On the Hardness of Leader Election in Asynchronous Distributed Systems with Crash Failures

Sung Hoon Park
Dept. of Computer Science & Engineering
Chungbuk National University, Cheongju, Korea

Yoon Kim*
Dept. of Computer Security
Korea National College of Rehabilitation and welfare, Pyongtaik , Korea

ABSTRACT

This paper is about the hardness of Leader Election problem in asynchronous distributed systems in which processes can crash but links are reliable. Recently, the hardness of a problem encountered in the systems is defined with respect to the difficulty to solve it despite failures: a problem is easy if it can be solved in presence of failures, otherwise it is hard [9]. It is shown in [9] that problems are classified as three classes: F (fault-tolerant), NF (Not fault-tolerant) and NFC (NF-completeness). Among those, the class NFC is the hardest problem to solve. It is also shown in [9] that the construction of Perfect Failure Detector (problem P) belongs to NFC. In this paper, we show that Leader Election is also one of NFC problems by using a general reduction protocol that reduces the Leader Election Problem to P. We use a formulation of the Leader Election problem as a prototype to show that it belongs to NFC.

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

1. INTRODUCTION

The Leader Election problem [1] requires that a unique leader be elected from a given set of processes. The problem has been widely studied in the research community [2,3,4,5,6]. One reason for this wide interest is that many distributed protocols need an election protocol.

In spite of such a wide research, the problem is known to be unsolvable in asynchronous distributed systems with crash failures. It follows from so-called FLP results [7]. The proof of the impossibility of Consensus in [7] assumes that it is impossible for a process to determine whether another process has crashed, or is just very slow. This assumption is widely cited as the "reason" for the impossibility result.

There are other problems that cannot be solved in asynchronous distributed systems with crash failures for the same intuitive reason that Consensus cannot be solved. In particular, the Leader Election problem cannot be solved if a crashed process cannot be distinguished from a slow process.

Recently, the hardness of a problem encountered in the systems was defined with respect to the difficulty to solve it despite failures: a problem is easy if it can be solved in presence of failures, otherwise it is hard [9]. According to the paper [9], problems are classified as three classes of problems, i.e. **F**, **NF** and **NFC** (NF-completeness). Among those problems, the problems belonging to **NFC** are defined to be the most difficult problems to solve in presence of failures. It is shown in [6] that the *Terminating reliable Broadcasting*, the *Non-Blocking Atomic commitment* and the construction of the *Perfect Failure Detector* (problem **P**) are the problems that belong to **NFC**, while the *Consensus* problem belongs to **NF** but neither to **F** nor to **NFC**.

It is shown in [10] that the *Leader Election* problem is at least as hard as the *Consensus* problem. An interesting question is then whether the *Leader Election* problem belongs to **NFC** or not. How much hard is the *Leader Election* problem to solve in asynchronous distributed systems? This is the topic of this paper, which focuses on the difficulty to solve this problem in reliable asynchronous distributed systems.

Determining that a problem Pb_1 is harder than a problem Pb_2 has a very important practical consequence, namely, the cost of solving Pb_1 cannot be less than that of solving Pb_2. To determine the hardness of the *Leader Election* problem, we used a *reduction protocol*. This means that if any algorithm A_1 that solves a problem Pb_1 can be transformed into an algorithm A_2 that solves a problem Pb_2, the problem Pb_1 is at least as hard as the problem Pb_2 ($Pb_1 \geq Pb_2$).

In this paper, we used a formulation of the Leader Election problem as a prototype. We reduced the prototype algorithm to solve the Leader Election problem into an algorithm to solve

* *Corresponding author. E-mail: ykim@hanrw.ac.kr*
Manuscript received Feb 9, 2005 ; accepted Mar 11, 2005

the construction of the *Perfect Failure Detector* problem and showed that the *Leader Election* problem belongs to NFC.

Actually, the main difficulty in solving a problem in presence of process crashes lies in the detection of crashes. To address this problem, Chandra, Hadzilacos and Toueg have introduced and investigated the notation of *Failure Detectors* [11]. Those are distributed *oracles* related to the detection of failures. A failure detector of a given class is a device that gives hints on set of processes that it suspects to have crashed. It is shown in [7] that the *Consensus* problem can be solved in the FLP model augmented with *Failure Detectors* satisfying some completeness and accuracy properties. A *Perfect Failure Detector* class (problem *P*) is central to decide whether certain problem is in NFC or not. A *Perfect Failure Detector* eventually suspects each crashed process in a permanent way (*Strong Completeness*), and never suspects a process before it crashes (*Strong Accuracy*).

The rest of the paper is organized as follows. In Section 2, we describe our system model and definitions. In Section 3, this paper presents an algorithm to solve *Leader Election* and specifies the properties of the *Leader Election* problem. In Section 4, this paper studies the reduction protocol that transforms an algorithm to solve the Leader Election problem into the algorithm to solve the *Perfect failure Detector* problem and shows that *Leader Election* is one of NF-complete problems with respect to reliable asynchronous distributed systems. Finally, Section 5 summarizes the main contributions of this paper and discusses related and future works.

2. MODEL AND DEFINITIONS

2.1 Asynchronous Distributed Systems

Our model of asynchronous computation with crash failures is the one described in [7]. We call FLP such a system model. In the following, we only recall some informal definitions and results that are needed in this paper. We consider a distributed system composed of a finite set of n processes $\Omega = \{1,2,...,n\}$ completely connected through a set of channels. Communication is by message passing, asynchronous and reliable. A process fails by simply stopping the execution (*crashing*), and the failed process does not recover. A correct process is the one that does not crash. Byzantine failures are not considered. At least one process is correct in the systems.

Asynchrony means that there is no bound on communication delays or process relative speeds. It can not distinguish a slow processor from a crashed node. Between any two processes there exist two unidirectional channels. Processes communicate by sending and receiving messages over these channels: there is no shared memory. The channels are nonfaulty: they do not lose, generate, or garble messages. The channels need not be to FIFO. The state of a channel is the set of messages that have been sent along the channel but not yet received. A reliable channel ensures that a message, sent by a process i to a process j, is eventually received by j if both are correct (i.e. do not crash). To simplify the presentation of the model, it is convenient to assume the existence of a discrete global clock. This is merely a fictional device inaccessible to processes. The range of clock ticks is the set of natural numbers.

A process has a set of states, one of which is denoted the initial state. The state of a process i consists of the values of all internal variables of the process. A global state of the system is a set of process and channel states. An initial global state is the global state in which each process state is an initial state and each channel state is the empty set.

An *event* e is an action that maps the global state of the system from Σ to Σ' such that Σ' differs from Σ in the local state of exactly one process i and the state of at most one channel incident on the process i. In this case, we say that e is an event of process i. A history of a process i is a sequence of events $h_i = e_i^0 e_i^1 ... e_i^k$, where e_i^k denotes an event of process i occurred at time k. Histories of correct processes are infinite. If not infinite, the history of the process i terminates with the event CRASH_i^k (process i crashes at time k).

Definition 1 *A run of the system is an infinite sequence of global states of the system:* $r = (\Sigma_0, \Sigma_1, \Sigma_2, \ldots)$ *where* Σ_0 *is an initial global state and there exists a sequence of events* $(e_0, e_1, e_2, \ldots)$ *such that for all* $i \geq 0$, $\Sigma_{i+1} = e_i(\Sigma_i)$.

We specify properties of systems using a predicate logic over global states and a linear-time temporal logic over (infinite) suffixes of runs ([16]).
We use the following as the meaning of the two temporal logic modal operators $\Diamond$ and $\models$.

Definition 2 *Let* $s = (\Sigma_0, \Sigma_1, \Sigma_2, \ldots)$ *be a suffix of a run, let* φ *be a predicate, and let P be a temporal logic formula.* Then,

- $(s, k) \models \varphi$ *iff* $\Sigma_k \models \varphi$.
- $(s, k) \models \Diamond P$ *iff* $\exists j \geq k: (s, j) \models P$
- $(s, k) \models \models P$ *iff* $\forall j \geq k: (s, j) \models P$

Furthermore, we abbreviate $(s, 0) \models P$ *as* $s \models P$.

We refer to the pair (s, k) as the *prefix* of the infinite sequence of states s.
We use the following definition of a protocol:

Definition 3 *A protocol is a many-to-many mapping from a prefix of a run to a global state.*

Thus, repeatedly applying a protocol to an initial global state will generate a run. This represents the execution of a (possibly non-deterministic) program over time. We apply this procedure of executing a protocol A to extend a prefix (s, k) to $(s', k' > k)$, meaning that $(s, k) = (s', k)$ and that for each state Σ'_l: $k < l \leq k'$: $\Sigma'_l \in A((s', l-1))$.

One protocol A can be reduced into B to generate a single run as follows: given a prefix of a run, one of the protocols is chosen non-deterministically and fairly to generate the next global state.

2.2 Failures and Failure detectors

We define the following events associated with crashing, detecting failures, and ceasing to detect failures:

- $CRASH_i$ denotes the event whereby process i crashes.
- $FAILED_i(j)$ denotes the event whereby process i detects the failure of process j (or "suspects" j).
- $stop_i(j)$ denotes the event whereby process i stops suspecting process j ("stops" the suspicion of j). This event is executed when process i receives a message from process j while i suspects j.

We define the Boolean predicates $CRASH_i$ and $FAILED_i(j)$ as follows:

- $\forall i,j$: $CRASH_i$ and $FAILED_i(j)$ are false in an initial global state.
- $CRASH_i$ is true in the global state resulting from $CRASH_i$ and in every global state thereafter.
- $FAILED_i(j)$ is true in the global state resulting from $FAILED_i(j)$ and in every global state until $stop_i(j)$ is executed.

$CRASH_i$ is stable by definition but $FAILED_i(j)$ is not.

We assume that crashes can occur spontaneously, and that there is no restriction on the set of processes that may crash at any time in any run. Hence, if $(r, k) \models \neg CRASH_i$ then the prefix (r, k) can be extended to $(r', k+1)$ such that $(r', k+1) \models CRASH_i$

We define a *failure detector* as a set of runs satisfying properties that relate crash events to failed events. A failure detection protocol is a protocol that generates those runs in a failure detector.

Failure detectors are abstractly characterized by completeness and accuracy properties [11]. These problems have been defined previously in [7,12,13]. Completeness characterizes the degree to which crashed processes are permanently suspected by correct processes. Accuracy restricts the false suspicions that a process can make. In [5], Chandra and Toueg define the *Perfect Failure Detector* as follows.

Perfect Failure Detector (*P*) The problem of building a perfect failure detector (problem P) consists in designing a protocol that provides processes with a list of suspects with such that the following two properties are satisfied [5]:

- **Strong Completeness:** $\forall r : r \models \forall i : \nvdash (CRASH_i \Rightarrow \forall j : \Diamond(CRASH_j \vee \nvdash FAILED_j(i)))$ (Eventually every process that crashes is permanently suspected by every correct process).

- **Strong Accuracy:** $\forall r : r \models \forall i, j : \nvdash (FAILED_i(j) \Rightarrow CRASH_j)$ (No process is suspected before it crashes).

A process i queries the perfect failure detector by invoking P-QUERY which returns a list of suspects.

Failure detectors characterized by Strong Accuracy are reliable: no false suspicions are made. Otherwise, they are unreliable. For example, failure detectors of P are reliable, whereas failure detectors of S are unreliable.

2.3 Reducibility and Transformation

The notation of *problem reduction* first has been introduced in the problem complexity theory [15], and in the formal language theory [14]. It has been also used in the distributed computing [9,12,13]. We consider the following definition of problem reduction.

An algorithm A *solves* a problem Pb_1 if every run of it satisfies the specification of the problem Pb_1. A problem Pb_1 is said to be solvable with the algorithm A if there is an algorithm that solves Pb_1 with the algorithm A. A problem Pb_1 is said to be reducible to a problem Pb_2 (denoted $Pb_1 \geq Pb_2$), if the protocol A that solves the problem Pb_2 can be transformed into any protocol to solve Pb_1. If Pb_1 is not reducible to Pb_2, we say that Pb_1 is harder than Pb_2 (denoted $Pb_1 > Pb_2$).

The problem of transforming Pb_1 into Pb_2 belongs to F. This ensures the closure of F and NF with respect to problem reductions. As we mentioned in the introduction, $Pb_1 \geq Pb_2$ means that the problem Pb_1 is at least as hard as the problem Pb_2 in presence of process failures. If the Pb_2 is not solvable or requires some additional assumptions to solve it, then the Pb_1 is also unsolvable under the same assumptions of Pb_2 or requires at least as many assumptions as those of Pb_1 to be solved.

If $Pb_1 \geq Pb_2$ and $Pb_2 \geq Pb_1$, then Pb_1 and Pb_2 are said to be equivalent (i.e. denoted by $Pb_1 \equiv Pb_2$). As an example, let us consider the FIFO Broadcast Problem (FB) and the Casual Broadcast Problem (CB): it is shown in [18] that these problems are equivalent, FB≡CB. It is also shown in [9] that the Non-Blocking Atom Commitment problem (NBAC), the construction of the Perfect Failure Detector problem (P) and the Terminating Reliable Broadcast problem (TRB) are equivalent problems, NBAC ≡ P ≡TRB.

2.4 Problem Classes

We are interested in the set of problems that can be solved in asynchronous systems. A problem is specified by a set of properties. A protocol A solves a problem Pb in a system M, if each run of A in M satisfies the properties that specifies the problem Pb.

Recently, like a **NP-completeness** theory in sequential computing, it was shown in [9] that the hardness of problems encountered in a distributed computing was defined with respect to the difficulty to solve it despite of failures. A problem is easy if it can be solved in presence of failures, otherwise it is hard. According to the paper [9], those problems are defined following three sets of problems:

- **F** : the set of problems that can be solved despite arbitrarily many number of process crashes in the FLP model (**F** stand for Fault-tolerant)
- **NF** : the set of problems that can be solved when there is no process crash in the FLP model (NF stand for Not Fault tolerant).
- **NF-Complete**: A distributed computing problem Pb_1 belongs to the class **NFC** (the class of NF-complete problems) (1) if it belongs to NF and (2) if it is at least

as hard as any problems belonging to **NF** problems, i.e., if the following property is satisfied: $\forall Pb_2 \in NF \quad (Pb_1 \geq Pb_2)$.

Reliable Broadcast (RB) and Casual Broadcast (CB) are the classical distributed problems belonging to **F**. Terminating Reliable Broadcast (TRB) and Consensus (CONS) are the typical problems that belong to NF but do not belong to **F**. It is also shown in [9] that the Terminating reliable Broadcasting problem, the Non-Blocking Atomic commitment problem (NBAC) and the construction of perfect failure detector problem (P) belong to **NFC**, while CONS belongs to NF but neither to F nor to NFC. This is illustrated on Figure 1. More generally, this figure depicts the structure of the class NF. Among those problems, the ones belonging to **NFC** are defined to be the most difficult problems to solve in presence of failures.

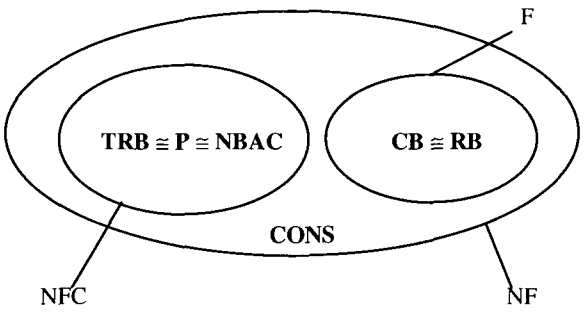

Fig. 1: A Hierarchy of Problems in the FLP Model.

3. THE LEADER ELECTION PROBLEM

Leader Election is an important problem to solve for the construction of fault tolerant systems. It is closely related to the primary-backup approach (since choosing a primary replica is like electing a leader), an efficient form of passive replication. It is also closely related to group communication [18], which (among other uses) provides a powerful basis for implementing active replication.

3.1 Specification

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

More formally, let $Leader_i$ be a predicate that indicates that process i considers itself the leader. The Leader Election Problem is specified by the following two properties. One is for safety and the other is for liveness.

The safety requirement asserts that all the nodes connected to the system never disagree on the leader when the nodes are in the state of a normal operation.

Safety : $\forall r : r \models \vdash (\exists i : Leader_i \Rightarrow \forall j : j \neq i : \neg Leader_j)$

The liveness requirement asserts that all the processes should eventually progress to be in the state of normal operation in which all processes connected to the system agree to the only one leader.

Liveness : $\forall r : r \models \vdash (\neg Leader_j \Rightarrow \Diamond \exists i : Leader_i)$

An election protocol is a protocol that generates runs that satisfy the Leader Election specification.

3.2 Leader Election protocol

We use a formulation of the Leader Election problem as a prototype to verify that it belongs to NFC.

Theorem 3 *The following LE-ELT algorithm solves the leader election problem.*

LE-ELT algorithm:

- Each process has a unique ID number that is known by all processes.
- The leader is initially the process with the lowest ID number, i.e. process *1*. The role of a leader is rotated one by one in ascending order with ID whenever a current leader crashes. For example, in the case of a leader process k crash, the first candidate for new leader is process $k+1$ and the second candidate is process $k+2$ and so on. If the leader process n crashes, then process *1* is the first candidate for the new leader. Therefore, the *priori* of a process is changed at every election and decided relatively depending on the ID of the crashed leader.
- If process i detects that the leader k crashes, it broadcasts this information to all processes by using the primitive ***LE-ELT***-BROADCAST(k). Upon receiving such a message, every receiver detects the failure of all processes between the ex-leader and itself.
- When process j detects the failure of all processes that are in between those processes, it becomes the leader and notifies to all other processes that he is the new leader. All processes in the set receive this notification by using the primitive ***LE-ELT***-DELIVER(j).

Proof. The LE-ELT algorithm satisfies the two conditions.

Safety : Proof by contradiction

Consider a run r in which two leaders are to be elected. That means that $\forall r : r \models\vdash (\exists i : Leader_i \Rightarrow \forall j : j \neq i : \neg Leader_j)$ is false.

Then, we can state it formally as follows.

$\forall r : r \models \vdash (\exists i : Leader_i \Rightarrow \forall j : j \neq i : \neg Leader_j)$ is false

implies $\exists r : r \models \Diamond(\exists i,j : j \neq i : Leader_i \wedge Leader_j)$ (1)

Let $s = (\Sigma_0,\Sigma_1,\Sigma_2, \ldots)$ be a suffix of such a run r.

(1) implies s $\models \Diamond\ (\exists i,j : j \neq i : \text{Leader}_i \wedge \text{Leader}_j)$ (2)

To be a leader, each of them should have detected the failures of all the processes between the ex-leader k and itself before declaring itself to be a leader. Let $\Omega_{ki} = \{ k+1, \ldots, i-1 \}$ be the set of processes that are in between process k and process i. Then,

(2) implies s $\models \Diamond\ (\exists i,j : (\forall m: m\in\Omega_{ki} : \neg\text{CRASH}_i \wedge \text{CRASH}_m) \wedge$

$(\forall n: n\in\Omega_{kj} : \neg\text{CRASH}_j \wedge \text{CRASH}_n)\ )$ (3)

But the *prior* of i is higher than that of j or the converse is true. That means that the process j belongs to Ω_{ki} or the process i belongs to Ω_{kj}. Thus,

(3) implies s $\models \Diamond\ (\ \exists i,j : (\forall m,n: m\in\Omega_{ki},\ n\in\Omega_{kj} : (\neg\text{CRASH}_i \wedge \text{CRASH}_m \wedge$

$\neg\text{CRASH}_j \wedge \text{CRASHn}) \wedge (\ j\in\Omega_{ki} \vee i\in\Omega_{kj}\ )$

implies s $\models \Diamond\ (\ \exists i,j : (\forall m,n: m\in\Omega_{ki},\ n\in\Omega_{kj} : (\neg\text{CRASH}_i \wedge \text{CRASH}_m \wedge i\in\Omega_{kj}\ ) \vee$

$(\neg\text{CRASH}_j \wedge \text{CRASH}_n \wedge j\in\Omega_{kj}\ )$ (4)

But the predicate $i\in\Omega_{kj}$ implies that the *process i* crashed.

(4) implies s $\models \Diamond\ (\ \exists i,j: (\neg\text{CRASH}_i \wedge \text{CRASH}_i) \vee (\neg\text{CRASH}_j \wedge \text{CRASH}_j)\ )$

implies $\exists k \geq 0$: (s, k) $\models$ $(\exists i,j: (\neg\text{CRASH}_i \wedge \text{CRASH}_i) \vee (\neg\text{CRASH}_j \wedge \text{CRASH}_j)\ )$.

This is a contradiction.

Liveness : Proof by Contradiction

When a leader process j crashes, some processes that have detected it eventually broadcast the message to inform all processes of the failure of the leader. All the processes that received the message instantly start an election protocol. Consider a run r in which there is no leader elected after terminating the election protocol, i.e. $\forall r : r \models \nvdash (\neg\ \text{Leader}_j \Rightarrow \Diamond\exists i : \text{Leader}_i\ )\ )$ is false.

We can state it more formally as follows.

$\exists r : r \models \neg\nvdash (\ \text{Leader}_j \vee \Diamond\exists i : \text{Leader}_i\ )$

implies $\exists r : r \models \Diamond(\neg\text{Leader}_j \wedge \nvdash \forall i : \neg\text{Leader}_i\ ))$ (5)

Let $s = (\Sigma_0,\Sigma_1,\Sigma_2,\ldots)$ be the suffix of such a run r, then

(5) implies s $\models \Diamond(\neg\text{Leader}_j \wedge \nvdash \forall i : \neg\text{Leader}_i\ ))$ (6)

To be a leader, each process in the system should detect the crash of the processes that are in between the ex-leader and itself. The predicate ($\nvdash \forall i$:$\neg\text{Leader}_i$) implies that there has been no process that detected the crash of all processes in the set Ω, i.e. $\nvdash\ (\ \forall i,m : m\in\Omega_{ji} : \neg\ (\neg\text{CRASH}_i \wedge \text{CRASH}_m)$.

So, (6) implies s $\models \Diamond(\neg\text{Leader}_j \wedge \nvdash \forall i,m : m\in\Omega_{ji} : \neg\ (\neg\text{CRASH}_i \wedge \text{CRASH}_m))$

implies s $\models \Diamond\nvdash\ (\neg\text{Leader}_j \wedge \forall i,m: m\in\Omega_{ji} : \text{CRASH}_i \vee \neg\text{CRASH}_m\ )$

implies s $\models \Diamond\nvdash\ (\neg\text{Leader}_j \wedge \forall i : \text{CRASH}_i\ )$ or

s $\models \Diamond\nvdash\ (\neg\text{Leader}_j \wedge \forall i,m: m\in\Omega_{ji} : \neg\text{CRASH}_m\ )$

implies $\exists k\geq 0,\ \forall k' \geq k$: (s, k') $\models$ $(\neg\text{Leader}_j \wedge \forall i : \text{CRASH}_i\ )$ or

$\exists k \geq 0,\ \forall k' \geq k$: (s, k') $\models$ $(\neg\text{Leader}_j \wedge \forall i,m : m\in\Omega_{ji} : \neg\text{CRASH}_m\ )$ (7)

But $\exists k\geq 0,\ \forall k' \geq k$: (s, k') $\models (\neg\text{Leader}_j \wedge \forall i : \text{CRASH}_i\ )$ is false by the initial condition that at least one process is correct in the system. We can prove inductively that $\exists k \geq 0,\ \forall k' \geq k$: (s, k') $\models (\neg\text{Leader}_j \wedge \forall i,m : m\in\Omega_{ji} : \neg\text{CRASH}_m\ )$ is also false. Therefore (7) is false. This is a contradiction.

The process with high *prior* eventually wins the election and declares that it is the new leader. This means that there is at least one process that detected the failures of all the processes between the ex-leader and itself.

4. LEADER ELECTION BELONGS TO NFC

This section shows that *Leader Election* is one of *NF-completeness* problems. It is shown on [9] that the construction of *Perfect Failure Detector* (problem *P*) that is one of the *NF* problems belongs to *NFC*. To show that *Leader Election* is also one of NF-complete problems, we should verify that the Leader Election problem is as hard as or harder than the construction of Perfect Failure Detector problem, i.e. *LE-ELT* $\geq$ *P*. To attain this goal, we design a protocol that assuming *LE-ELT*, solves *P*. Such a protocol is called a *reduction protocol*.

4.1 From Leader Election to P

Protocol 1 shows a reduction protocol that transforms the protocol solving the Leader Election problem into a protocol solving the construction of Perfect Failure Detector problem *P*.

A process i is composed of two tasks T_1 and T_2. Task T_2 is used to answer the queries of the upper layer when this layer invokes *P-query*. T_2 returns it the current value of $suspected_i$. The process i manages a local variable $suspected_i$ that is initialized $\varnothing$.

In *task* T_1, each process executes an infinite sequence of rounds, each round simulating and synchronizing executions of a Leader Election protocol solving instances of Leader Election.

During a round, process i considers all processes belonging to the set regardless of its failure. So, the process i solves an instance of $\lfloor n \rfloor$ on the LE-ELT algorithm in every round.

At every instance of a round, the process i sends the message of leader failure to all processes and waits for the termination of the Leader Election protocol. Upon receipt of the result informing the new leader, process i compares it to the first candidate that is decided on the rotating leader policy. If the ID of newly elected leader is not equal to the ID of the first candidate, then process i concludes that the first candidate process has crashed. For example, when the leader process j-1 crashes, the process j is the first candidate for a leader. If it is not crashed, the process j might have been elected as the next new leader.

Due to the liveness property of ***LE-ELT*** algorithm, the result of election is notified to process i. In this case, the result of this ***LE-ELT*** instance is necessarily the ID of process j. The primitives corresponding to this instance of the LE-ELT algorithm invoked by process i are denoted ***LE-ELT***-BROADCAST(p) and ***LE-ELT***-DELIVER(j).

```
% A process i execute %
suspected_i ← ∅; r_i ← 0;
cobegin
  task T1 : while true do % (Infinite) sequence of synchronized rounds %
                p ← 1; r_i ← r_i + 1;
                while p++ ≤ n do
                    LE-ELT-BROADCAST_i(p);
                % leader_candidate is the first candidate for new leader %
                leader_candidate := (p + 1) mod n;
                LE-ELT-DELIVER_i( new_leader);
                If new_leader ≠ leader_candidate then
                                suspected_i ← suspected_i ∪ { leader_candidate }
                end-if
            end-while
        end-while
  task T2 : when P-Query do return(suspected_i);
coend
```

Protocol 1: LE-ELT ≥ P

Theorem 4.1 *Protocol 1 builds a perfect failure detector.*

Proof. The algorithm in Protocol 1 satisfies the two conditions.

Strong Completeness: Proof by contradiction

Consider a run r in which some processes that crash are permanently not suspected by some correct process. This means that the property of strong completeness is false, i.e. $\forall r : r \models \forall i : \vdash (CRASH_i \Rightarrow \forall j : \Diamond(\neg CRASH_j \Rightarrow \vdash FAILED_j(i)))$ is false.

We can state it more formally as follows.

$\neg(\forall r : r \models \forall i : \vdash (CRASH_i \Rightarrow \forall j : \Diamond(\neg CRASH_j \Rightarrow \vdash FAILED_j(i))))$

implies $\exists r : r \models \exists i : \Diamond(\neg(\neg CRASH_i \vee \forall j : \Diamond(CRASH_j \vee \vdash FAILED_j(i))))$

implies $\exists r : r \models \exists i : \Diamond(CRASH_i \wedge \exists j : \vdash (\neg CRASH_j \wedge \Diamond\neg FAILED_j(i)))$ (8)

Let $s = (\Sigma_0, \Sigma_1, \Sigma_2, \ldots)$ be the suffix of such a run r. Then,

(8) implies $s \models \exists i : \Diamond(CRASH_i \wedge \exists j : \vdash (\neg CRASH_j \wedge \Diamond\neg FAILED_j(i)))$ (9)

The predicate $\Diamond\neg FAILED_j(i)$ implies that j never has suspected the failure of i, i.e., $(i \notin suspect_j)$ is true.

(9) implies $s \models \exists i : \Diamond(CRASH_i \wedge \exists j : \vdash (\neg CRASH_j \wedge (i \notin suspect_j)))$ (10)

The predicate $(i \notin suspect_j)$ implies that j has delivered i as a new leader at every round.

(10) implies $s \models \exists i : \Diamond(CRASH_i \wedge \exists j : \vdash (\neg CRASH_j \wedge \Diamond(\text{LE-ELT-Deliver}(new\text{-}leader) = i)))$ (11)

The fact that process j has delivered process i as a new leader implies that process i is not crashed.

(11) implies $s \models \exists i : \Diamond(CRASH_i \wedge \exists j : \vdash (\neg CRASH_j \wedge \neg CRASH_i))$

implies $s \models \exists i : \Diamond\vdash (CRASH_i \wedge \exists j : (\neg CRASH_j \wedge \neg CRASH_i))$

implies $s \models \exists i,j : \Diamond(CRASH_i \wedge \neg CRASH_i)$

implies $\exists k \geq 0 : (s, k) \models \exists i,j : (CRASH_i \wedge \neg CRASH_i)$. This is a contradiction.

Strong Accuracy: Proof by contradiction

Consider a run r in which some processes are suspected before crash by some correct processes. That means that the property of strong accuracy, i.e. $\forall r : r \models \forall i,j : \vdash (FAILED_i(j) \Rightarrow CRASH_j)$ is false.

We can state it more formally as follows.

$\exists r : r \models \exists i,j : \Diamond(FAILED_i(j) \wedge \neg CRASH_j)$ (12)

Let $s = (\Sigma_0, \Sigma_1, \Sigma_2, \ldots)$ be the suffix of such a run r, then

(12) implies $s \models \exists i,j : \Diamond(FAILED_i(j) \wedge \neg CRASH_j)$ (13)

The fact that the predicate $FAILED_i(j)$ is true means that the process j belongs to the $suspect_i$.

So, (13) implies $s \models \exists i,j : \Diamond((j \in suspect_i) \wedge \neg CRASH_j)$ (14)

The fact that $(j \in suspect_i)$ is true implies that the predicate $\neg\Diamond(\text{LE-ELT-Deliver}_i(new\text{-}leader) = j)$ is true, that means that the process i has never delivered the process j as a new leader.

So, (14) implies s $\models \exists i,j$: $\Diamond(\neg\Diamond$(LE-ELT-Deliver$_i$(*new-leader*)$=j)\wedge\neg$CRASH$_j$)

implies s $\models \exists i,j$: $\Diamond(\not\models$ (LE-ELT-Deliver$_i$(*new-leader*)$\neq j)\wedge\neg$CRASH$_j$)

implies s $\models \exists i,j$: $\Diamond\not\models$ (CRASH$_j\wedge\neg$CRASH$_j$)

implies $\exists k\geq 0$: $(s, k) \models \exists i,j$: (CRASH$_j\wedge\neg$ CRASH$_j$). This is a contradiction.

So, combined with the fact that the problem P belongs to the class NFC, it is possible to conclude that Leader Election and P are equivalent problems, LE-ELT $\equiv$ P.

5 . CONCLUSION

In this paper, we reduced an algorithm to solve the Leader Election problem into the algorithm to solve the construction of perfect failure detector problem and showed that the Leader Election problem belongs to the NF-completeness problem that is most difficult problem to solve in asynchronous systems.

To be our knowledge, it is however the first time that the hardness of Leader Election problem is discussed in asynchronous systems. The fact that the Leader Election problem is at least as hard as the construction of Perfect Failure Detector problem has a very important consequence, namely, the cost of solving the Leader Election problem cannot be less than that of solving the construction of Perfect Failure Detector problem.

Actually, the main difficulty in solving such a problem in presence of process crashes lies in the detection of crashes. Given the results of the previous section, it is clear that any problem whose specification implies Leader Election is at least as hard as the construction of Perfect Failure Detector problem. For example, the asynchronous version of the Primary Backup problem ([8]) requires that there is no more than one primary server at any time and that there is always eventually a primary server, and so *Primary Backup* implies Leader Election. In fact, it is easy to implement Primary Backup using a Perfect Failure Detector since the construction of *Perfect Failure Detector* is at least as hard as that of *Primary Backup*.

There are also problems that do not resemble Leader Election that belong to the NF-complete problem. The Terminating Reliable Broadcast problem is one example [11,18]. In this problem, if a correct process sends a message, then that message is eventually received by all other correct processes; if a faulty process sends a message, then all correct processes eventually receive the same message. In [9], it is claimed that the Terminating Reliable Broadcast and the construction of Perfect Failure Detector are equivalent problems that belong to NFC. Therefore, Leader Election is also equivalent to Terminating Reliable Broadcast since both are equivalent to the construction of Perfect Failure Detector problem.

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] Fischer M.J., Lynch N. and Paterson M.S Impossibility of Distributed Consensus with One Faulty Process. *Journal of the ACM*, 32(2):374-382, April 1985.

[8] N. Budgiraja, K. Marzullo, F.B.Schneider, and S. Toueg. Primary-backup protocols: lower bounds and optimal implementations. In *Proceedings of the Third IFIP Working Conference on Dependable Computing for Critical Applications*. IFIP 10.4, September 1992.

[9] Eddy Fromentin, Michel R RAY, Frederic TRONEL. On Classes of Problems in Asynchronous Distributed Systems. In *Proceedings of Distributed Computing Conference*. IEEE 10.4, June 1999.

[10] L. Sabel and K. Marzullo. Simulating fail-stop in asynchronous distributed systems. In *proceedings of the Thirteenth Symposium on Reliable Distributed Systems, pages* 138-147. IEEE, Oct. 1994.

[11] Chandra T. and Toueg S. Unreliable Failure Detectors for Reliable Distributed Systems. *Journal of the ACM*, 43(1):225-267, March 1996.

[12] Guerraoui R. Revisiting the Relationship Between Non-Blocking Atomic Commitment and Consensus. *Proc. of the 9th Int. Workshop on Distributed Algorithms (WDAG)*, Springer-Verlag, LNCS 972, pp. 87-100. September 1995.

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

[14] Hopcroft J.E. and Ullman J.D. *Introduction to Automata Theory, Languages and Computation*. Addison Wesley, Reading, Mass., 418 pages, 1979.

[15] Garey M.R. and Johnson D.S. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Freeman W.H & Co, New York, 340 pages, 1979.

[16] A. Pnueli. The temporal logic of programs. In *Proceedings of the 18th Annual Symposium of Foundations of Computer Science*. ACM, November 1977.

[17] S. Mullender, editor. *Distributed Systems, chapter 4*, ACM Press frontier series. Addison Wesley, second edition, 1993.

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

Sung-Hoon Park

He received the B.S in economics and statistics from Korea university in 1982, M.S in Computer science from Indiana University USA in 1991 and received Ph.D. in computer science and engineering from Korea university in 2000. In 2004, he has been an associate professor in chungbuk national university Korea. His main research interests include distributed system, mobile computing and theory of computation.

Yoon Kim

He received the B.S. degree in Mechanical Engineering from Hanyang University, Seoul, Korea in 1982 and M.S. degrees in Computer Science from Stevens Institute of Technology, New Jersey, U.S.A. in 1988 respectively. Since then he was with Unicom Technology as a system engineer in U.S.A. Currently, he is an assistant professor of Information Security at Korea National College of Rehabilitation and Welfare. His main research interests include mobile computing and distributed system.