Set Agreement and the Loneliness Failure Detector in Crash-Recovery Systems

Sergio Arévalo[1], Ernesto Jiménez[1], and Jian Tang[2]

[1] Universidad Politécnica de Madrid, 28031 Madrid, Spain
{sergio.arevalo,ernes}@eui.upm.es
[2] Distributed System Laboratory (LSD), Universidad Politécnica de Madrid, 28031 Madrid, Spain
tjapply@gmail.com

Abstract. The *set agreement* problem states that from n proposed values at most $n-1$ can be decided. Traditionally, this problem is solved using a failure detector in asynchronous systems where processes may crash but not recover, where processes have different identities, and where all processes initially know the membership. In this paper we study the set agreement problem and the weakest failure detector $\mathcal{L}$ used to solve it in asynchronous message passing systems where processes may crash and recover, with homonyms (i.e., processes may have equal identities) and without a complete initial knowledge of the membership.

1 Introduction

The *k-set agreement* problem [9] guarantees from n proposed values at most k can be decided. Two cases of this problem have received special attention: *consensus* (when $k = 1$), and *set agreement* (when $k = n-1$). The k-set agreement problem that is trivial to solve when the maximum number of processes that may crash (denoted by t) is lesser than k, or the maximum number of different proposed values (denoted by d) is equal or lesser than k (i.e., $t < k$ or $d \leq k$), becomes impossible to solve in an asynchronous system where processes may crash when $t \geq k$ and $d > k$ ([6], [15], [21]). To circumvent this impossibility result, many works can be found in the literature where the asynchronous system is augmented with a failure detector [20] to achieve k-set agreement. A failure detector [7] is a distributed tool that each process can invoke to obtain some information about process failures. There are many classes of failures detectors depending on the quality and type of the returned information ($\diamond\mathcal{P}$, Σ, $\mathcal{FS}^*$, ψ, ...).

A very important issue to solve k-set agreement is to identify the information needed about processes failures. We say that a failure detector class X is the weakest [7] to achieve k-set agreement if the information returned by any failure detector D of this class X is necessary and sufficient to solve k-set agreement.

This work has been partially funded by the Spanish Research Council (MICCIN) under project TIN2010-19077, by the Madrid Research Foundation (CAM) under project S2009/TIC-1692 (cofunded by ERDF & ESF).

In other words, with the failure information output by any failure detector D' of any class Y that solves k-set agreement, a failure detector $D \in X$ can be built on any asynchronous system augmented with a failure detector $D' \in Y$. We say that a class X is strictly weaker than Y (denoted by $X \prec Y$) if a failure detector $D \in X$ can be obtained from a system augmented with any failure detector $D \in Y$, and the opposite is not possible.

In message passing systems, Ω is the weakest failure detector to solve consensus (i.e., 1-set agreement) when a majority of processes do not crash [8], and $\mathcal{L}$ [13] is the weakest failure detector to solve set agreement (i.e., (n-1)-set agreement). For all $2 \leq k \leq n-2$, to find the weakest failure detector to achieve k-set agreement is an open question.

New assumptions have been studied trying to solve k-set agreement in a more realistic message passing systems. In [1] consensus and failures detectors are presented in an extension of the crash-stop model where processes can crash and recover (called crash-recovery model, and by extension, the systems with this failure model are denoted by crash-recovery systems). It is easy to see that these systems are generalizations of systems where processes fail by crashing-stop. A typical definition of a system [7] defines links between processes as reliable (i.e., each sent message is delivered to all alive processes without errors and only once). For the sake of extending traditional system assumptions, consensus and failure detectors are studied when *fair-lossy* links are used [1] (i.e., messages can be lost, but if a process sends permanently a message m to a same alive process, message m is also received permanently).

Sometimes the assumption of knowing the membership in advance is not possible when a run starts (e.g., in a p2p network where servers working as seeds are unknown a priori, and they are possibly different in each run, or even in the same run). This assumption is relevant because, for instance, even though Ω is implementable when the membership is unknown, none of the original eight classes of failures detectors proposed in [7] ($\mathcal{P}$, $\diamond\mathcal{P}$, $\mathcal{S}$, ...) are implementable if each process does not know initially the identity of all processes [17]. Note that any failure detector implementation for a system S with the assumption of unknown membership initially also works in any system S' with the same assumptions except that the membership is known (we say that S is a generalization of S').

Finally, homonymy is a novel assumption included in current systems where privacy is an important issue [12]. Homonymy allows to assign the same identity to more than one process (all processes with the same identity are homonymous). Note that a classical system of n processes with a different identity per process is a particular case of an homonymous system (there are n sets of homomymous processes of size 1). Similarly, anonymity [5] can be considered as a particular case of homonymy (there is a unique set of homomymous processes of size n, or, in other words, all processes are homonymous).

Related work As we said previously, new assumptions have been studied trying to solve k-set agreement in a more realistic way. Consensus and failure detectors were presented in asynchronous systems where processes may crash and recover [1]. Besides processes that in a run do not crash (*permanently-up*) and

processes that crash and stop forever (*permanently-down*), new classes of processes may appear in a run of a crash-recovery system: processes that crash and recover several times but after a time are always up (*eventually-up*), processes that crash and recover several times but after a time are always down (*eventually-down*), and processes that are permanently crashing and recovering (*unstable*). In these crash-recovery systems a process is said to be *correct* in a run if it is either permanently-up or eventually-up. On the other hand, an *incorrect* process in a run is either a permanently-down, eventually-down or unstable process. In [1] is proven that consensus with the failure detector $\diamond\mathcal{P}$ [7] is impossible to solve if the number of permanently-up processes in a run can be lesser or equal to the number of incorrect processes. There are in the literature several implementations of consensus and Ω for crash-recovery message passing systems ([1], [16], [18]).

Even though the initial knowledge of the membership is not always possible, different grades of knowledge are also possible. For example, Ω is implementable if each process initially only knows its own identity [17], or if each process also knows n (i.e., the number of processes of the system) [3].

In [2] new classes of failure detectors are introduced to work in homonymous systems. In that paper consensus is also implemented with the counterparts of the weakest failure detectors in classical message passing systems with unique processes' identities: Ω [8] when a majority of processes are correct (its counterpart is called $H\Omega$), and $\langle\Omega, \Sigma\rangle$ [11] when a majority of processes can crash (its counterpart is called $\langle H\Omega, H\Sigma\rangle$).

Regarding set agreement in message passing systems, in the literature we find only two works using the weakest failure detector $\mathcal{L}$ in crash-stop asynchronous systems ([13], [4]). In [13] a total order of process' identifiers and the initial knowledge of the membership is necessary. In [4] set agreement is implemented in systems where the knowledge of n is required.

The failure detector $\mathcal{L}$ is defined and implemented for crash-stop message passing systems in [4] and [19]. $\mathcal{L}$ is a failure detector defined for crash-stop systems in such a way that it always returns the boolean value *false* in some process p_i, and, if there is only one correct process p_j, eventually p_j returns *true* permanently. Nevertheless, in both implementations the algorithms always output *false* in all processes in runs where all processes are correct (i.e., in fail-free runs), which are most frequent in practice. This behaviour is relevant because the complexity of all algorithms that implement set agreement with $\mathcal{L}$ (our algorithm presented in this paper included) is improved if the number of processes that return *true* increases.

Our work Trying to generalize the results to the maximum number of systems as possible, this paper is devoted to study set-agreement in message passing systems with the weakest failure detector $\mathcal{L}$ in crash-recovery asynchronous systems with homonyms and without a complete initial knowledge of the membership. In our crash-recovery system model the maximum number of different processes that may crash and recover is so weak ($t = n$) that set-agreement can be solved but consensus can not [1]. An algorithm that implements set-agreement

for crash-recovery systems using $\mathcal{L}$ with homonyms and without initial knowledge of membership is presented in this paper.

We also show in this paper that it is not possible to implement $\mathcal{L}$ even in synchronous crash-recovery systems when $t = n$, or in partially-synchronous crash-recovery systems when $t = n - 1$. We introduce an algorithm that implements $\mathcal{L}$ in synchronous crash-recovery systems when $t = n - 1$. This algorithm works when a subset of processes' identities are known by all processes.

Note that, to our knowledge, both algorithms presented in this paper are the first that implement set agreement and $\mathcal{L}$ in crash-recovery systems. These are also the first algorithms that work with homonyms and without initial knowledge of membership in crash-stop systems.

This paper is organized as follows. The crash-recovery model is presented in Section 2. Definitions of set agreement and failure detector $\mathcal{L}$ are included in Section 3. In Section 4 we have an implementation of set agreement. The implementability of $\mathcal{L}$ is studied in Section 5. An implementation of $\mathcal{L}$ is presented in Section 6. We finish our paper with the conclusions in Section 7.

2 System Model

Processes The message passing system is formed by a set Π of processes, such that the size n of Π is greater than 1. We use $id(i)$ to denote the identity of the process $p_i \in \Pi$.

Homonymy There could be homonymous processes [2], that is, different processes can have the same identity. More formally, let ID be the set of different identities of all processes in Π. Then, $1 \leq |ID| \leq n$. So, in this system, $id(i)$ can be equal to $id(j)$ and p_i be different of p_j (we say in this cases that p_i and p_j are homonymous). Note that anonymous processes [5] are a particular case of homonymy where all processes have the same identity, that is, $id(i) = id(j)$, for all p_i and p_j of Π (i.e., $|ID| = 1$).

Unknown knowledge of membership Every process $p_i \in \Pi$ initially knows its own identity $id(i)$, but p_i does not know the identity of any subset of processes, or the size of any subset of Π, different of their trivial values. That is, process p_i only knows initially that $id(i) \in ID$ and $|\Pi| > 1$.

Time Processes are asynchronous, and, for analysis, let us consider that time advances at discrete steps. We assume a global clock whose values are the positive natural numbers, but processes cannot access it.

Failures Our system uses basically the failure model of crash-recovery proposed in [1]. In this model processes can fail by crashing (i.e., stop taking steps), but crashed processes may have a *recovery* if they restart their execution (i.e., they may recover). A process is *down* while it is crashed, otherwise it is *up*. Let us define a *run* as the sequence of steps taken by processes while they are up. So, in every run, each process $p_i \in \Pi$ belongs to one of these five classes:

- *Permanently-up*: Process p_i is always alive, i.e., p_i never crashes.

- *Eventually-up*: Process p_i crashes and recovers repeatedly a finite number of times (at least once), but eventually p_i, after a recovery, never crashes again, remaining alive forever.
- *Permanently-down*: Process p_i is alive until it crashes, and it never recovers again.
- *Eventually-down*: Process p_i crashes and recovers repeatedly a finite number of times (at least once), but eventually p_i, after a crash, never recovers again, remaining crashed forever.
- *Unstable*: Process p_i crashes and recovers repeatedly an infinite number of times.

In a run, a permanently-down, eventually-down or unstable process is said to be *incorrect*. On the other hand, a permanently-up or eventually-up process in a run is said to be *correct*. The set of incorrect processes in a run is denoted by $Incorrect \subseteq \Pi$. The set of correct processes in a run is denoted by $Correct \subseteq \Pi$. Hence, $Incorrect \cup Correct = \Pi$.

Unless otherwise is said, we will assume that there is no limitation in the number of correct (or incorrect) processes in each run, that is, $t = n$ (being t the maximum number of different processes that can crash and recover).

Features and use of the network The processes can invoke the primitive $broadcast(m)$ to send a message m to all processes of the system (except itself). This communication primitive is modeled in the following way. The network is assumed to have a directed link from process p_i to process p_j for each pair of processes $p_i, p_j \in \Pi$ $(i \neq j)$. Then, $broadcast(m)$ invoked at process p_i sends one copy of message m along the link from p_i to p_j, for each $p_{j \neq i} \in \Pi$. If a process crashes while broadcasting a message, the message is received by an arbitrary subset of processes.

Unless otherwise is said, links are asynchronous and fair-lossy [1]. A link is fair-lossy if it can lose messages, but if a process p_i sends a message m permanently (i.e., an infinite number of times) to a correct process p_j, process p_j receives m permanently (i.e., infinitely often). A fair-lossy link [1] does not duplicate or corrupt messages permanently, nor generates spurious messages.

Process status after recovery Following the same model of [1], when a process p_i recovers, it has lost all values stored in its variables previously to crash, and it has also lost all previous received messages. A special case are *stable storage variables*. All values stored in this type of variables will remain available after a crash and recovery. Note that stable storage variables have their cost (in terms of operations latencies), and the algorithms have to reduce their use as far as possible.

Unless otherwise is stated, we consider, like in [1], that when a process p_i crashes executing an algorithm $\mathcal{A}$, if process p_i recovers, it knows this fact, that is, p_i starts executing from a established line of $\mathcal{A}$ different of line 1.

Nomenclature The asynchronous system with homonymy and with unknown membership previously defined in this section is notated by $HAS_f[\emptyset, \emptyset, n]$.

We denote by $HAS_f[X, Y, t]$ the system $HAS_f[\emptyset, \emptyset, n]$ augmented with the failure detector X ($\emptyset$ means no failure detector), and where all processes initially

know the identities of processes of Y ($\emptyset$ means unknown membership). The third parameter t indicates the maximum number of different processes that can crash and recover (n means that all processes can crash and recover). The sub-index f in the notation is used to denote that links are fair-lossy. For example, $HAS_f[\mathcal{L}, \Pi, n]$ denotes the asynchronous system with homonymous processes and fair-lossy links, enriched with the failure detector $\mathcal{L}$, where all processes initially know the identity of the members of Π, and where all processes can crash and recover. The classical definition of asynchronous systems found in the literature could be denoted by $AS_r[\emptyset, \Pi, t]$. That is, an asynchronous system without homonymy, with reliable links (i.e., where each sent message is delivered to all alive processes without errors and only once), where at most t processes can crash, and where all processes initially know the identity of the members of Π.

We will use HAS to denote a homonymous asynchronous system where the parameters are not relevant. Similarly, we use AS instead of HAS to indicate that it is a classical system where each process has a different identity.

3 Definitions

First, we will formalize here the set agreement problem [9].

Definition 1. *(Set agreement). In each run, every process of the system proposes a value, and has to decide a value satisfying the following three properties:*
1. Validity: Every decided value has to be proposed by some process of the system.
2. Termination: Every correct process of the system eventually has to decide some value.
3. Agreement: The number of different decided values can be at most $n - 1$.

It is easy to see that if $t = n$ and there is not any stable storage variable, if all processes crash jointly previously to decide, and after that they recover, all proposed values will be lost forever. Then, the Validity Property can not be preserved, and, hence, set agreement can not be solved. Thus, any algorithm that implements set agreement needs to use stable storage variables.

Like in [1], we consider that a process p_i proposes a value v when process p_i writes v into a predetermined stable storage variable. Similarly, a process p_i decides a value v when process p_i writes v into another predetermined stable storage variable. Hence, after a recovery, a process p_i, reading these variables, can know easily if a value has already been proposed and/or decided.

The set agreement problem can not be solved in asynchronous systems where any number of processes can crash and not recover ([6], [15], [21]). To circumvent this impossibility result, we use a failure detector [7].

The failure detector $\mathcal{L}$ [13] was defined for asynchronous systems with the crash-stop failure model. We adapt here this definition of $\mathcal{L}$ to asynchronous systems where processes can crash and recover. Let us consider that each process

p_i has a local boolean variable $output_i$. We denote by $output_i^\tau$ this variable at time τ. Let us assume that the value in $output_i$ is *false* while process p_i is crashed (i.e, $output_i^\tau = false$, for all time τ while p_i is down). In each run, a failure detector of class $\mathcal{L}$ satisfies the following two properties:
1. Some process p_i always returns in its variable $output_i$ the value *false*, and
2. If p_i is the unique correct process, then there is a time after which p_i always returns in its variable $output_i$ the value *true*.

More formally, the definition of $\mathcal{L}$ for crash-recovery systems is the following.

Definition 2. *(Failure detector $\mathcal{L}$). For all process $p_i \in \Pi$ and run R, $output_i^\tau = false$ if process p_i is down at time τ in run R. Furthermore, the variable $output_i$ of every process $p_i \in \Pi$ must satisfy in each run R:*
1. $\exists p_i : \forall \tau$, $output_i^\tau = false$, *and*
2. $(Correct = \{p_i\}) \implies \exists \tau : \forall \tau' \geq \tau, output_i^{\tau'} = true$

To solve set agreement, we augment our asynchronous system $HAS_f[\emptyset, \emptyset, n]$ with the loneliness failure detector $\mathcal{L}$, which is the weakest failure detector to achieve set agreement in classical asynchronous message passing systems AS with the crash-stop failure model [13]. As we said previously, we denote this system enhanced with $\mathcal{L}$ as $HAS_f[\mathcal{L}, \emptyset, n]$.

4 Implementing Set Agreement in the Crash-Recovery Model

In this section we present the algorithm $\mathcal{A}_{set}$ (see Figure 1) that implements set agreement in homonymous asynchronous systems with unknown membership and with the failure detector $\mathcal{L}$, that is, in $HAS_f[\mathcal{L}, \emptyset, n]$.

Differently from $\mathcal{A}_{set}$, all algorithms presented in the literature to solve set agreement with $\mathcal{L}$ ([4] and [13]), besides working in crash-stop asynchronous systems, they need to know the system membership to work.

4.1 Explanation of $\mathcal{A}_{set}$

$\mathcal{A}_{set}$ is the algorithm of Figure 1 executed in $HAS_f[\mathcal{L}, \emptyset, n]$ to solve set agreement. Let $id(i)$ be the identifier of process p_i. Note that the values of these process identifiers could be whatever that imposes an order that allows to compare them. Also note that several identifiers can be the same (homonymous processes).

Like in [1], we consider that a process p_i proposes a value v (that is, $propose_i(v)$ is invoked) by writing v into a stable storage variable $PROP_i$. Similarly, a process p_i decides a value v (that is, $decide_i(v)$ is invoked) by writing v into another stable storage variable DEC_i. Let us suppose that both variables have the value $\perp$ previously to any invocation. If a process p_i recovers, it can see easily if it has already proposed or decided a value (that is, if $propose_i(v)$ or $decide_i(v)$ were invoked) reading these stable storage variables and checking if their values are different of $\perp$.

```
propose_i(v): % by writing v into PROP_i
 (1)  v_i ← v;
 (2)  start task 1

task 1:
 (3)  end_i ← false;
 (4)  repeat each η time
 (5)     % Phase 0
 (6)     broadcast (PH0, id(i), v_i);
 (7)     if (PH0, id(k), v_k) is received then
 (8)        if (⟨id(k), v_k⟩ ≤ ⟨id(i), v_i⟩) then
 (9)           v_i ← v_k;
 (10)          decide_i(v_k); % by writing v_k into DEC_i
 (11)          end_i ← true
 (12)       end if
 (13)    else
 (14)       % Phase 1
 (15)       if (PH1, v_k) is received then
 (16)          v_i ← v_k;
 (17)          decide_i(v_k); % by writing v_k into DEC_i
 (18)          end_i ← true
 (19)       else
 (20)          if (L.output_i=true) then % returned by L
 (21)             decide_i(v_i); % by writing v_i into DEC_i
 (22)             end_i ← true
 (23)          end if
 (24)       end if
 (25)    end if
 (26) until end_i;
 (27) start task 2

task 2:
 (28) repeat forever each η time
 (29)       broadcast (PH1, v_i);
 (30) end repeat

when process p_i recovers:
      % by checking PROP_i
 (31) if (propose_i() was invoked) then
           % by checking DEC_i
 (32)      if (decide_i() was invoked) then
 (33)           v_i ← DEC_i;
 (34)           start task 2
 (35)      else
 (36)           v_i ← PROP_i;
 (37)           start task 1
 (38)      end if
 (39) end if
```

Fig. 1. The algorithm $\mathcal{A}_{set}$ for set agreement in $HAS_f[\mathcal{L}, \emptyset, n]$.

The variable v_i is used by process p_i to keep the current estimate of its decision value (lines 9 and 16). This variable v_i contains initially the value v proposed by process p_i when it invokes $propose_i(v)$ (line 1). In order to remember, in case of recovering, the changes in v_i before crashing, a process p_i uses the stable storage variables $PROP_i$ and DEC_i (lines 33 and 36).

$propose_i(v)$ starts task 1. This task is a loop that executes lines 6-25 each η time until a decision is taken (and, hence, variable $end_i = true$).

Each process p_i in phase 0 broadcasts a $(PH0, id(i), v_i)$ message with a proposal v_i (initially v_i is p_i's proposal v, line 1) to the rest of processes of the system. After that, process p_i can decide a proposed value if a $(PH0, id(k), v_k)$ message is received. This value v_k is only decided if the condition $\langle id(k), v_k\rangle \leq \langle id(i), v_i\rangle$ happens. This condition is a shortcut for $(id(k) < id(i)) \vee [(id(k) = id(i)) \wedge (v_k \leq v_i)]$. That is, process p_i decides v_k if process p_k has a lesser identifier or, if they have the same identifier, v_k is lesser or equal than v_i. When a process decides, it moves to phase 1. If process p_i has not decided in phase 0, it can decide a value already decided by another process if a $(PH1, v_k)$ message is received. If after that phase 1 process p_i has not decided yet, it can decide its value v_i if the failure detector $\mathcal{L}$ returns *true* (i.e., $\mathcal{L}.output_i = true$). Note that at most $n-1$ processes can get *true* in this variable $output_i$ (from Condition 1 of Definition 2).

Finally, if process p_i decided in phase 0, phase 1, or locally because $\mathcal{L}.output_i = true$, the loop of lines 6-25 finishes, and task 2 starts. As links are not reliable

(but fair-lossy) and processes may crash and recover, with task 2 process p_i guarantees the propagation of its decided value v_i to the rest of processes. This value is broadcast in a $(PH1, v_i)$ message. The propagation is preserved repeating forever this broadcast invocation (lines 28-30).

If a process p_i crashes and recovers while running the algorithm, it always executes, after the recovery, lines 31-39. If process p_i proposed a value v but it crashed before writing any decision value in DEC_i, then p_i will get the proposed value from the stable storage variable $PROP_i$ (line 36). In other case, v_i will obtain its decided value from stable storage variable DEC_i (line 33). If it has already proposed and decided a value, process p_i starts task 2 to propagate this decided value (line 34). If process p_i has proposed a value but it has not decided yet, it starts task 1 to look for a value to decide (line 37).

4.2 Proofs of $\mathcal{A}_{set}$ in $HAS_f[\mathcal{L}, \emptyset, n]$

Lemma 1. *(Validity) For each run, if a process p_i of the system $HAS_f[\mathcal{L}, \emptyset, n]$ decides a value v', then v' has to be proposed by some process of the system $HAS_f[\mathcal{L}, \emptyset, n]$.*

Proof. The variable v_i has initially, when p_i starts for the first time, the value v proposed by process p_i when it invokes $propose_i(v)$ (line 1). Note that if process p_i recovers after proposing a value v but before writing any value in DEC_i, then $v_i = v$ (line 36). Thus, $v_i = v$ is broadcast in $(PH0, v_i)$ messages permanently (line 6 of p_i). So, this value $v_i = v$ only changes if:

Case 1: $(PH0, id(k), v')$ is received from some process p_k such that $\langle id(k), v' \rangle \leq \langle id(i), v \rangle$ (lines 7-12 of p_i). Then, $v_i = v'$ and $DEC_i = v'$, being v' the initial value proposed by process p_k.

Case 2: $(PH1, v')$ is received (lines 15-18 of p_i). We have three subcases:

Case 2.1: $(PH1, v')$ was broadcast by some process p_j after receiving $(PH0, id(k), v')$ of p_k ($p_k \neq p_j$) such that $\langle id(k), v' \rangle \leq \langle id(j), v \rangle$ (lines 7-12 and task 2 of p_j). Then, $v_i = v'$ and $DEC_i = v'$, being v' the initial value proposed by process p_k.

Case 2.2: $(PH1, v')$ was broadcast by some process p_j after receiving $(PH1, v')$ of other process p_x (lines 15-18 and task 2 of p_j). Note that this $(PH1, v')$ is broadcast, like in Case 2.1, when process p_x receives $(PH0, id(k), v')$ of some process p_k such that $\langle id(k), v' \rangle \leq \langle id(x), v \rangle$. Then, $v_i = v'$ and $DEC_i = v'$, being v' the initial value proposed by process p_k.

Case 2.3: $(PH1, v')$ was broadcast by process p_k when $output_k = true$ (lines 20-23 and task 2 of p_k). Then, $v_i = v'$ and $DEC_i = v'$, being v' the initial value proposed by process p_k.

Therefore, for each run, if a process p_i of the system decides a value v', then v' has to be proposed by some process of the system.

Lemma 2. *(Agreement) For each run, the number of different decided values in the system $HAS_f[\mathcal{L}, \emptyset, n]$ is at most $n - 1$.*

Proof. Let us suppose, by the way of contradiction, that there is a run R such that the number of different decided values is n. From Lemma 1, each decided value in R has to be one of the proposed values. Hence, if we find in this run R a proposed value which is not decided, we reach a contradiction.

Note that if in run R there are two processes p_i and p_j such that p_i proposes v_i, and p_i proposes v_j being $v_i = v_j$, then the statement of this lemma is trivial. So, we consider that $v_i \neq v_j$, for all p_i and p_j of the system.

Let us denote by G the set of processes that decide in this run R not executing lines 20-23. Note that $G \neq \emptyset$ from Condition 1 of Definition 2. Also note that this implies that every process $p_j \notin G$ decides its own proposed value.

Let us assume that $p_i \in G$ is the process with the greatest pair $\langle id(i), v\rangle$ among processes in G. Let us also assume that p_i proposes v_i. So, if contradiction holds, v_i has to be decided by p_i or by another different process p_j. We now analyze both cases and we will see that it is impossible that some process decides this value v_i in run R. Hence, we reach a contradiction.

Case 1: Process p_i decides v_i. As p_i, by definition, has the greatest pair $\langle id(i), v\rangle$ among processes in G, it did not receive any $(PH1, v_i)$ message from any process in G. Due to the fact that every process $p_j \notin G$ decides its own proposed value v_j (being $v_j \neq v_i$), process p_i did not receive any $(PH1, v_i)$ message from any process p_j. Then, it is impossible that process p_i decides its own proposed value v_i.

Case 2: Process p_j decides v_i, being $j \neq i$. As every process $p_k \notin G$ decides its own proposed value v_k (being $v_k \neq v_i$), then process $p_j \in G$. Hence, if process p_j decides v_i, which is a different value of its own proposed value v_j, it is because p_j receives a $(PH0, id(l), v_i)$ or $(PH1, v_i)$ message from some process $p_l \in G$. This is impossible because, by definition, p_i has the greatest pair $\langle id(i), v_i\rangle$ among processes in G, and $\langle id(i), v_i\rangle \leq \langle id(x), v_x\rangle$ is always false for all $p_x \in G$ (line 8).

Therefore, we reach a contradiction, and, for each run, the number of different decided values is at most $n - 1$.

Lemma 3. *(Termination) For each run, every process $p_i \in Correct$ of the system $HAS_f[\mathcal{L}, \emptyset, n]$ eventually decides some value.*

Proof. Let us suppose, by the way of contradiction, that there is a run R such that a correct process p_i never decides. Hence, if process $p_i \in Correct$ never decides in run R it is because lines 10, 17 and 21 are never executed.

Let us prove that this situation is impossible. If line 21 is never executed, then $\mathcal{L}.output_i = false$ permanently. If this is so, it is because there is at least another process p_k that is correct (from Condition 2 of Definition 2). Note that p_i, after its last recovery (if any), will be permanently broadcasting $(PH0, id(i), v_i)$ messages, being v_i the proposed value of p_i (line 6 of p_i). Hence, if process p_i never receives $(PH1, -)$ messages (lines 15-18 of p_i) it is because all processes p_l (included p_k) that receive the messages of p_i have a lesser pair $\langle id(l), v_l\rangle$ than $\langle id(i), v_i\rangle$ (line 8 of p_l). Nevertheless, process p_i will receive $(PH0, id(k), v_k)$ messages of p_k because links are fair-lossy, and correct process p_k also broadcasts $(PH0, id(k), v_k)$ messages permanently (line 6 of p_k). Then, p_i will execute line

10 because $\langle id(k), v_k \rangle < \langle id(i), v_i \rangle$. Hence, process p_i will decide v_k in run R. Therefore, we reach a contradiction, and, for each run, every process $p_i \in Correct$ eventually decides some value.

Theorem 1. *The algorithm of Figure 1 implements set agreement in the system* $HAS_f[\mathcal{L}, \emptyset, n]$.

Proof. From Lemma 1, Lemma 2 and Lemma 3, the validity, agreement and termination properties (respectively) are satisfied in every run. Hence, the algorithm of Figure 1 solves set agreement in the system $HAS_f[\mathcal{L}, \emptyset, n]$.

5 On the Implementability of $\mathcal{L}$ in the Crash-Recovery Model

In this section we prove that the failure detector $\mathcal{L}$ can not be implemented, even in a synchronous system where the membership is known, if up to n different processes can crash and recover, that is, $\mathcal{L}$ is not realistic [10]. We also prove in this section that the failure detector $\mathcal{L}$ can not be implemented in a partially synchronous system even if the membership is known and up to $n-1$ different processes can crash and recover.

Let $SS_r[\emptyset, \Pi, n]$ be a system like $AS_r[\emptyset, \Pi, n]$ but synchronous, that is, the maximum time to execute a step is bounded and known by every process, and the time to deliver a message is also known by all processes. Hence, $SS_r[\emptyset, \Pi, n]$ is a synchronous system where all processes have different identities, links are reliable, the membership is known, and the maximum number of processes that can crash and recover is $t = n$. Similarly, let $PSS_r[\emptyset, \Pi, n]$ be a system like $SS_r[\emptyset, \Pi, n]$ but partially synchronous [14], that is, the maximum time to execute a step by each process p_i is bounded, but unknown by every process different of p_i, and the time to deliver a message is bounded but unknown.

Lemma 4. *For every run, if in* $SS_r[\emptyset, \Pi, t]$ *or* $PSS_r[\emptyset, \Pi, t]$ *when* $t \geq n-1$ *a process* $p_i \in Correct$ *stops receiving messages from the rest of processes at some time* τ*, there is a time* $\tau' \geq \tau$ *where* $output_i^{\tau'} = true$.

Proof. Let us assume, by the way of contradiction, that there is a run R where some correct process p_i stops receiving messages from the rest of processes at some time τ, but for all time $\tau' \geq \tau$ it has $output_i^{\tau'} = false$.

Let us consider another run R' behaving exactly like R until time τ, and at this time τ all alive processes crash permanently except p_i. From Condition 2 of Definition 2 of $\mathcal{L}$, there is a time τ' where $output_i = true$. Note that each process only knows that in a run the rest of processes can crash, but it does not know a priori how many processes will crash or who they will be. Then, R and R' are indistinguishable until time τ' for p_i, and, hence, there is a time τ' where $output_i = true$ in R, which is a contradiction.

The following theorem shows that failure detector $\mathcal{L}$ can not be implemented in $SS_r[\emptyset, \Pi, n]$.

Theorem 2. *There is no algorithm $\mathcal{A}$ that implements the failure detector $\mathcal{L}$ in every run of a system $SS_r[\emptyset, \Pi, n]$, even if there is not any unstable process.*

Proof. Let us assume, by the way of contradiction, that there is an algorithm $\mathcal{A}$ that implements the failure detector $\mathcal{L}$ in every run of a system $SS_r[\emptyset, \Pi, n]$, even if there is not any unstable process.

For simplicity, let us consider that $\Pi = \{p_1, p_2, \ldots, p_n\}$, and that all these n processes of Π are eventually-up (hence, correct). Let us construct a valid run R of $\mathcal{A}$ as follows. For each process p_i, at time τ_i all processes crash except process p_i. From lemma 4, there is a time $\tau_i' \geq \tau_i$ where $output_i = true$. Now, all crashed processes recover at this time τ_i'. Let τ_1=0, and $\tau_i' < \tau_{i+1}$, $i = 1, \ldots, n$. Finally, after time τ_n' all processes keep alive in R (i.e., there is no unstable processes). Then, at time τ_n' all processes have had $output = true$ at some time, which violates Condition 1 of Definition 2. Hence, we reach a contradiction.

Therefore, there is no algorithm $\mathcal{A}$ that implements the failure detector $\mathcal{L}$ in every run of a system $SS_r[\emptyset, \Pi, n]$, even if there is not any unstable process.

The following theorem shows that failure detector $\mathcal{L}$ can not be implemented in $PSS_r[\emptyset, \Pi, n-1]$.

Theorem 3. *There is no algorithm $\mathcal{A}$ that implements the failure detector $\mathcal{L}$ in every run of a system $PSS_r[\emptyset, \Pi, n-1]$, even if there is not any unstable process.*

Proof. From Lemma 4, there is a time τ_i after which each process $p_i \in Correct$ sets $output_i = true$ if it stops receiving messages from the rest of processes. Let us consider that every process p_i in a run R is permanenly-up (hence, correct) and takes an step after a time τ which is greater than the maximum time τ_i, for all process $p_i \in \Pi$. Note that processes do not know a piori the time needed by other processes to take a step in run R, nor the number of other processes that are correct in R. Hence, there is a time $\tau' \geq \tau$ after which every process p_i has $output_i = true$, which violates the Condition 1 of Definition 2 of $\mathcal{L}$. Therefore, there is no algorithm $\mathcal{A}$ that implements the failure detector $\mathcal{L}$ in every run of a system $PSS_r[\emptyset, \Pi, n-1]$, even if there is not any unstable process.

6 Implementing $\mathcal{L}$ in the Crash-Recovery Model

From Section 5, we know that the failure detector $\mathcal{L}$ neither can be implemented in a synchronous system when until $t = n$ processes can crash and recover, that is, $\mathcal{L}$ is not realistic [10], nor in a partially synchronous system where $t = n - 1$. Now, we enrich here the system with a property such that we can circumvent this impossibility result. This property reduces to $t = n - 1$ the number of processes that can crash and recover in a synchronous system. Note that all algorithms found in the literature that implement the loneliness failure detector $\mathcal{L}$ ([4], [19]) work in systems where processes can crash but not recover and where up to $t = n - 1$ processes can crash and where the membership is totally known. Therefore, we present in this section an implementation of $\mathcal{L}$ (denote it by $\mathcal{A}_{\mathcal{L}}$)

for a synchronous system with homonymous processes, a partial knowledge of the membership, and where until $t = n - 1$ different processes can crash and recover.

6.1 Model

Let HSS be a system like HAS but synchronous. By synchronous we mean that processes start their execution at the same time, the time to execute a step is bounded and known by every process, the time to deliver a message sent through a link is at most Δ units of time, and this time is also known by all processes. For simplicity, we consider that the local execution time is negligible with respect to Δ (i.e., the execution time of a line of the algorithm is zero).

6.2 Algorithm $\mathcal{A}_{\mathcal{L}}$

We show in this section that the algorithm $\mathcal{A}_{\mathcal{L}}$ of Figure 2 implements the failure detector $\mathcal{L}$ in $HSS_r[\emptyset, Y, n-1]$ when $|Y| \geq 2$ and two processes of Y have different and known identities.

For each process p_i, $output_i$ is initially *false* (line 3). Process p_i uses the boolean value of the stable storage variable $restarted_i$ to communicate to the other processes if it has ever crashed (initially is *false*, line 1). If process p_i recovers, it will execute lines 19-20, and $restarted_i$ will be *true* (line 19). By definition of the system HSS used to execute $\mathcal{A}_{\mathcal{L}}$, process p_i knows at least two processes' identifiers with different values. These two known identifiers of Y with different value are $IDENT_1$ and $IDENT_2$ in Figure 2. Then, each process p_i whose identifier is neither $IDENT_1$ nor $IDENT_2$ changes $output_i$ to *true* (lines 4-6). Every η time, $\eta > \Delta$, each process p_i broadcasts heartbeats with $(alive, restarted_i)$ messages that arrive synchronously (at most Δ units of time later) to the rest of processes of the system (line 8). Note that we select a value η greater than Δ to allow that messages broadcast in line 8 arrive to processes on time in each iteration of line 9.

After Δ units of time, process p_i analyzes the messages received (rec_i) to see if it has to set $output_i$ to true (lines 11-17). Note that once $output_i = true$, process p_i never changes it to *false* again while it is running. Only if process p_i crashes and recovers, line 3 is executed again and $output_i$ is *false* again, but $restarted_i$ will be *true* in this case. The variable $count_i$ counts the number of heartbeats received by p_i from processes that are up, and that have never crashed (lines 12-14). If this number of messages is 0, then p_i sets $output_i = true$ (lines 15-17).

Note that in all algorithms in the literature that implement set agreement with $\mathcal{L}$ (our algorithm $\mathcal{A}_{set}$ included), the performance is improved if processes obtain *true* from $\mathcal{L}$ as soon as possible. This happens because a process of set agreement can decide locally (without waiting to receive any message) if *true* is returned by $\mathcal{L}$. For that reason, our algorithm $\mathcal{A}_{\mathcal{L}}$ with a partial knowledge of the membership immediately sets $output = true$ permanently in $n-2$ processes (lines 4-6 of Figure 2).

```
init:
(1)  restarted_i ← false; % stable storage variable
(2)  start task 1

task 1:
(3)  output_i ← false;
       % IDENT_1 and IDENT_2 are two
       % identifiers known by all processes
(4)  if ((id(i) ≠ IDENT_1) ∧ (id(i) ≠ IDENT_2)) then
(5)          output_i ← true
(6)  end if
(7)  repeat forever each η time
(8)       broadcast (alive, restarted_i);
(9)       wait Δ time;
(10)      let rec_i be the set of (alive, restarted) messages received;
(11)      count_i ← 0;
(12)      for_each ((alive, restarted) ∈ rec_i such that restarted = false) do
(13)           count_i ← count_i + 1
(14)      end for_each
(15)      if (count_i = 0) then
(16)           output_i ← true
(17)      end if
(18) end repeat

when process p_i recovers:
(19) restarted_i ← true; % stable storage variable
(20) start task 1
```

Fig. 2. Algorithm $\mathcal{A}_{\mathcal{L}}$ for process p_i to implement $\mathcal{L}$

Theorem 4. *The algorithm $\mathcal{A}_{\mathcal{L}}$ implements the failure detector $\mathcal{L}$ in a system $HSS_r[\emptyset, Y, n-1]$ when $|Y| \geq 2$ and two processes of Y have different identities.*

The proof of this theorem is omitted due to space limitations.

7 Conclusions

We study the *set agreement* problem in message passing systems with the weakest failure detectors $\mathcal{L}$ in crash-recovery asynchronous systems with homonymous processes and without a complete initial knowledge of the membership.

References

1. M. K. Aguilera, W. Chen and S. Toueg. Failure Detection and Consensus in the Crash-Recovery Model. Distributed Computing vol. 13(2), pp. 99–125, 2000.
2. S. Arévalo, A. Fernández Anta, D. Imbs, E. Jiménez and M. Raynal, Failure Detectors in Homonymous Distributed Systems (with an Application to Consensus). Proc. IEEE 32nd IEEE Int. Conf. on Distributed Computing Systems (ICDCS), pp. 275–284, 2012.
3. S. Arévalo, E. Jiménez, M. Larrea and L. Mengual. Communication-efficient and crash-quiescent Omega with unknown membership. Information Processing Letters 111 (4), pp. 194–199, 2011.

4. M. Biely, P. Robinson and U. Schmid U. Weak Synchrony Models and Failure Detectors for Message Passing (k-)Set Agreement. Proc. 11th International Conference on Principles of Distributed Systems (OPODIS'09), Springer Verlag LNCS 5923, pp. 285-299, 2009.
5. F. Bonnet and M. Raynal. Anonymous Asynchronous Systems: The Case of Failure Detectors. Distributed Computing, in press. DOI 10.1007/s00446-012-0169-5, 2013.
6. E. Borowsky and E. Gafni. Generalized FLP impossibility result for t-resilient asynchronous computations. STOC 1993: Proceedings of the twenty-fifth annual ACM symposium on Theory of computing, pp. 91–100. ACM, New York (1993)
7. T. Chandra and S. Toueg. Unreliable Failure Detectors for Reliable Distributed Systems. Journal of the ACM, 43(2), pp. 225–267, 1996.
8. Chandra T., Hadzilacos V. and Toueg S. The Weakest Failure Detector for Solving Consensus. Journal of the ACM, 43(4), pp. 685–722, 1996.
9. S. Chaudhuri. More Choices Allow More Faults: Set Consensus Problems in Totally Asynchronous Systems. Information and Computation, vol. 105, pp. 132–158, 1993.
10. C. Delporte-Gallet, H. Fauconnier and R. Guerraoui. A Realistic Look At Failure Detectors. Proc. 42th International IEEE Conference on Dependable Systems and Networks (DSN'02), pp. 345-353, 2002.
11. C. Delporte-Gallet, H. Fauconnier, R. Guerraoui, V. Hadzilacos, P. Kuznetsov and S. Toueg. The Weakest Failure Detectors to Solve Certain Fundamental Problems in Distributed Computing. Proceedings of 23th ACM Symp. on Principles of Distrib. Comp. (PODC), pp. 338–346, 2004.
12. C. Delporte-Gallet, H. Fauconnier, R. Guerraoui , A. .M. Kermarrec, E. Ruppert and H. Tran. The Byzantine agreement with homonymous. Proceedings of 30th ACM Symp. on Principles of Distrib. Comp. (PODC), pp. 21–30, 2011.
13. C. Delporte-Gallet, H. Fauconnier, R. Guerraoui and A. Tielmann. The Weakest Failure Detector for Message Passing Set-Agreement. Lecture Notes in Computer Science(LNCS), ISBN: 978-3-540-87778-3, vol. 5218, pp. 109–120, 2008.
14. D. Dolev, C. Dwork and L. Stockmeyer. On the minimal synchronism needed for distributed systems. Journal of the ACM 34(1), pp. 77–97, 1987.
15. M. Herlihy and N. Shavit. The topological structure of asynchronous computability. Journal of the ACM 46(6), pp. 858–923, 1999.
16. M. Hurfin, A. Mostefaoui, and M. Raynal. Consensus in asynchronous systems where processes can crash and recover. In Proceedings of the 17th IEEE Symposium on Reliable Distributed Systems (SRDS'98), pp. 280–286, 1998.
17. E. Jiménez, S. Arévalo, A. Fernández. Implementing unreliable failure detectors with unknown membership. Information Processing Letters 100 (2), pp. 60–63, 2006.
18. C. Martín, M. Larrea and E. Jiménez. Implementing the Omega Failure Detector in the Crash-recovery Failure Model. Journal of Computer and System Sciences, 75(3), pp. 178–189, 2009.
19. A. Mostefaoui, M. Raynal and J. Stainer. Relations Linking Failure Detectors Associated with k-Set Agreement in Message-Passing Systems. In Proceedings of the 13th International Symposium (SSS 2011), LNCS vol. 6976, pp. 341–355, 2011.
20. M. Raynal. Communication and Agreement Abstractions for Fault-Tolerant Asynchronous Distributed Systems. Morgan & Claypool Publishers, 250 pages, 2010.
21. M. Saks and F. Zaharoglou. Wait-free k-set agreement is impossible: The topology of public knowledge. SIAM Journal on Computing, 29(5), pp. 1449-1483, 2000.