair of processes is connected by a communication channel. That is, every process can send messages to and can receive messages from any other. We assume processes are able to probe a communication channel for incoming messages. Communication channels are considered to be reliable, FIFO, and to have an infinite buffer capacity. A reliable channel ensures that a message, sent by a process p_i to a process p_j, is eventually received by p_j if p_i and p_j are correct (i.e. do not crash).

Asynchrony means that there is no bound on communication delays or process relative speeds. A process that has been infinitely slow for some time and has been unresponsive to other processes may become responsive again at any time. Therefore, processes can only suspect other processes to have crashed, using local failure detectors.

A failure detector is a distributed oracle which gives hints on failed processes. We consider algorithms that use failure detectors. Local failure detectors are assumed to be inaccurate and incomplete. That is, local failure detectors may erroneously suspect that other, operational processes have crashed or that crashed processes are operational. Since local failure detectors run independently at each process, one local failure detector may perceive a failure, but other detectors may perceive it at a different time or not at all.

The failure model allows processes to crash, silently halting their execution. Because of the unpredictable delays experienced by the system, it is impossible to use time-outs to accurately detect a process crash.

We assume that a process communicates with its local failure detector through a special receive-only channel on which the local failure detector may place a new list of id's of processes not suspected to have crashed. We call this list the local connectivity view of the process. Each process considers the last local connectivity view received from its local failure detector as the current one.

2.2 Election Specifications

The Election problem is described as follows: At any time, as most one process considers itself the leader, and at any time, if there is no leader, a leader is eventually elected. More formally, the Election Problem is specified by the following two properties:

- *Safety*: All processes in the local connectivity view of the process never disagree on a *leader*.
- *Liveness*: All processes should eventually progress to be in a state in which all processes connected to the system agree to the *only one* leader.

3. CIRCUMVENTING THE IMPOSSIBILITY RESULT

In this section, we relate the election specification to other ways to solve the election problem.

- In an asynchronous model augmented by global failure detectors, processes have access to modules that (by definition) eventually reflect the state of the system. Therefore, progress and safety can be guaranteed unconditionally.
- In a timed asynchronous model, processes must react to an input, producing the corresponding output or changing state,

within a known time bound. Under this model, progress and safety can be guaranteed if no failures and recoveries occur for a known time needed to communicate in a timely manner.
- In a completely asynchronous model, progress cannot always be guaranteed without violating safety and failure detectors in practice eventually reflect the system state, but they must be considered arbitrary. Correct processes react in practice within finite time, but this time cannot be quantified. Therefore, in order to guarantee a solution, we need a weaker specification of the problem.

Our approach falls into the last category that originated with Garcia-Molina's work [2]. Our election algorithm, however, differs from Garcia-Molina's in several ways.

- Processes in Garcia-Molina's model do not need to wait to get consensus about the current leader's crash. If one process suspects that the leader failed, it may attempt to elect the new leader. Garcia-Molina's specification says that, if one process attempts to be a new leader, it eventually should be elected as a leader. Our specification requires all processes in a set to agree on the current leader crash before changing their new leader.
- Garcia-Molina's specification allows a solution in which the attempted change of a leader divides all processes into several sub-groups. Our specification does not allow such a sub-group because it states that if all processes in a system agree on a new leader, they must eventually accept such a leader.

In our model stability is also required for progress, but, at variance of the above case, it is not necessarily related to the state of the system. In other words, eventual progress is required when there is agreement among a set of the local failure detectors, even if failures and recoveries continue to occur in the system.

4. ELECTION ALGORITHM

We provide a stable algorithm that solves the Leader Election problem given in Section 2. The algorithm is based on the three asynchronous phases.
- A prepare phase, in which a process propose a new leader that the other processes agree with.
- A ready phase, in which all process that agree on the new leader acknowledge the reservation of the potential leader.
- A commit phase, in which the new leader is finally elected, and all process accept it their only leader.

4.1 Solution Sketch

The main idea for the algorithm is as follows. A process *p* that is informed by its local failure detector of a leader's crash and that has the smallest id among processes in its new local connectivity view sends a message to all processes in its view proposing to change the current leader with the new leader.
Each process received the message records this proposal until the potential leader in its local view is the same as the proposed new leader in its local view. At which point, it responds by sending back an Accept or Retry message to the process that proposed the leader update. The Accept message is sent if the process agrees on the proposed leader in its local current view.
Upon sending the Accept message, the process reserves the prospective leader, so that no other proposal is accepted for that system. Upon receiving a Retry message, the proposing process returns the normal state of the algorithm, sending a new Abort message to all processes in its view.
When the proposing process has collected Accept messages from all processes in its view, it starts the commit phase by sending commit messages, ordering other processes in its view to commit the leader update. Upon receiving a commit message, the processes accept the reserved prospective leader as a their new leader.

4.2 Code Description

The code is shown in Fig. 1. The first received command in Fig. 1 shows how a process *p*, when informed of a change in its local connectivity view, set its view to be current and checks if the current leader has crashed. If the leader has crashed, it set the variable *LeaderStatus* to be false. When *LeaderStatus* is false, the *StartElection* procedure in Fig.1 is called and the process *p* checks that it is the minimum id among the processes in v_p. If *p* is the minimum id, it increases the round and proposes itself as a new prospective leader and initializes its *ack* array to zero.
The next received commands in Fig. 1 check for incoming messages from other processes. These may be proposals for a new leader (Propose), rejections to propose a new leader (Rejection), acceptances of a proposed new leader (Accept), orders to commit a new leader (Commit) or orders to abort a proposed new leader (Abort).

```
Upon received v_p from FD:
    CurView := true;
    If CurLeader ∉ v_p then LeaderStatus := 0;
    end-if
   If p = min(v_p) then Round := Round +1;
                Call start_election();
   end-if

Upon received (Propose,PropLeader, k) from q:
     Prop:= true; CurView := false;
    NewLeader:= PropLeader ; RoundIn := k;
     Call reply_election();

Upon received (Reject, k) from q:
     If Round = k then
       Send (Abort, Round) to ∀ j ∈ v_p;
        For ∀j ∈ v_p, ack[j] :=0;
    end-if

Upon received (Accept, k) from j:
   If Round = k then ack[j] := 1;
     If for ∀q∈ v_p, ack[q] = 1 then
        Send (Commit, PropLeader, Round) to
                          ∀q∈ v_p;
```

```
        For ∀q ∈ v_p, ack[q] :=0;
      end-if end-if

Upon received (Commit, PropLeader, k) from j:
   If RoundIn = k then
                CurLeader := PropLeader;
                LeaderStatus := 1;
    end-if

Procedure Start_election():
   PropLeader := p;
   Send (Propose, PropLeader, Round) to
          ∀q∈ v_p;
   For ∀q∈ v_p, ack[q] :=0;

Procedure Reply_election();
   If ( CurView ∧ Prop ) then Prop:= false;
     If (Newleader≤min(v_p)∧RoundIn>Round)
       then Send ( Accept, PropIn) to q;
            Next = RoundIn + 1;
     end-if
    else Send (Reject, PropIn) to q;
   end-if
```

Fig. 1. The Algorithm.

Upon receiving a proposal message from process q, process p stores the new leader's id proposed by q at position q of the array NewLeader and stores the proposed round at position q of the array RoundIn, then sets position q of the array Prop to true to record the receipt of the proposal from q and sets the CurView to false to refresh the current view of the system

If process *p* later agrees on the proposed new leader, it sends a response to process *q* (see last guarded command in Fig. 1). The response is either an acceptance of the new leader at position NewLeader[*q*] if the minimum id among the process in v_p is greater or equal than the id of proposed NewLeader[*q*] and the proposed round greater than the current round; or it is an rejection to the proposed new leader if the minimum id among the process in v_p is less than the id of proposed NewLeader[*q*] or the proposed round less or equal than the current round.

A rejection to the proposed new leader consists of sending back to *q* the proposed round. An acceptance consists of acknowledging the proposed new leader at position NewLeader[*q*].

We now examine the guarded commands of the remaining message types. A process *p* that receives a rejection to the its proposal sends all processes in v_p a message to abort the proposed round and reinitializes the ack array to zero.

A process *p* that receives an acceptance regarding its proposed new leader receives the proposed round. If the received round is equal to the round of the most recent proposal sent, process *p* sets the element at position *q* in the array ack to 1 to record the acceptance.

Then, it inspects the ack array to check if all entries are 1. If so, *p* starts the commit phase by broadcasting its previously proposed new leader and the corresponding proposed round PropRound to all processes in v_p and reinitializes the ack array to zero.

A process *p* that receives an order to commit a new leader at position *q* from process *q*, simply sets the current leader to the proposed new leader and sets the current round to the proposed round.

5. CORRECTNESS

We can ensure the correctness of the algorithm by proving that it satisfies the two properties of the specification given in Section 4.

5.1 Safety

Theorem 1. *The algorithm described in Section 4 satisfies the safety condition of the specification (Property 1, Section 2): At any point in time, all processes connected the system never disagree on a leader.*

Proof. Either all processes remain in the start state or some process *p* receives the proposed leader as its leader. In the start state, the safety property holds since all processes are in the state in which a leader has not been elected. If some process *p* receives its leader by committing a proposed leader at a given position *q*, it must have received a Commit message from some process *q*; therefore, *q* must have received Accept messages regarding its proposal of a new leader from all processes in v_p including *p*. It follows from the last guarded command in Fig. 1 that, if process *p* has accepted the proposal of process *q*, it will not accept any other proposal for new leader, making it possible to commit at most single proposed leader. Therefore, process *p* either commits the process at position *q* as a new leader or ends up with position *q* by aborting the proposed new leader. Therefore safety property holds.

5.2 Liveness

Theorem 2. *The algorithm described in Section 4 satisfies the liveness condition of the specification (Property 2, Section 4): All processes should eventually progress to be in a state in which all processes connected to the system agree to the only one leader.*

Proof. By contradiction, a non-progress means that the new leader is not elected forever even though there is no leader; therefore, no Commit messages must be sent. Since the number of processes is finite, there must be at least one process whose id is the minimum value in v_p and that process eventually sends a Propose message. Call this process *p*. By the code in Fig. 1, we see that, to have no Commit message, each time *p* sends a Propose message, it should be rejected by other process. It follows that, in order to abort infinitely many Propose messages, other process *q* must reject the proposed messages infinitely often.

Propose messages are rejected either when the minimum id of v_p is greater than the id of the proposed leader or because of a Propose message already has been received (see Fig. 1).

The first case is ruled out because it implies that some process always considers that there is a process that is alive and whose

id is less that the id of proposed new leader. But by strong completeness of a failure detector it is contradiction.
The second case is also ruled out, because it implies that other process q sends infinitely many proposals of the other leader. But by eventual strong accuracy of a failure detector, the process q knows that there is a process whose id is less that its id. Therefore it is contradiction.

6. CONCLUDING REMARKS

We have presented a stable election protocol with a reliable failure detector in completely asynchronous systems. We have assumed our local failure detectors to be inaccurate and incomplete. With this approach, the leader election specification states explicitly that progress without violation of safety cannot always be guaranteed. In practice, our requirement for progress is weaker than that stated in the original specification of having a set of processes sharing the same leader.
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. This depends on the actual rate of a leader failures and on the capacity of the failure detectors to track such failures.
In [10], Chandra and Toueg note that failure detectors defined in terms of global system properties cannot be implemented. This result gives strength to the approach of relaxing the specification and of having a stable election protocol. 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.

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, Jan. 1982, pp. 49-59.
[3] H. Abu-Amara and J. Lokre, “Election in Asynchronous Complete Networks with Intermittent Link Failures.” IEEE Transactions on Computers, vol. 43, no. 7, 1994, pp.778-788.
[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, 1995, pp.147-156.
[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, 1996, pp.157-171.
[6] G. Singh, “Leader Election in the Presence of Link Failures,” IEEE Transactions on Parallel and Distributed Systems, vol. 7, no. 3, March 1996, pp.231-236.
[7] M. Fischer, N. Lynch, and M. Paterson, “Impossibility of Distributed Consensus with One Faulty Process,” Journal of the ACM,(32) 1985, pp. 374-382
[8] T. Chandra and S.Toueg, “Unreliable Failure Detectors for Reliable Distributed Systems,” Journal of ACM, vol.43 no.2, 1996, pp. 225-267.
[9] D. Dolev and R Strong, “A Simple Model For Agreement in Distributed Systems,” In Fault-Tolerant Distributed Computing, pp. 42-50. B. Simons and A. Spector ed, Springer Verlag (LNCS 448), 1987.
[10] T. Chandra, V. Hadzilacos and S. Toueg, “The Weakest Failure Detector for Solving Consensus,” Journal of ACM, vol.43 no.4, 1996, pp. 685-722.

Sung-Hoon Park
He received the B.S. in statistics and economics from Korea University, Korea in 1982 and also received M.S., Ph.D. in computer engineering & science from Indiana University, Bloomington, USA in 1993 and 2000. Since then, he has been a professor in the Dept. of Computer Engineering, ChungBuk National Univ. His main research interests include distributed system and algorithm.