Unreliable failure detectors for asynchronous distributed systems

David Baelde
David.Baelde@ens-lyon.fr
E.N.S Lyon

directed by

Franck Petit
Petit@laria.u-picardie.fr
LaRIA

Vincent Villain
Villain@laria.u-picardie.fr
LaRIA

September 20, 2003

1 Introduction

Distributed computing is very attractive, but comes with new problems : information losses, overflow, or breakdowns. Most often, they are neglected. Indeed, it has been shown that the Consensus (a fundamental problem which requires that the processes agree on a common value) is unsolvable in a realistic computing model, i.e. completely asynchronous with possible crash failures [FLP85]. Intuitively, in an asynchronous environment, a process cannot decide if a component is either crashed or very slow.

Several approaches were designed to "bypass" that impossibility. One of them is self-stabilization, studied at LaRIA, which deals with transient faults. The principle is to design algorithms which can be executed from any initial state, and eventually work according to its specification. Snap-stabilization is stronger : from any initial step, the algorithm always behaves according to its specification. The first snap-stabilized algorithms were designed at LaRIA.

Another approach, which we are going to study, cope with definitive (crash) failures. Ideally, a black box should be attached to each process to indicate precisely the failures of the network. This black box is called a failure detector. But, the result of [FLP85] implies that it is impossible to implement such a perfect failure detector. That is why Chandra and Toueg introduces in [CHT96] the notion of unreliable failure detectors. Even if such detectors are still impossible to implement, practically, this approach allows to implement semi-algorithms. Theoretically, this approach also allows to introduce a hierarchy of the unreliable

failure detectors according to their power to solve classical problems. Chandra and Toueg introduced this hierarchy in [CHT96].

Later, Cho and Park have shown that the Leader Election was strictly harder to solve than the Consensus [CP02]. This last result is surprising. Consensus and Leader Election seem to be very close. Actually, one can think that all of these fundamental problems are equivalent. I had to understand the previous results and the differences between the problems. Focused on the Leader Election, less studied that the other problems, we had before anything to find a relevant definition for the problem, and then study it and check the not much formal results of Cho and Park. Finally, we studied several leader elections, which differences and main specificities will be presented.

In Section 2, we define the computing model. Then, in Section 3, we remind the main results. In the next section (Section 4), we study two close definitions of the Leader Election, and then the consequences of the addition of the stability property to these definitions. We conclude this report with Section 5.

2 The model

2.1 Asynchronous distributed systems

The system consists of a set of n processes $\Pi = \{p_1, p_2, \ldots, p_n\}$. Every pair of processes is connected by a reliable communication channel. We assume the existence of a global clock, which ticks belongs to $\mathbb{N}$. The processes cannot access to the clock, it is an abstract device.

The system is *asynchronous*. It means that no assumption is made on processes speeds and communication speeds. Moreover, there is no hypothesis on the order of messages delivery.

Note that the model is unchanged if we suppose that communication channels are lossy, but that a message sent infinitely many times is eventually delivered. Thus, the most inconvenient thing here is that a new process can't join the computing during the run. Meanwhile, the impossibility results are still valid when we suppose that a process can join the computing at any time — or any other addition to the computing model.

2.2 Crash failures

We consider systems where processes can crash. In order to describe these failures we use a failure pattern F.

$$F : \mathbb{N} \rightarrow \mathcal{P}(\Pi)$$

$$p \in F(t) \Leftrightarrow p \text{ is crashed at time } t$$

The following properties are assumed :

$\forall t \in \mathbb{N}, F(t) \subset F(t+1)$	A crash is definitive
$\forall t \in \mathbb{N}, \lvert F(t)\rvert < n$	There is at least one correct process

We denote by $correct(F, t)$ the set of correct processes at time t in the failure pattern F, and $correct(F)$ the set of correct processes at any time in F.

$$\begin{aligned} correct(F, t) &= \Pi - F(t) \\ correct(F) &= \cap_{t \in \mathbb{N}} correct(F, t) \end{aligned}$$

2.3 Unreliable failure detectors

A failure detector consists of $|\Pi|$ *failure detector modules*. Each module helps a process, giving him clues about processes that may have crashed. A module may lead its process to suspect a correct process, or trust a crashed process. That is why we call it an unreliable failure detector.

Formally, a failure detector is a mapping from failure patterns to *failure detector history* sets. In other words, for a single failure pattern, there can be many different failure detections.

In [CT96],a failure detector history is a mapping $H : \Pi \times \mathcal{T} \rightarrow \mathcal{P}(\Pi)$, such that $H(p, t)$ contains the processes that the failure detector module used by p suspects at time t. But this is not true in every case. In every proof in that article, no assumption is made on the type of the clue given by the failure detector module. For example, the module could give a boolean formula, such as "$p_1 \vee \neg(p_2 \wedge p_3)$" [1] .

2.4 Formal definitions

Let Π be a set of processes. We note $n = |\Pi|$.

Environment
An environment is a set of failure patterns. We'll denote by $\mathcal{E}_k$ the set of every failure pattern where there is strictly less than k failures. $\mathcal{E}_1$ contains the failure pattern where there is no crash, $\mathcal{E}_n$ contains every failure patterns.

Step
A step of the algorithm A is uniquely defined by the tuple (i, m, d, A), where i is the ID of the process that takes the step, m is the message he receives (λ if there is no incoming message), and d is the value output by i's failure detector module. We'll say that a step s is applicable to a configuration c if $m = \lambda$ or m is in the current message buffer. We generalize by induction : a sequence of step is applicable to a configuration c if the sequence is empty or if it is of the form h::t where the step h is applicable to the configuration and t is applicable to $h(c)$.

Run
A run of the algorithm A is a tuple $\langle F, H_{\mathcal{D}}, I, S, T \rangle$, where F is a failure pattern,

[1] This example was given in [CHT96], where a reduction $T_{\mathcal{D} \rightarrow \Omega}$ is built for any $\mathcal{D}$ that solves the Consensus, possibly $\mathcal{D} \notin \mathcal{P}(\Pi)^{\Pi \times \mathcal{T}}$.

$H_{\mathcal{D}} \in \mathcal{D}(F)$, I is an initial configuration of A, and $T \subseteq \mathbb{N}$ is a list of increasing time values, indicating when does the steps belonging to the list S occur. We must have $|S| = |T|$, and S applicable to I.
The run is said to be partial if T is finite. When we simply name it a run, T is infinite and the run guarantee that every correct process in F takes an infinite number of steps and eventually receives any message sent to it.

2.5 Reductions

We will say that $\mathcal{D}'$ reduces to $\mathcal{D}$, and note $\mathcal{D} \succeq \mathcal{D}'$, when there exists a distributed algorithm $T_{\mathcal{D}\to\mathcal{D}'}$ which emulates $\mathcal{D}'$, when running with the help of $\mathcal{D}$. We say that $\mathcal{D}$ and $\mathcal{D}'$ are equivalent, and note $\mathcal{D} \cong \mathcal{D}'$, when each of the detectors is reducible to the other. Finally, we will write $\mathcal{D} \succ \mathcal{D}'$, if $\mathcal{D} \succeq \mathcal{D}'$ and not $\mathcal{D}' \succeq \mathcal{D}$. The notation is extended to failure detector classes, and $\mathcal{A} \succeq \mathcal{B}$ means "$\forall(\mathcal{D}, \mathcal{D}') \in \mathcal{A} \times \mathcal{B}, \mathcal{D} \succeq \mathcal{D}'$" when $\mathcal{A}$ and $\mathcal{B}$ are classes.

In this paper, the reduction algorithms have variables $output_p$ that verify the following property, where $output_p(t)$ is the value of $output_p$ at time t.

$$\forall F, \forall H \in \mathcal{D}(F), ((p,t) \mapsto output_p(t)) \in \mathcal{D}'(F)$$

3 Previous works

Chandra and Toueg defined a few properties that failure detectors $\mathcal{D}$ may satisfy.

3.1 [CT96]'s classes

Strong completeness
Eventually every process that crashes is permanently suspected by *every* correct process.

$$\forall F, \forall H \in \mathcal{D}(F), \exists t \in \mathbb{N}, \forall p \in crashed(F), \forall q \in correct(F), \forall t' \geq t, p \in H(q,t')$$

Weak completeness
Eventually every process that crashes is permanently suspected by *some* correct process.

$$\forall F, \forall H \in \mathcal{D}(F), \exists t \in \mathbb{N}, \forall p \in crashed(F), \exists q \in correct(F), \forall t' \geq t, p \in H(q,t')$$

Strong accuracy
No process is suspected before it crashes.

$$\forall F, \forall H \in \mathcal{D}(F), \forall t \in \mathbb{N}, \forall p, q \in correct(F,t), p \notin H(q,t)$$

Weak accuracy
Some correct process is never suspected.

$$\forall F, \forall H \in \mathcal{D}(F), \exists p \in \mathit{correct}(F), \forall t \in \mathbb{N}, \forall q \in \mathit{correct}(F,t), p \notin H(q,t)$$

Eventual strong accuracy

$$\forall F, \forall H \in \mathcal{D}(F), \exists b \in \mathbb{N}, \forall t \geq b, \forall p, q \in \mathit{correct}(F,t), p \notin H(q,t)$$

Eventual weak accuracy

$$\forall F, \forall H \in \mathcal{D}(F), \exists b \in \mathbb{N}, \exists p \in \mathit{correct}(F), \forall t \geq b, \forall q \in \mathit{correct}(F,t), p \notin H(q,t)$$

The combinations of these properties lead to the definition of eight classes of failure detectors, as shown in Table 1. These classes are a reference in the domain.

Completeness	**Accuracy**			
	Strong	Weak	Eventual Strong	Eventual Weak
Strong	$\mathcal{P}$	$\mathcal{S}$	$\diamond\mathcal{P}$	$\diamond\mathcal{S}$
Weak	$\mathcal{Q}$	$\mathcal{W}$	$\diamond\mathcal{Q}$	$\diamond\mathcal{W}$

Table 1: Eight classes of failure detectors

We have $\mathcal{Q} \cong \mathcal{P}$, $\mathcal{W} \cong \mathcal{S}$, $\diamond\mathcal{Q} \cong \diamond\mathcal{P}$, and $\diamond\mathcal{W} \cong \diamond\mathcal{S}$, as shown in [CT96]. Thus, there is only four classes to study. The order between these classes is shown on Figure 1. $\mathcal{S}$ and $\diamond\mathcal{P}$ are incomparable.

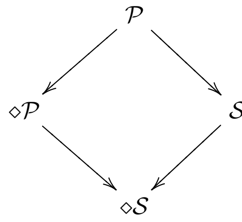

Figure 1: The four remaining classes, ordered

3.2 Consensus

Consensus
The Consensus is defined by the four following properties.

Termination	Every correct process eventually decides some value.
Uniform integrity	Every process decides at most once.
Agreement	No two correct processes decides differently.
Uniform validity	If a process decides v, then v was proposed by some process.

Theorem 3.1 (FLP) *Without failure detector, the Consensus is unsolvable in environments where a single crash failure is possible.*

Knowing this first result, Chandra and Toueg defined the model in which we are now working. By the way, they proved many results about Consensus.

Theorem 3.2 ([CT96]) *The Consensus problem is solvable using any $\mathcal{D} \in \mathcal{S}$.*

Theorem 3.3 ([CT96]) *The Consensus problem is solvable using any $\mathcal{D} \in \diamond\mathcal{S}$, if $2f < n$ (i.e. in $\mathcal{E}_{\lceil \frac{n}{2} \rceil}$).*

Theorem 3.4 ([CT96]) *Consensus cannot be solved using $\diamond\mathcal{P}$ when $2f \geq n$.*

In [CHT96], a new class is defined. It allows the authors to locate more precisely the weakest failure detector for the Consensus, especially in $\mathcal{E}_{\lceil \frac{n}{2} \rceil}$.

The failure detectors class Ω

Ω is the class of detectors $\mathcal{D}$ that verify the two following properties.

$\forall F, \forall H \in \mathcal{D}(F), H(p,t) \in \Pi$ [2]

$\forall F, \forall H \in \mathcal{D}(F),$

$\exists t \in \mathbb{N}, \exists q \in correct(F), \forall p \in correct(F), \forall t' > t, H(p,t) = q$

Theorem 3.5 ([CHT96]) *If $\mathcal{D}$ solves the Consensus, then $\mathcal{D} \succeq \Omega$.*

Lemma 3.6 ([CHT96]) $\Omega \succeq \diamond\mathcal{S}$.

Theorem 3.7 ([CHT96]) $\Omega =_{\mathcal{E}_{\lceil \frac{n}{2} \rceil}} \diamond\mathcal{S}$ *is the weakest class for solving the Consensus in $\mathcal{E}_{\lceil \frac{n}{2} \rceil}$.*

4 The Leader Election

4.1 The beginning of our study

Here is the Leader Election, as defined by Cho and Park.

Leader Election

Safety	All processes connected to the system never disagree on a *leader* when the nodes are in a state of normal operation.
Liveness	All processes should eventually progress to be in a state in which all processes connected to the system agree on the *only one* leader

Theorem 4.1 ([CP02]) *Election cannot be solved using $\diamond\mathcal{P}$ or $\mathcal{S}$.*

Theorem 4.2 ([CP02]) *$\mathcal{P}$ is sufficient to solve Election, using the algorithm on Figure 2.*

Figure 2: Leader election using $\mathcal{P}$

1. Each process has a unique ID number that is known by all processes a *priori*.
2. The leader is initially the process with the lowest ID number.
3. If a process detects a failure, it broadcasts this information to all other processes. Upon receiving such a message, the receiver detects the failure.
4. When a process detects the failure of all processes with lower ID numbers, then that process becomes the leader.

Theorem 4.3 ([CP02]) Among the eight classes*, a weakest failure detector to solve Election is the Perfect Failure Detector.*

4.2 The Leader Election

Before anything else, we had to design a formal definition of the Leader Election.

<u>**Leader Election**</u>
An algorithm solves the Leader Election if it satisfies the following properties. Each process has a variable $leader_p$, which value belongs to $\{\perp\} \cup \Pi$. We denote by $leader_p(t)$ the value of $leader_p$ at time t.

Safety $\forall t \in \mathbb{N}, \forall p, q \in correct(F, t),$
$\text{leader}_p(t) = \perp \vee \text{leader}_q(t) = \perp \vee \text{leader}_p(t) = \text{leader}_q(t)$

Liveness $(\exists t \in \mathbb{N}, p \in correct(F, t), leader_p(t) \notin correct(F, t)) \Rightarrow$
$(\exists t' > t, \exists q \in correct(F, t'), \forall p \in correct(F, t'), leader_p(t') = q)$

4.3 The Boolean Leader Election

There is some other way to define our problem.

<u>**Boolean Leader Election**</u>
An algorithm solves the Leader Election if it satisfies the following properties. Each process has a variable $leader_p$, which value belongs to $\{true, false\}$. We denote by $leader_p(t)$ the value of $leader_p$ at time t.

Safety $\forall t \in \mathbb{N}, \forall p, q \in correct(F, t),$
$\neg(leader_p(t) \wedge leader_q(t))$

Liveness $(\exists t \in \mathbb{N}, \nexists p \in correct(F, t), leader_p(t)) \Rightarrow$
$(\exists t' > t, \exists q \in correct(F, t'), leader_q(t'))$

$\mathcal{L}_\mathcal{B}$ is solvable with $\mathcal{P}$, with the algorithm on Figure 2. This is not true for $\mathcal{L}$, because one cannot know who is the smallest alive process except if it is the

smallest process which is not suspected by its own failure detector module. A reason is that the smallest process which is not suspected by a process p can be crashed, since completeness properties are only eventual properties.

Meanwhile, we'll show that these two definitions are not very different. It is clear that the Boolean Leader Election is weakest than the other, probably strictly[3]. But most of our results are true for both of the definitions.

4.4 What does correct mean ?

There is a difference between being correct at time t and being correct. In problems definitions, properties are often given for correct processes only. Does it mean that processes that will crash can do what they want , even a "long time" before the failure ? Fortunately not. The determinism makes the algorithm satisfy the properties on processes that are not yet crashed, until they are distinguishable from the (for ever) correct ones.

4.5 The class $\mathcal{L}$

First, we define the weakest class that solves the Leader Election. Obviously, this class exists for every problem. Using it is useful in the proofs. For the Consensus and the Mutex problems, the weakest classes in $\mathcal{E}_{\lceil \frac{n}{2} \rceil}$ have been found, and were defined very independently from the problem, what was very elegant. Here, the definition comes directly from the problem, but fortunately, we could use it simply. [4]

One can talk about problems instead of detectors class, and *vice versa*. For example, there is no possible misunderstood when we say that a class reduces to a problem (that means that the class reduces to the problem's weakest detector class) or that a process reduces to a class (every detector in the class solves the problem).

<u>**The failure detector class $\mathcal{L}$**</u>
$\forall \mathcal{D} \in \mathcal{L}, \forall F \in \mathcal{P}(\Pi)^{\mathbb{N}}, \forall H \in \mathcal{D}(F),$

1 $$\forall t \in \mathbb{N}, \forall p \in \mathit{correct}(F,t), H(p,t) \in \{\bot\} \cup \Pi$$

2 $$\forall t \in \mathbb{N}, \forall p,q \in \mathit{correct}(F,t), H(p,t) = \bot \vee H(q,t) = \bot \vee H(p,t) = H(q,t)$$

3 $$(\exists t \in \mathbb{N}, p \in \mathit{correct}(F,t), H(p,t) \notin \mathit{correct}(F,t)) \Rightarrow$$
$$(\exists t' > t, \exists q \in \mathit{correct}(F,t'), \forall p \in \mathit{correct}(F,t'), H(p,t') = q)$$

Theorem 4.4 *$\mathcal{L}$ is the weakest class of failure detector for solving the Leader Election. $\mathcal{L}_0$, the weakest failure detector in $\mathcal{L}$ [5] , is the weakest failure detector*

[3] For instance, $\mathcal{M} \succeq \mathcal{L}_{\mathcal{B}}$ is obvious, but I don't believe in $\mathcal{M} \succeq \mathcal{L}$.

[4] This is not the case with Mutex or unsynchronized leader election (c.f. Section 5.1), which are "more interactive" problems than the simple leader election.

[5] $\mathcal{L}_0(F) = \cup_{\mathcal{D} \in \mathcal{L}} \mathcal{D}(F)$

for solving the Leader Election. Formally, we have :

$$\mathcal{D} \text{ solves the leader election} \Leftrightarrow \mathcal{D} \succeq \mathcal{L}$$

Proof

- $\Leftarrow$ It is easy to see that $\mathcal{L}$ solves the Leader Election.
- $\Rightarrow$ Let $\mathcal{D}$ be a failure detector that is sufficient for the Leader Election. Using $\mathcal{D}$ and the algorithm A that solves the Leader Election with $\mathcal{D}$, we can build a detector $\mathcal{D}' \in \mathcal{L}$, using the algorithm on Figure 3.

□

Figure 3: $T_{\mathcal{D}\to\mathcal{D}'}$ where $\mathcal{D}' \in \mathcal{L}$

```
output_p ←⊥
cobegin
  || leader_election()                         // using D
  || do forever
         output_p ← leader_p
     done
coend
```

The theorem allows us to denote by $\mathcal{L}$ either the class or the corresponding problem. We can do the same with $\mathcal{L}_{\mathcal{B}}$, the Boolean Leader Election.

4.6 Results

4.6.1 How to solve the Leader Election ?

Theorem 4.5 *The algorithm on Figure 4 solves $\mathcal{L}$ using $\mathcal{P}$.*

Proof

No correct process blocks [6]

The Consensus never blocks, so the only blocking operation is wait. Proof by induction on the wait steps p1 and p2. Every correct process obviously reaches the first wait step. If every correct process reaches the nth wait step, then for all p, for all q, q is either correct — and the unblocking message from q is eventually delivered by p — or not — and it is eventually suspected, by the *strong completeness* of $\mathcal{P}$. Thus p passes the wait step, and thus reaches the $(n+1)^{th}$ step.

[6] The bold lines the important steps of the proofs, announcing what will be proved in the next step.

Figure 4: $T_{\mathcal{P}\rightarrow\mathcal{L}}$

```
do forever
  p1  leader_p ←⊥
  p1  v_p ← p
  p1  Consensus()
  w1  send choice_done to all
  w1  for each q ∈ Π, wait for (choice_done from q or suspects(q))
  p2  leader_p ← decided_p
  w2  send round_done to all
  w2  for each q ∈ Π, wait for (round_done from q or suspects(q))
  p3  wait for suspects(leader_p)
done
```

No two correct process are on different sides of a same wait step at the same time

Obvious from the algorithm. The correct process p passes the wait if and only if the correct process q has entered the step, by the *strong accuracy* of $\mathcal{P}$.

Safety is satisfied

Consider *ab absurdo* the first time where safety is broken. We have $leader_p = l \in \Pi$ and $leader_q = l' \in \Pi$, with $l \neq l'$, and q in step p2. Thus p is in the corresponding w1 or w2, thus $l = \perp$. Contradiction.

Liveness is also satisfied

There is infinitely many instants t where the first correct process reaches step w2, while all of the other are in step p2.

Bonus : stability

The optional line p3 gives stability [7] to the implemented leader election.

□

Theorem 4.6 *$\mathcal{S}$ and $\diamond\mathcal{P}$ are not sufficient to solve these leader elections, in $\mathcal{E}_{n-1}$.*

Proof

The leader crashes

Consider a run where $leader_p = p_1$ at time t_1 for every process p. At time $t_1 + 1$, p_1 crashes. All other processes are correct. Thus there exists t_2 such that $leader_p = p_2$ for every $p \neq p_1$ at time t_2. For this run, the failure detector history can be the perfect description of the failure pattern, it satisfies $\diamond\mathcal{P}$ and $\mathcal{S}$ properties.

The leader doesn't crash

[7] The definition of stability comes in section 4.8.

Now, let's build another run. No process crashes. It begins as the first : same communication delays, same steps until t_1. The history is the same until t_2 — $\diamond\mathcal{P}$ and $\mathcal{S}$ allow it. Between time $t_1 + 1$ and t_2, no step is given to p_1. Thus, $leader_{p_1} = p_1$ for all $t \in [0; t_2]$. And at time t_2, $leader_p = p_2$ for every $p \neq p_1$, since the algorithm is determinist. It breaks the safety.
□

The proof is the one from [CP02], expanded and explained. It took a very long time for me to accept it, as I did not understood fully the asynchronicity of the model : I had not noticed that a process could be stopped during an arbitrary delay, but had only in mind the consequences of the asynchronicity on the communications.

4.6.2 What does the Leader Election provides ?

The next result is new and remarkable : nothing among the problems and classes previously studied reduces to our Leader Election. This problem belongs to an other dimension . . .

Theorem 4.7 *$\mathcal{L} \succeq \diamond\mathcal{S}$ and $\mathcal{L}_{\mathcal{B}} \succeq \diamond\mathcal{S}$ are absurd.*

Proof

Ab absurdo. We assume the existence of an algorithm $T_{\mathcal{L}_0 \to \mathcal{D}}$ where $\mathcal{D} \in \diamond\mathcal{S}$. We'll suppose that $\Pi = \{p_0, p_1\}$, and each process is correct, in every run of the transformation. The proof can be immediately extended for any Π, if we assume that $|\Pi| - 2$ processes crashes at time 0, but it is more clear for $|\Pi| = 2$.

About suspicions

In the following we'll always use the same trick. Suppose that p_1 is the leader at time t. If communications are delayed for a sufficiently long time, then p_1 will be lead to suspect p_0. That is because the run were p_0 crashes at time t and p_1 remains to be the leader forever — it is valid : $H \in \mathcal{L}_0(F)$ — must emulate an eventually strongly complete detector.

We now build, by induction, a sequence of runs R_n such that H'_n, the emulated history, doesn't satisfy weak accuracy until n : $p_{n \bmod 2}$ is suspected by the other process at time $t_n > n$.

Run R_0

p_1 is the leader forever, and suspects p_0 at time $t_0 > 0$ in H'_0. The communications are delayed until t_0. All messages sent are delivered at $t_0 + 1$.

Run R_{k+1}

Suppose that $p_{k \bmod 2}$ is suspected at $t_k > k$ in H'_k.

We modify of the run, after t_k : $p_{k \bmod 2}$ is the leader forever, the communications are delayed until $t_{k+1} > t_k + 1$, great enough to make $p_{k \bmod 2}$ suspects $p_{1+k \bmod 2}$.

We have $t_{k+1} > t_k$, thus $t_{k+1} > k + 1$, and $p_{1+k \bmod 2}$ is suspected at t_{k+1} in H'_{k+1}. The condition $t_{k+1} > t_k + 1$ is here to guarantee that a message is never delayed forever.

Run R

The limit run R — which is heavy to define formally, but do exists — emulates an history which doesn't satisfy the eventual weak accuracy. This is absurd.
□

This proof is very efficient. It also shows that $\mathcal{L} \succeq \mathcal{S}$, $\mathcal{L} \succeq \Omega$, $\mathcal{L} \succeq \mathcal{P}$, $\mathcal{L} \succeq \mathcal{T}$ and $\mathcal{L} \succeq \diamond\mathcal{P}$ are absurd.

4.7 Classification

The following diagrams are a good mean to visualize all the results about Leader Election, and other problems. The symbol $\rightarrow$ represents $\succ$. The dotted version of the symbol represents an open question. It would have been too heavy to draw every question that remains, we've only put down the more important ones. In the general case, $\mathcal{E}_n$, the order is represented on Figure 5. $\mathcal{T} \not\succeq \mathcal{S}$ is shown in the Appendix.

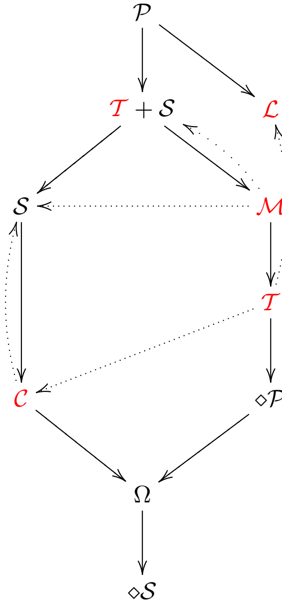

Figure 5: The classification of the detector classes, in $\mathcal{E}_n$

In $\mathcal{E}_{\lceil \frac{n}{2} \rceil}$, the diagram (Figure 6) is simpler. Some questions among those that we have previously drawn are solved immediately. We haven't put down the remaining ones.

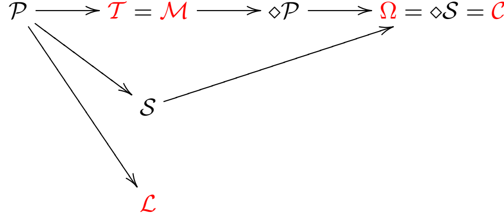

Figure 6: The classification of the detector classes, in $\mathcal{E}_{\lceil \frac{n}{2} \rceil}$

4.8 About stability

Stability
The leader election is stable if a leader remains to be the leader until it crashes.

$$\forall F \in \mathcal{P}(\Pi)^{\mathbb{N}}, \forall (t,d) \in \mathbb{N}^2, \forall p \in correct(F,t),$$

$$(leader_p(t) \neq leader_p(t+d) \Rightarrow leader_p(t) \in F(t+d))$$

The stability is an interesting property that the leader election may satisfy. Actually, it was part of the Leader Election at the beginning of our work. But we thought that it was not necessary and removed it. Later, I found that the stability property would change many things in our results.

We will denote by $\mathcal{L}_{\mathcal{S}}$ and $\mathcal{L}_{\mathcal{SB}}$ the two problems made by the addition of the stability to $\mathcal{L}$ and $\mathcal{L}_{\mathcal{B}}$.

Proposition 4.8 *Stability doesn't come for free in $\mathcal{L}$ and $\mathcal{L}_{\mathcal{B}}$.*

Proof
Obviously, $\mathcal{L}_{\mathcal{S}} \succeq \Omega$: at any time, each process trusts its leader, or itself if it has no leader. And we have shown in Theorem 4.7 that $\mathcal{L} \succeq \Omega$ and $\mathcal{L}_{\mathcal{B}} \succeq \Omega$ were absurd.
□

First, we note that Theorem 4.5 and Theorem 4.6 stand for $\mathcal{L}_{\mathcal{S}}$ and $\mathcal{L}_{\mathcal{SB}}$. But, obviously, the proof of Theorem 4.7 is no more correct when we add stability — meanwhile, part of the result stands (Lemma 4.9). The Stable Leader Election may be a more interesting problem, in the sense that it can be more linked with the other fundamental problems. For example, in $\mathcal{E}_{\lceil \frac{n}{2} \rceil}$, the Stable Leader Election solves the Consensus.

Lemma 4.9 *$\mathcal{L} \succeq \mathcal{P}$ is absurd, for the four leader elections.*

Proof

Ab absurdo. Let $\mathcal{D}$ be the weakest detector in $\mathcal{L}$, suppose that there exists $T_{\mathcal{D}\to\mathcal{D}'}$ for some $\mathcal{D}' \in \mathcal{P}$: for any failure pattern F, and $H \in \mathcal{D}(F)$, the algorithm build some $H' \in \mathcal{D}'(F)$.

A crash

We choose F to be the failure pattern where a single process p_2 crashes, at time 0. We choose H in $\mathcal{D}(F)$:

$$\forall t \in \mathbb{N}, \forall p \in correct(F), H(p,t) = p_1$$

The emulated history H' satisfies strong completeness and strong accuracy.

$$\exists t_1 \in \mathbb{N}, \forall p \in correct(F), \forall t \geq t_1, H'(p,t) = \{p_2\}$$
$$\forall p \in correct(F), \forall t < t_1, H'(p,t) = \emptyset$$

A delay

Now, p_2 is correct, but its communications are delayed until time $t_1 + 1$. We choose H, defined in the previous case. That is possible since H belongs to $\mathcal{D}(F' = t \mapsto \emptyset)$ too. Thus, the behavior of every computer but p_2 is the same as in the first run, until $t_1 + 1$. H', the emulated "perfect" history is the same for the two cases until $t_1 + 1$. It breaks the strong accuracy at time t_1.

□

5 Conclusion

We have studied a few leader elections, and proved again that among the eight classes defined by Chandra and Toueg, the perfect failure detector is needed to solve these problems. The leader elections are hard to solve, but we proved that they can not solve anything known. Fortunately, we found that the stability property makes the leader elections solve Ω, and thus solve the Consensus in $\mathcal{E}_{\lceil \frac{n}{2} \rceil}$.

5.1 Towards a new definition ?

We designed a third definition, original and interesting, but unfortunately unachieved. It illustrates well a big that I have learned : "formalizing the problem is already a big problem".

Keep it ?

In the previous definitions, the Safety condition requires a kind of synchronization. This should be avoided while working in an asynchronous system. We wrote a new definition from this idea : *the system doesn't need to be safe when it is unused.*

Leader Election

Each process has a variable $leader_p$, which value belongs to Π. We denote by $leader_p(t)$ the value of $leader_p$ at time t. Each process has an associated user that can run, at any time, a procedure $app_req(i)$ for some $i \in \mathbb{N}$, meaning that it wants that all the process run an application, which may require a leader.

We suppose that for all $i \in \mathbb{N}$ and $r \in \Pi$, r never runs twice $app_req(i)$. An algorithm solves the problem if it provides app_req and satisfies the following properties for every users and every run.

Safety If two correct processes p and q run $app_run(r, i)$, respectively at time t_p and t_q, then $leader_p(t_p) = leader_q(t_q)$.

Liveness If a correct process r executes $app_req(i)$ at time t, then for all correct process p there exists $t' > t$, such that p runs $app_run(r, i)$ at time t'.

First, we have to define another problem and an important result about it ([CHT96]).

Reliable Broadcast

Validity. If a correct process R-broadcasts a message m, then it eventually R-delivers m.

Agreement. If a correct process R-delivers a message m, then all correct processes eventually R-deliver m.

Uniform integrity. For any message m, every process R-delivers m at most once, and only if m was previously R-broadcast by $sender(m)$.

The Reliable Broadcast is solvable without any failure detector.

Atomic Broadcast

The Atomic Broadcast is the Reliable Broadcast with one more property to satisfy.

Total order. If two correct process p and q deliver two messages m and m', then p delivers m before m' if and only if q delivers m before m'.

Theorem 5.1 ([CT96]) *Consensus and Atomic Broadcast are equivalent in asynchronous systems.*

Theorem 5.2 *This Leader Election reduces to Atomic Broadcast.*

Corollary 5.3 *This Leader Election is solvable in $\mathcal{E}_n$ using $\mathcal{S}$.*

Proof

One may check easily that the algorithm on Figure 7 solves the Leader Election using Atomic Broadcast.

□

This definition looks more reasonable, and makes the problem easier to solve. That sounds good. But this leader election is in fact much more weaker than the others. Actually, we found that it can be solved without any failure detector, as shown by the algorithm on Figure 8. That looks unrealistic ! The error is that an application should not be launched by a single process, usually many processes require the start. It lead us to the Mutex problem : to start the

Figure 7: Leader Election using Atomic Broadcast

```
. r ← 0
. leader_p ←⊥
. Procedure app_req
  A-send(p, i)
. Procedure leader_election
  do forever
      wait  A-deliver(leader, q, i) when i = r + 1
      r ← i
      leader_p ← q
  done
. Procedure liveness
  do forever
      A-send(leader, p, r)
  done
. cobegin
    || leader_election()
    || liveness()
    || do forever
          wait  A-recv(q, i)
          app_run(q, i)
       done
  coend
```

Figure 8: A weak definition

```
  leader_p ← p
  procedure app_req(i)
      send (req, p, i) to all
|| do forever
      wait for receiving (run, q, i)
      leader_p ← q
      app_run(q, i)
  done
```

application — being its leader — a process should need to enter the CS of a mutex. Maybe an interesting and realistic definition of the leader election would make it reducible to the Mutual Exclusion ? That would contrast with the first definitions that we studied, in their stable form, which were more linked with the Consensus than with the Mutual Exclusion. We had not enough time to go further in that considerations. But we believe that a new definition, or at least the idea behind it, could be interesting.

6 Appendix

Lemma 6.1 *$\mathcal{T} \succeq \mathcal{S}$ is absurd.*

Proof

Ab absurdo. We suppose the existence of an algorithm $T_{\mathcal{T}_0 \rightarrow \mathcal{D}}$ for some $\mathcal{D} \in \mathcal{S}$. Let $\Pi = \{p_1, \ldots, p_n\}$.

The survivor

Consider a first run, where every process except p_1 crashes at time 0. H is defined as follows.

$$\forall t \in \mathbb{N}, H(p_1, t) = \{p_1\}$$

We check that H and F defined here verify : $H \in \mathcal{T}_0(F)$

$\mathcal{D}$ must satisfy the strong completeness : $\exists b_1 \in \mathbb{N}, output_{p_1}(t > b_1) = \Pi - \{p_1\}$

The survivor, again

We consider the same run, except that p_2 plays the role of p_1 and vice-versa.

$$\forall t \in \mathbb{N}, H(p_2, t) = \{p_2\}$$

$$\exists b_2 \in \mathbb{N}, output_{p_2}(t > b_2) = \Pi - \{p_2\}$$

The trap

We define a last run. We use F, the failure pattern where every process p_i where $i > 2$ crashes at time 0, and p_1 and p_2 are correct. We suppose that all communications are delayed until $max(b_1, b_2) + 1$, and define H :

$$\begin{aligned} \forall i \in \{1,2\}, \forall t \leq max(b_1, b_2) + 1, \quad H(p_i, t) = & \quad \{p_i\} \\ \forall i \in \{1,2\}, \forall t > max(b_1, b_2) + 1, \quad H(p_i, t) = & \quad \{p_1, p_2\} \end{aligned}$$

Firstly, we check that $H \in \mathcal{T}_0(F)$.

Thus, the emulated module running on process p_1 behaves exactly as in the first case until $max(b_1, b_2) + 1$. Same thing for p_2's module.

At time $max(b_1, b_2) + 1$ the two correct processes suspect each other. Weak accuracy is broken.

□

An alternate proof that makes use of [DFGK02] :

Proof

If $\mathcal{S}$ reduces to $\mathcal{T}$, $\mathcal{T} + \mathcal{S} \cong \mathcal{T}$, and $\mathcal{T}$ solves Mutex in $\mathcal{E}_n$. That is absurd.

□

It is more difficult to design the proof for $\neg(\mathcal{M} \succeq \mathcal{S})$, since there is no characterization of $\mathcal{M}$.

References

[FLP85] M.J. Fischer, N.A. Lynch, and M.S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(3):374-382, April 1985.

[CT96] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed Systems. *Journal of the ACM*, 43(2):225-267, March 1996.

[CHT96] Tushar Deepak Chandra, Vassos Hadzilacos and Sam Toueg. The weakest Failure Detector for Solving Consensus. *Journal of the ACM*, 43(4):685-722, March 1996.

[CP02] Mi-Hui Cho and Sung-Hoon Park. The Weakest Failure Detector for Solving Election Problems in Asynchronous Distributed Systems. *In Conference EURASIA-ICT*, 2002.

[ADFT01] Marcos K. Aguilera, Carole Delporte-Gallet, Hugues Fauconnier, and Sam Toueg. Stable Leader Election (Extended abstract). In *15th International Symposium on Distributed Computing*, Springer-Verlag, editor LNCS, Lisbonne, 2001.

[DFGK02] Carole Delporte-Gallet, Hugues Fauconnier, Rachid Guerraoui, Petr Kouznetstov. Mutual Exclusion in Asynchronous Systems with Failure Detectors. *EPFL*. IC Technical Report ID: 200227.

Contents

1 Introduction 1

2 The model 2
- 2.1 Asynchronous distributed systems 2
- 2.2 Crash failures 2
- 2.3 Unreliable failure detectors 3
- 2.4 Formal definitions 3
- 2.5 Reductions 4

3 Previous works 4
- 3.1 [CT96]'s classes 4
- 3.2 Consensus 5

4 The Leader Election 6
- 4.1 The beginning of our study 6
- 4.2 The Leader Election 7
- 4.3 The Boolean Leader Election 7
- 4.4 What does correct mean ? 8
- 4.5 The class $\mathcal{L}$ 8
- 4.6 Results 9
  - 4.6.1 How to solve the Leader Election ? 9
  - 4.6.2 What does the Leader Election provides ? 11
- 4.7 Classification 12
- 4.8 About stability 13

5 Conclusion 14
- 5.1 Towards a new definition ? 14

6 Appendix 18