Agreement in wider environments with weaker assumptions

Sergio ARÉVALO* Ernesto JIMÉNEZ* Jian TANG†

* EUI, Universidad Politécnica de Madrid, 28031 Madrid, Spain
† Distributed System Lab (LSD), Universidad Politécnica de Madrid, 28031 Madrid, Spain

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 do 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 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], [16], [22]). To circumvent this impossibility result, many works can be found in the literature where the asynchronous system is augmented with a failure detector [21] 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, 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. 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

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).

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] ($\mathcal{P}$, $\diamond\mathcal{P}$, $\mathcal{S}$, $\ldots$) are implementable if each process does not know initially the identity of all processes [18]. 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 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], [17], [19]).

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 [18], 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 presented 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 anonymous systems but the knowledge of n is required.

The failure detector $\mathcal{L}$ is defined and implemented for crash-stop message passing systems in [4] and [20]. $\mathcal{L}$ 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 a correct process p_j, eventually p_j returns *true* permanently. Nevertheless, in both implementations the algorithms always output *false* in all processes in runs which are most frequent in practice: where all processes are correct (i.e., in fail-free runs). 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 $t = n$ of different processes that may crash and recover is so weak 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. This algorithm works with different grades of initial knowledge of system membership (that is, either (a) each process only knows its identity, or (b) the number n of processes is known by every process, or (c) a subset of processes' identities are known by all processes). Due to the fact that, to our knowledge, there is no previous work in the literature that solves set-agreement with $\mathcal{L}$ in crash-recovery systems, we compare our results with previous papers that implement set-agreement and $\mathcal{L}$ in crash-stop systems.

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

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., $|ID| = 1$).
Knowledge of membership Every process $p_i \in \Pi$ knows its own identity $id(i)$, but p_i does not know the identity of any subset I of processes, or the size x of any subset of Π, different of their trivial values. That is, process p_i does not know $I \subseteq ID$ or $x \leq n$, except that $\{id(i)\} \subset I$ and $x > 1$.
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 four 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. Furthermore, p_i is alive or crashed an unbounded and unknown time. We distinguish in this class a special case of unstable process:
  - *Eventually-less-unstable*: Process p_i eventually, in each recovery, executes more lines of code until crashing again. More formally, there is a recovery r^j of p_i after which if p_i executed $l(r^j)$ lines of code before crashing, in each next recovery $r^{i+1} > r^i$, for all $i \geq j$, p_i will execute $l(r^{i+1}) > l(r^i)$ lines of code.

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, $Incorrect \cup Correct = \Pi$. Let us also denote by $Down \subseteq Incorrect$ the set of permanently-down and eventually-down processes in a run.

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. An 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 defined in this section is denoted by $HAS_f[\emptyset, \emptyset, c/r]$. We will use HAS when the parameters are not relevant. When it is needed, we use AAS instead of HAS to indicate that it is an anonymous system, that is, all processes have the same identity. Similarly, AS is an asynchronous system where each process has a different identity.

We denote by $HAS_f[X, Y, c/r]$ the system $HAS_f[\emptyset, \emptyset, c/r]$ 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). Note that the third parameter c/r means that processes can crash and recover (we will use c if processes can crash but do not recover), and the sub-index f that links are fair-lossy. For example, $HAS_f[\mathcal{L}, \Pi, c/r]$ denotes the asynchronous system with homonymous processes and fair-lossy links, enriched with the failure detector $\mathcal{L}$, and where all processes know the identity of the members of Π. The classical definition of asynchronous systems found in the literature could be denoted by $AS_r[\emptyset, \Pi, c]$. 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 processes can crash but not recover, and where all processes initially know the identity of the members of Π.

We extend the notation to $HAS^n_f[\emptyset, \emptyset, c/r]$ if it is a system like $HAS_f[\emptyset, \emptyset, c/r]$ but augmented with the knowledge of n. Then, every process p_i in this system knows the size n of Π (i.e., process p_i knows n despite it does not know the identity of the rest of processes of Π). Similarly, we extend $AAS^n_f[\emptyset, \emptyset, c/r]$ with respect to $AAS_f[\emptyset, \emptyset, c/r]$.

3 Definitions

We will formalize first 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 and 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 can know easily if a value has already been proposed and/or decided reading these variables.

The set agreement problem can not be solved in asynchronous systems $AS_r[\emptyset, \Pi, c]$ where any number of processes can crash ([6], [16], [22]). 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 and the rest of processes are (permanently or eventually) down, 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. $(Correct = \{p_i\} \wedge |Down| = n-1) \implies \exists \tau : \forall \tau' \geq \tau, output_i^{\tau'} = true$

To solve set agreement, we augment our asynchronous system $HAS_f[\emptyset, \emptyset, c/r]$ with the loneliness failure detector $\mathcal{L}$, which is the weakest failure detector to achieve set agreement in message passing systems $AS_r[\emptyset, \Pi, c]$ [13]. As we said previously, we denote this system enhanced with $\mathcal{L}$ as $HAS_f[\mathcal{L}, \emptyset, c/r]$.

In the following definition we say that an algorithm A implemented in a system S generalizes A' implemented in S', if A can be implemented in S' and the opposite is not possible.

Definition 3. *(Generalization). An algorithm A that solves the problem P in a system S is a generalization of the algorithm A' that solves the same problem P in S', denoted by $A' \subset A$, if A also solves P in S' and the opposite does not happen (i.e., A' does not solve P in S).*

4 Implementing Set Agreement in the Crash-Recovery Model

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

We can observe that in systems where processes can crash and recover, eventually-up and eventually-down processes are an extension of the permanently-up and permanently-down processes (respectively) of systems with the crash-stop model of failures. An special case is the existence of unstable processes. These processes are unpredictable, in the sense that they could be so fast on crashing and recovering that even though failure detectors consider them as permanently alive, they can not execute enough lines of code of the set agreement algorithm to communicate its state (sending messages) before crashing again. Then, the assumption of unstable processes in the system requires a property to guarantee that an unstable process collaborates to decide when there is a unique correct process.

Property 1. (Less-unstability) For each run, if there is some unstable process p_j and $Correct = \{p_i\}$, then some process p_k has to be eventually-less-unestable in the system $HAS_f[\mathcal{L}, \emptyset, c/r]$.

Note that this property does not enforce any condition in $HAS_f[\mathcal{L}, \emptyset, c/r]$ when there is no unstable processes, or when there is at least one unstable procress but there is not only one correct process.

The algorithms presented in [4] and [13] implement set agreement in $AAS_r^n[\mathcal{L}, \emptyset, c]$ and $AS_r[\mathcal{L}, \Pi, c]$, respectively. We show in this section that the algorithm of Figure 1 implements set agreement in $HAS_f[\mathcal{L}, \emptyset, c/r]$ enriched with Property 1. Hence, it is

easy to see that $\mathcal{A}_{set}$ implemented in $HAS_f[\mathcal{L}, \emptyset, c/r]$ with Property 1 is a generalization of the algorithms of [13] and [4]. More formally, let $\mathcal{A}'$ and $\mathcal{A}''$ be the set agreement algorithms of [13] and [4], respectively. From Definition 3, $\mathcal{A}' \subset \mathcal{A}_{set}$ and $\mathcal{A}'' \subset \mathcal{A}_{set}$.

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

$\mathcal{A}_{set}$ is the algorithm of Figure 1 executed in $HAS_f[\mathcal{L}, \emptyset, c/r]$ 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). To simplify the code of the algorithm, we consider that the execution of each concurrent task X of every process p_i is not perpetually postponed (i.e., starvation is not possible). Hence, in each run, process p_i eventually takes steps executing lines of task X if it is up, and the conditions that activate X are fulfilled.

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 $\perp$ 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 $\perp$.

The variable v_i is used by process p_i to keep the current estimate of its decision value (lines 10 and 17). This variable v_i contains initially the value v proposed by process p_i when it invokes $propose_i(v)$ (line 2). In order to remember, in case of recovering, the changes in v_i before crashing, a process p_i uses an stable storage variable $status_i$ (lines 9, 16 and 22).

$propose_i(v)$ starts four tasks. For simplicity, we consider that tasks 2, 3 and 4 are executed atomically. Task 1 is used in phase 0 ($PH0$) by each process p_i to broadcast $(PH0, id(i), v_i)$ messages permanently with a proposal v_i (initially v_i is p_i's proposal v, line 2) to the rest of processes of the system. Task 2 allows process p_i to decide a proposed value when 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_i is greater or equal than v_k. When a process decides, it moves to phase 1 ($PH1$). Task 3 allows process p_i to decide a value already decided by another process when a $(PH1, v_k)$ message is received. With task 4 process p_i decides its value v_i when the failure detector D ($\mathcal{L}$ in $HAS_f[\mathcal{L}, \emptyset, c/r]$) returns *true*, i.e., $D.output_i = true$. Note that at most $n-1$ processes can execute this task 4 to solve set agreement (from Condition 1 of Definition 2).

As links are not reliable (but fair-lossy) and processes may crash and recover, with task 5 process p_i guarantees the propagation of its decided value to the rest of processes (decision taken in either task 2, 3 or 4).

If a process p_i crashes and recovers meanwhile it is running the algorithm, then it always executes lines 29-38. If process p_i proposed a value v but it crashed before writing any estimate or decision value in $status_i$, then p_i will get the proposed value from the stable storage variable $PROP_i$ (line 31). In other case, v_i will obtain its estimate or

```
init:
(1)  status_i ← ⊥; % stable storage variable

propose_i(v): % by writing v into stable storage PROP_i
(2)  v_i ← v;
(3)  start tasks 1, 2, 3 and 4 %tasks 2, 3 and 4 executed atomically


task 1:
(4)  repeat forever each η time
(5)      broadcast (PH0, id(i), v_i);
(6)  end repeat

task 2:
(7)  when (PH0, id(k), v_k) is received:
(8)      if (⟨id(k), v_k⟩ ≤ ⟨id(i), v_i⟩) then
(9)          status_i ← v_k;
(10)         v_i ← v_k;
(11)         decide_i(v_k); % by writing v_k into stable storage DEC_i
(12)         stop tasks 1, 3 and 4;
(13)         start task 5
(14)     end if

task 3:
(15) when (PH1, v_k) is received:
(16)     status_i ← v_k;
(17)     v_i ← v_k;
(18)     decide_i(v_k); % by writing v_k into stable storage DEC_i
(19)     stop tasks 1, 2 and 4;
(20)     start task 5

task 4:
(21) when D.output_i=true: % returned by the failure detector D
(22)     status_i ← v_i;
(23)     decide_i(v_i); % by writing v_i into stable storage DEC_i
(24)     stop tasks 1, 2 and 3;
(25)     start task 5

task 5:
(26) repeat forever each η time
(27)     broadcast (PH1, v_i);
(28) end repeat

when process p_i recovers:
(29) if (propose_i() was invoked) then % by checking PROP_i
(30)     if (status_i ≠ ⊥) then v_i ← status_i
(31)     else v_i ← PROP_i
(32)     end if
(33)     if (decide_i() was invoked) then % by checking DEC_i
(34)         start task 5
(35)     else
(36)         start tasks 1, 2, 3 and 4 %tasks 2, 3 and 4 ex. atomically
(37)     end if
(38) end if
```

Fig. 1. The algorithm $\mathcal{A}_{set}$ for set agreement in $HAS_f[\mathcal{L}, \emptyset, c/r]$ (code for process p_i).

decided value from stable storage variable $status_i$ (line 30). If it has already proposed and decided a value, process p_i starts task 5 to propagate this decided value (line 34). If process p_i has proposed a value but it has not decided yet, it starts tasks 1, 2, 3 and 4 to look for a value to decide (line 36).

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

Due to space limitations, proofs are omitted here.

Lemma 1. *(Validity) For each run, if a process p_i of the system $HAS_f[\mathcal{L}, \emptyset, c/r]$ 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 is at most $n-1$.*

Lemma 3. *(Termination) For each run, every process $p_i \in Correct$ in the system eventually decides some value.*

Theorem 1. *The algorithm of Figure 1 implements set agreement in a system where $t = n$ and preserving Property 1 (Less-unstability).*

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 a system where $t = n$ and preserving Property 1.

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

We now 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. Therefore, we present in this section an implementation of $\mathcal{L}$ (denote it by $\mathcal{A}_{\mathcal{D}}(knowledge)$) for a synchronous system with homonymous processes, a different knowledge of the membership, and where until $t = n-1$ different processes can crash and recover.

It is worthy to note that $\mathcal{A}_{\mathcal{D}}(knowledge)$ does not need to use any stable storage variable.

5.1 Model

Let HSS be a system like HAS but with two differences: it is synchronous, and when a process p_i recovers, it does not need to know it, i.e., if p_i recovers, it starts to execute line 1 again. By synchronous we mean that the time to execute a step is bounded and known by every process, and 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.

The following property states that only $t = n-1$ processes can crash and recover in the system.

Property 2. (Up-permanency) For each run, some process has to be permanently-up in the system ($HSS_r[\emptyset, \emptyset, c/r]$, $HSS_r[\emptyset, Y, c/r]$, or $HSS^n_r[\emptyset, \emptyset, c/r]$).

This Property 2 in a system with the crash-stop model only states that until $t = n - 1$ different processes can crash. Note that all algorithms found in the literature that implement the loneliness failure detector $\mathcal{L}$ ([4], [20]) work in systems where processes can crash but not recover and where up to $t = n - 1$ processes can crash.

The algorithms presented in [20] and in [4] (let us denote them by $\mathcal{A}'$) implements the loneliness failure detector in $SS_r[\mathcal{L}, \Pi, c]$ when up to $n-1$ processes can crash. We show in this section that the algorithm of Figure 2 (called $\mathcal{A}_{\mathcal{D}}(knowledge)$) implements the failure detector $\mathcal{L}$ in $HSS_r[\emptyset, \emptyset, c/r]$, $HSS^n_r[\emptyset, \emptyset, c/r]$ and $HSS_r[\emptyset, Y, c/r]$ (when $|Y| \geq 2$, and two processes of Y have different identities) enriched with Property 2. Hence, from Definition 3, it is easy to see that $\mathcal{A}' \subset \mathcal{A}_{\mathcal{D}}(knowledge)$.

5.2 Algorithm $\mathcal{A}_{\mathcal{D}}(knowledge)$

In Figure 2 we present an algorithm that implements $\mathcal{L}$ with different grades of initial knowledge of the system membership, and without using any stable storage variable. The parameter $knowledge$ states the kind of system in which the algorithm is running. Then, if process p_i executes the algorithm with the argument $none$ in $knowledge$ (denoted by $\mathcal{A}_{\mathcal{D}}(none)$), process p_i is running in $HSS_r[\emptyset, \emptyset, c/r]$. Similarly, the argument $size$ in $knowledge$ ($\mathcal{A}_{\mathcal{D}}(size)$) states that it is running in $HSS^n_r[\emptyset, \emptyset, c/r]$, and the argument $partial$ in $knowledge$ ($\mathcal{A}_{\mathcal{D}}(partial)$) states that it is running in $HSS_r[\emptyset, Y, c/r]$, being $|Y| \geq 2$, and two processes of Y have different identities.

For each process p_i, $output_i$ is initially *false* (line 1). If $knowledge = partial$, process p_i knows at least two identifiers with different value because it is running in $HSS_r[\emptyset, Y, c/r]$, being $|Y| \geq 2$, and two processes of Y have different identities (these two known identifiers of Y with different value are $IDENT_1$ and $IDENT_2$ in Figure 2). Then, all process p_i whose identifier is neither $IDENT_1$ nor $IDENT_2$ changes to *true* (lines 2-6).

In task 1, every η time, each process p_i broadcasts $(alive, count_i)$ messages that arrive synchronously (at most Δ units of time later) to the rest of processes of the system (line 10). If $knowledge = size$, the variable $count_i$ is used by each process p_i to indicate the maximum number of processes that p_i believes may have had *true* in their variable $output$ at some moment of the run. After Δ units of time, process p_i analyzes the messages received (rec_i) to see if p_i sets $output_i$ to true (lines 12-26). 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 1 is executed again and $output_i$ is *false* another time. We have two cases to analyze depending on the value of $knowledge$:

- $knowledge = none$ or $knowledge = partial$. If the number of messages received is 0, then $output_i = true$ (lines 20-22).
- $knowledge = size$. Process p_i, using its variable aux_i, adds the values of $count$ from all $(alive, count)$ messages received (rec_i). Then, $count_i$ is updated by p_i with aux_i and the number of messages received (lines 13-19). If process p_i does not receive any message or $count_i$ is lesser than $(n-1)$, then it sets $output_i = true$ and increases $count_i$ in 1 (lines 23-26).

```
(re)init: % if p_i recovers, it starts executing line 1
(1)  output_i ← false;
(2)  if (knowledge = partial) then
           % IDENT_1 and IDENT_2 are two
           % identifiers known by all processes
(3)      if ((id(i) ≠ IDENT_1) ∧ (id(i) ≠ IDENT_2)) then
(4)          output_i ← true
(5)      end if
(6)  end if;
(7)  count_i ← 0;
(8)  start task 1

task 1:
(9)  repeat forever each η time
(10)     broadcast (alive, count_i);
(11)     wait Δ time;
(12)     let rec_i be the set of (alive, count) messages received;
(13)     if (knowledge = size) then
(14)         aux_i ← 0;
(15)         for_each (alive, count) ∈ rec_i do
(16)             aux_i ← aux_i + count;
(17)         end for_each;
(18)         count_i ← count_i + aux_i + |rec_i|;
(19)     end if
(20)     if ((knowledge ≠ size) ∧ (|rec_i| = 0)) then
(21)         output_i ← true
(22)     end if
(23)     if ((knowledge = size) ∧
             ((|rec_i| = 0) ∨ (count_i < n − 1))) then
(24)         output_i ← true;
(25)         count_i ← count_i + 1
(26)     end if
(27) end repeat
```

Fig. 2. Algorithm $\mathcal{A}_{\mathcal{D}}(knowledge)$ for process p_i to implement $\mathcal{L}$ (the parameter $knowledge$ establishes the initial knowledge of the system membership of p_i)

.

5.3 $\mathcal{A}_{\mathcal{D}}(partial)$, $\mathcal{A}_{\mathcal{D}}(size)$ and $\mathcal{A}_{\mathcal{D}}(none)$ implement $\mathcal{L}$

We show in this section that the algorithm $\mathcal{A}_{\mathcal{D}}$ executed with the arguments *partial*, *size* or *none* in the parameter $knowledge$ implements the failure detector $\mathcal{L}$ where $t = n-1$ (i.e., Property 2 has to be preserved).

The proofs of this section are not included here due to space limitations.

Theorem 2. *The algorithm $\mathcal{A}_{\mathcal{D}}(partial)$ implements the failure detector $\mathcal{L}$ in a system $HSS_r[\emptyset, Y, c/r]$, where $|Y| \geq 2$ and two processes of Y have different identities, enhanced with Property 2.*

Theorem 3. *The algorithm $\mathcal{A}_{\mathcal{D}}(size)$ implements the failure detector $\mathcal{L}$ in a system $HSS^n_r[\emptyset, \emptyset, c/r]$ enhanced with Property 2.*

Theorem 4. *The algorithm $\mathcal{A}_{\mathcal{D}}(none)$ implements the failure detector $\mathcal{L}$ in a system $HSS_r[\emptyset, \emptyset, c/r]$ enhanced with Property 2.*

6 Conclusions

We study the *set-agreement* problem in message passing systems with the weakest failure detector $\mathcal{L}$ in crash-recovery asynchronous systems with homonymous processes and without a complete initial knowledge of the membership. An implementation of set-agreement using $\mathcal{L}$ is presented for crash-recovery systems with homonyms and without initial knowledge of 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). IEEE 32nd IEEE Int. Conf. on Distributed Computing Systems (ICDCS), China, June 2012. To appear.
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. DISC, ISBN of LNCS: 978-3-642-15762-2, vol. 6343, pp. 206–220, 2010.
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. C. Delporte-Gallet, H. Fauconnier and S. Toueg. The minimum information about failures for solving non-local tasks in message-passing systems. Distributed Computing vol. 24(5), pp. 255–269, 2011.
15. 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.
16. M. Herlihy and N. Shavit. The topological structure of asynchronous computability. Journal of the ACM 46(6), pp. 858–923, 1999.
17. 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.
18. E. Jiménez, S. Arévalo, A. Fernández. Implementing unreliable failure detectors with unknown membership. Information Processing Letters 100 (2), pp. 60–63, 2006.
19. 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.
20. 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.
21. M. Raynal. Communication and Agreement Abstractions for Fault-Tolerant Asynchronous Distributed Systems. Morgan & Claypool Publishers, 250 pages, 2010.
22. 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.