Technical Report

UCAM-CL-TR-694
ISSN 1476-2986

Number 694

Anti-Ω: the weakest failure detector for set agreement

Piotr Zieliński

July 2007

ISSN 1476-2986

Anti-Ω: the weakest failure detector for set agreement

Piotr Zieliński
piotr.zielinski@cl.cam.ac.uk
Cavendish Laboratory, University of Cambridge, UK

Abstract

In the set agreement problem, n processes have to decide on at most $n-1$ of the proposed values. This paper shows that the anti-Ω failure detector is both sufficient and necessary to implement set agreement in an asynchronous shared-memory system equipped with registers. Each query to anti-Ω returns a single process id; the specification ensures that there is a correct process whose id is returned only finitely many times.

1 Introduction

In the set agreement problem, n processes have to decide on at most $n-1$ proposed values [5]. Set agreement has long been believed to be one of the weakest non-anonymous problem that is not wait-free implementable in an asynchronous system. Recently, the problem of finding the weakest failure detector to implement it received considerable attention [6, 14, 23]. This paper presents the solution: it shows that the detector "anti-Ω" [26] is both sufficient and necessary to implement set agreement in a shared-memory system equipped with registers.

Each query to the anti-Ω detector returns a process id. The detector guarantees that there is a *correct* process whose id will be returned only finitely many times. In other words, this process id will be eventually never output by the detector. Anti-Ω might not stabilize: it is possible that more than one process id will be returned infinitely often. Anti-Ω is the weakest non-implementable eventual failure detector, that is, any non-implementable eventual failure detector can implement anti-Ω [26].

Essentially, this paper makes two complementary claims: that anti-Ω is both *sufficient* and *necessary* to implement set agreement. Section 3 shows sufficiency by presenting an anti-Ω based algorithm that can implement set agreement in any environment equipped with registers. Section 5 uses shows that anti-Ω is required to implement any non-wait-free-implementable problem, including set agreement. The method used is inspired by the simulation method of [4].

Section 2 defines set agreement, and establishes the system model, which is then refined in Section 4. Section 6 discusses the consequences of the results showed here, and suggests some open questions and possible directions of future work.

2 System model and problem statement

The system considered in this paper is a standard shared-memory model. It consists of a fixed number n of processes $p_1, \ldots, p_n$, which communicate using some number of shared read-write registers. In addition to performing read and write operations, a process can query any of the available failure detectors. For example, Section 3 assumes that each process has access to the anti-Ω detector [26].

Each query to anti-Ω returns a process id. The detector guarantees that, in any run, some correct process id will eventually never be returned. In other words, there is a correct process whose id is returned only a finite number of times.

In k-set agreement, each process can propose a single value. The following requirements must be met [5]

Validity. Each decision was proposed by some process.

Agreement. There are at most k different decisions.

Termination. Eventually all correct processes will decide.

Set agreement is k-set agreement with $k = n - 1$, because this is the highest non-trivial k; in n-set agreement, each process can just decide on its own proposal. Set agreement is not wait-free implementable in a purely asynchronous system [2, 18, 24].

3 Anti-Ω implements set agreement

This section shows an implementation of set agreement using anti-Ω. It consists of two steps: (i) using anti-Ω to implement an equivalent detector vector-Ω, and then (ii) using vector-Ω to implement set agreement.

3.1 Implementing vector-Ω using anti-Ω

Vector-Ω is a vector of $n - 1$ subdetectors $\Omega_1, \ldots, \Omega_{n-1}$, each returning a single process id for each query. At least one Ω_i is correct, that is, eventually keeps returning the same correct process for all queries [4]; the other Ω_j's can behave arbitrarily. The processes do not know which Ω_i is/are correct.

Vector-Ω can implement anti-Ω by always outputting a process that is *not* among the $\leq n - 1$ processes selected by the subdetectors $\Omega_1, \ldots, \Omega_{n-1}$. By definition, at least one Ω_i will eventually keep outputting the same correct process p. Process p will therefore eventually never be output by the emulated anti-Ω, which completes the proof.

Using anti-Ω to implementing vector-Ω is slightly more complicated. Figure 1 presents an algorithm inspired by the order oracle [26]. Each process p_i maintains a single-writer register $counters_i$, in which it stores an n-element vector. Each element $counters_i[k]$ counts the number of times the anti-Ω detector at process p_i returned process p_k (lines 2–4).

Lines 5–10 emulate vector-Ω at each process p_i. First, the process reads the *counter* vectors from all processes and sums them into a temporary local vector *total*. Roughly speaking, $total[k]$ is the number of times anti-Ω returned p_k so far, at any process. Lines

```
counters_i ← [0, ..., 0]                              { single-writer register, n entries }
loop forever
  output ← anti-Ω detector output
  increment counters_i[output]                                 { atomicity not required }
when vector-Ω is queried do
  for k = 1, ..., n do
    total[k] ← counters_1[k] + ··· + counters_n[k]             { atomicity not required }
  let q_1, ..., q_k, ..., q_n be the permutation of {p_1, ..., p_n}
    ordered wrt increasing total[k] (ties broken deterministically)
  return [q_1, ..., q_{n-1}] as the outputs of Ω_1, ..., Ω_{n-1}
```

Figure 1: Implementing vector-Ω with anti-Ω.

```
instance Consensus_i uses Ω_i (for i = 1, ..., n − 1)
function setagreement(v) is
  for i = 1, 2, ..., n − 1 do
    propose v to Consensus_i
  wait until some Consensus_i decides, say on v'
  return v'
```

Figure 2: Implementing set agreement with vector-Ω.

8–10 order the list of all processes with respect to increasing $total[k]$, and return all but the last as the output of vector-Ω.

Theorem 3.1. *The algorithm in Figure 1 implements vector-Ω.*

Proof. Let F be the set of processes that anti-Ω outputs only finitely many times. At any correct processes, all entries $total[k]$ with $k \notin F$ will be then increasing without limit. On the other hand, entries $total[k]$ with $k \in F$, will eventually stop increasing, and be the same at all correct processes. As a result, eventually the prefix $q_1, \ldots, q_{|F|}$ of $q_1, \ldots, q_n$ (lines 8–9) will consists of all processes in F, and will be the same at all correct processes.

Anti-Ω guarantees that F contains at least one correct process, say p. Therefore, there is an $i \leq |F|$ such that eventually always $q_i = p$. To finish the proof, we must show that $|F| \leq n - 1$, but is true because anti-Ω must output at least one process id infinitely often. □

3.2 Implementing set agreement using vector-Ω

Figure 2 shows a simple implementation of set agreement using vector-Ω. Each process proposes its value v to $n - 1$ independent parallel instances of Consensus [19]. Each instance $Consensus_i$ uses Ω_i provided by vector-Ω, which is sufficient to implement Consensus [4, 11, 17, 19]. Since at least one Ω_i behaves like Ω, at least one instance $Consensus_i$ will decide (Termination), on the value proposed by one of the processes (Validity). Each

$Consensus_i$ decides on at most one value, so the total number of different decision cannot exceed $n-1$ (Agreement). Note that even instances $Consensus_j$ with Ω_j that behave arbitrarily cannot violate safety properties (Validity and Agreement), because this misbehaviour cannot be discovered by examining a finite prefix of a run [15].

3.3 k-vector-Ω and k-set agreement

The algorithm in Figure 2 can be easily generalized to implement k-set agreement, by replacing all instances of $n-1$ with k. This includes using k-vector-Ω detector, which is the same as vector-Ω, but consists of k subdetectors Ω_i, instead of $n-1$. Vector-Ω is k-vector-Ω with $k = n-1$. Note that n-vector-Ω is trivially implementable by having each Ω_i constantly output process p_i.

3.4 Detectors anti-Ω and Υ

Guerraoui et al. [14] proposed the failure detector Υ and showed that it is capable of implementing set agreement. Each query to Υ returns a non-empty set of processes. The detector guarantees that, eventually, all queries will be returning the same non-empty set T of processes, which is *different* from the set of correct processes. Detector Υ is the weakest non-implementable *stable* eventual failure detector [14].

Anti-Ω is the weakest (not necessarily stable) non-implementable eventual failure detector, which is strictly weaker than Υ [26].

It is fairly easy to see a special case of this result, namely, that Υ is no weaker than anti-Ω. Assume that each process p_i has a local instruction counter, which it periodically writes to a dedicated shared single-writer register $counter_i$. Anti-Ω can then be implemented by always returning the member p_j of the latest Υ's output with the lowest $counter_j$. If the eventual stable output T of Υ contains a faulty process, then only faulty processes will eventually be output by the emulated anti-Ω (because their *counter*'s eventually stop increasing). Otherwise, there is a correct $p \notin T$, so this p will eventually never be output. In both cases, the anti-Ω requirements are satisfied.

4 System model again

In the rest of the paper, I will be considering an equivalent, but more precisely specified, model. As before, we operate in a shared memory system consisting of n processes $p_1, \ldots, p_n$. This time, however, processes communicate using immediate atomic snapshot memory, which is powerful yet still equivalent to the ordinary register model [3].

4.1 Immediate atomic snapshot model

The immediate atomic snapshot model, depicted in Figure 3, consists of n processes and the main memory, which contains n single-writer registers $reg_1, \ldots, reg_n$, one per process. From each process p_i's point of view, a run is a sequence of alternating steps (i) process p_i uses the provided memory snapshot $shapshot_i$ to compute the text value $towrite_i$ to be written to reg_i, (ii) the memory performs the requested write and provides the process with $snapshot_i = [reg_1, \ldots, reg_n]$. The value $token_i \in \{\mathsf{process}, \mathsf{memory}\}$

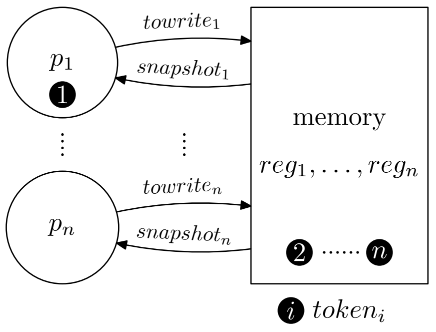

Figure 3: Immediate snapshot model diagram.

```
initially for each $p_i$ is
  $token_i =$ process
  $snapshot_i = [reg_1, \ldots, reg_n]$        { initial $reg_i$ specified by the application }
  $towrite_i = \bot$
action $a$ of kind $update(B)$ is
  enabled if $token_i =$ memory for all $p_i \in B$
    for all $p_i \in B$ do $reg_i \leftarrow towrite_i$
    for all $p_i \in B$ do $snapshot_i \leftarrow [reg_1, \ldots, reg_n]$
    for all $p_i \in B$ do $token_i \leftarrow$ process
action $a$ of kind $step(p_i)$ is
  enabled if $token_i =$ process
    $towrite_i \leftarrow localcomp_i(snapshot_i, external(a))$
    $token_i \leftarrow$ memory
```

Figure 4: Immediate snapshot model specification.

indicates whether the next step should be taken by p_i or the memory. Initially, all tokens are at the processes, each $snapshot_i = [reg_1, \ldots, reg_n]$, and the initial $reg_1, \ldots, reg_n$ are specified by the application.

Figure 4 gives the formal description of the system. The system progresses by executing two kinds of actions: $update(B)$ executed by the memory, and $step(p_i)$ executed by each process p_i. The action $update(B)$, with B being a non-empty *block* (set) of processes is enabled, if all processes in B are waiting for the memory (nothing is said about $token_j$ for $p_j \notin B$). Action $update(B)$ performs all the writes requested by processes in B, takes a snapshot for all of them, and moves the corresponding tokens back to the processes.

Action $step(p_i)$ is enabled if p_i has its token. It uses the provided $snapshot_i$ to compute the next value $towrite_i$ to be written to the memory, and moves the token back to the memory. This computation is performed by a function $localcomp_i$, which can be thought of as the algorithm being executed. The meaning of $external(a)$ is explained in Section 4.2 below.

4.2 Actions and runs

We need to distinguish between action kinds, such as "$\mathit{update}(\{p_1, p_3\})$", and individual actions, for example, "the second $\mathit{update}(\{p_1, p_3\})$ in the current run". The difference is similar to that between a function and a function invocation. An *action kind* is a piece of code describing how an action of this kind modifies the state. An *action* is an opaque object, with a unique identity within a *run*, which has its kind accessible through the predicate *kind*. Action a is executed by determining $\mathit{kind}(a)$ and then modifying the system state according to Figure 4. For brevity, "let $a = \textit{some-kind}$" is sometimes used to mean "let a be any action with $\mathit{kind}(a) = \textit{some-kind}$".

A *run* $\mathcal{A}$ is a set $\mathit{actions}(\mathcal{A})$ of actions, together with mappings $\mathit{time}_{\mathcal{A}}$ and $\mathit{external}_{\mathcal{A}}$. For each action a, the value $\mathit{time}_{\mathcal{A}}(a)$ is the time at which a is executed. No two actions have the same $\mathit{time}_{\mathcal{A}}(a)$. Function $\mathit{external}_{\mathcal{A}}(a)$ is the external information available to action a, such as the failure detector query result, which might refer to the current time $\mathit{time}_{\mathcal{A}}(a)$. The run is *purely asynchronous* if there is no external information, that is, $\mathit{external}_{\mathcal{A}}(a) = \perp$ for all actions a. Executing a run consists of executing all actions $a \in \mathit{actions}(\mathcal{A})$ atomically in the order of increasing $\mathit{time}_{\mathcal{A}}(a)$. When $\mathcal{A}$ is clear from the context, I drop the subscript from $\mathit{time}_{\mathcal{A}}$, $\mathit{external}_{\mathcal{A}}$, etc.

In any run $\mathcal{A}$, processes can fail by crashing, but the memory never crashes. Nodes that never crash are *correct*, the others are *faulty*. Let $\mathit{correct}(\mathcal{A})$ be the set of correct processes, and $\mathit{inf}(\mathcal{A}) \subseteq \mathit{correct}(\mathcal{A})$, the set of processes that perform infinitely many steps. A run $\mathcal{A}$ is *fair* if no action of a correct node is enabled forever without being executed; or, equivalently, iff $\mathit{inf}(\mathcal{A}) = \mathit{correct}(\mathcal{A})$.

This paper uses the following symbols for actions of the following kinds

a	any action	s_i	$\mathit{step}(p_i)$
s	$\mathit{step}(p)$ for some p	u_B	$\mathit{update}(B)$
u	$\mathit{update}(B)$ for some B	u_i	$\mathit{update}(B)$ with $p_i \in B$

In addition, a^k denotes the k-th action in the particular group, for example, s_i^k is the k-th step of process p_i, and u^k is the k-th update action. Each such symbol uniquely identifies an action in a run.

Some algorithms in this paper manipulate information about step actions s_i^k. Each s_i^k can be represented as a pair (i, k), which and uniquely identifies the action within the run as the k-th step of p_i. This method is analogous to representing a process p_i by its id i, which also uniquely identifies it.

4.3 Causal precedence relation

The heart of the proof in Section 5 lies in simulating a failure detector in purely asynchronous runs. To show the correctness of such a simulation, we will have to prove that the run with a simulated detector is indistinguishable from some (possibly unfair) run with a real detector. For this reason, we need a precise notion of *causal precendence*, introduced below.

Let $\mathit{view}(a)$ be the set of all state variables accessed by action a, including those in

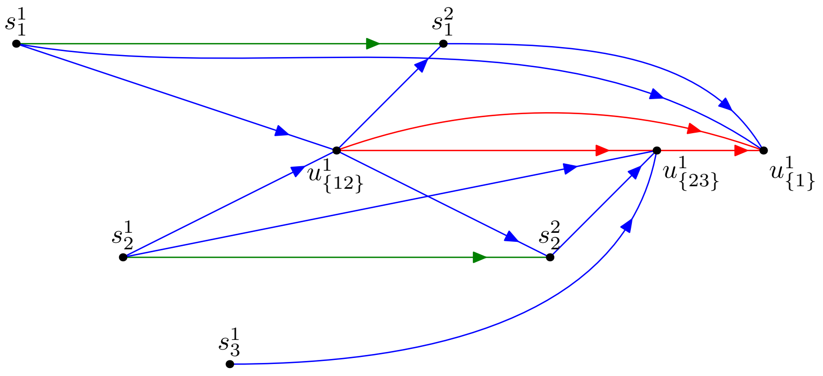

Figure 5: Relations $\rightsquigarrow$ (arcs) and $\overset{*}{\rightsquigarrow}$ (paths) in the run s_1^1, s_2^1, s_3^1, $u_{\{12\}}^1$, s_1^2, s_2^2, $u_{\{23\}}^1$, $u_{\{1\}}^1$.

enabledness tests of a. The value of $\mathit{view}(a)$ depends solely on $\mathit{kind}(a)$. In our case,

$$\begin{aligned}\mathit{view}(u_B) &= \{\mathit{reg}_1, \ldots, \mathit{reg}_n\} \cup \{\, \mathit{token}_i, \mathit{towrite}_i, \mathit{snapshot}_i \mid p_i \in B\,\},\\ \mathit{view}(s_i) &= \{\mathit{token}_i, \mathit{towrite}_i, \mathit{snapshot}_i\},\end{aligned}$$

where $\mathit{kind}(u_B) =$ "*update*(B)" and $\mathit{kind}(s_i) =$ "*step*(p_i)".

Two actions a and a' *conflict* if they access the same state variable, that is, $\mathit{view}(a) \cap \mathit{view}(a') \neq \emptyset$. In our case, the following pairs of actions conflict: (i) any two update actions, u and u', (ii) a step action s_i and an update action u_i, both involving p_i, (iii) two step actions, s_i and s_i', by the same process p_i. Let us define

$$a \rightsquigarrow a' \qquad \overset{\text{def}}{\Longleftrightarrow} \qquad \mathit{time}(a) < \mathit{time}(a') \ \wedge\ \mathit{view}(a) \cap \mathit{view}(a') \neq \emptyset.$$

The causal precedence relation $\overset{*}{\rightsquigarrow}$ is the transitive closure of $\rightsquigarrow$, and $\overset{0*}{\rightsquigarrow}$ is the reflexive closure of $\overset{*}{\rightsquigarrow}$.

As an example, consider the run

$$s_1^1, \quad s_2^1, \quad s_3^1, \quad u_{\{12\}}^1, \quad s_1^2, \quad s_2^2, \quad u_{\{23\}}^1, \quad u_{\{1\}}^1,$$

where s_i^k is the k-th step of process p_i, and u_B^k is the k-th action *update*(B). In Figure 5, nodes represent actions, and arcs represent $\rightsquigarrow$, divided in three above categories of conflicting actions: (i) red, (ii) blue, (iii) green. The paths correspond to $\overset{*}{\rightsquigarrow}$, for example, $s_2^1 \overset{*}{\rightsquigarrow} u_{\{1\}}^1$ because $s_2^1 \rightsquigarrow s_2^2 \rightsquigarrow u_{\{23\}}^1 \rightsquigarrow u_{\{1\}}^1$.

Lemma 4.1. *Any runs $\mathcal{A}$, with the same actions$(\mathcal{A})$, relation $\overset{*}{\rightsquigarrow}_{\mathcal{A}}$, and mapping external$_{\mathcal{A}}$ are indistinguishable.*

Lemma 4.2. *In a fair run, for any step s_j and any correct p_i, there is a step s_i such that $s_j \overset{*}{\rightsquigarrow} s_i$.*

(All proofs missing from the main text are in the appendix.)

Block runs

Consider the sequence of all update actions $u^k = \mathit{update}(B_k)$ in a run. Lemma 4.3 shows that the sequence of blocks $B_1, B_2, \ldots$ uniquely determines the causal precedence relation $\overset{*}{\rightsquigarrow}$. If the system is purely asynchronous ($\mathit{external}(a) = \perp$ for all actions a), then, by Lemma 4.1, $\overset{*}{\rightsquigarrow}$ determines the entire run. As a result, the sequence $B_1, B_2, \ldots$ can be treated as a run, called a *block run* [1]. Note that this simplification applies only to purely asynchronous systems, without failure detection.

Lemma 4.3. *The block sequence $B_1, B_2, \ldots$ determines $\overset{*}{\rightsquigarrow}$.*

4.4 Failure detection

A *failure pattern* is a function $\mathit{alive}(t)$, which returns the set of non-crashed processes at any given time $t \in \mathbb{R}$. Crashed processes do not recover, therefore $t < t' \implies \mathit{alive}(t) \supseteq \mathit{alive}(t')$. A run is consistent with a failure pattern *alive* if crashed processes do not take steps, that is,

$$p_i \in \mathit{alive}(\mathit{time}(s_i)) \quad \text{for all steps } s_i \in \mathit{actions}(\mathcal{A}). \tag{1}$$

Note that consistency does not require fairness: there might be correct processes that take only finitely many steps ($\mathit{inf}(\mathcal{A}) \subseteq \mathit{correct}(\mathcal{A}) = \bigcap_t \mathit{alive}(t)$).

A *failure detector history* is a function $\mathit{history}(p, t)$, which returns the result of process p querying the detector at time t. Therefore, in systems equipped with a failure detector, each step s satisfies

$$\mathit{external}(s_i) = \mathit{history}(p_i, \mathit{time}(s_i)). \tag{2}$$

Finally, a failure detector specification is a function that for each possible failure pattern *alive* returns the set of possible histories *history*. Note that the failure detector behaviour depends only on the failure pattern, and not, for example, on when processes take steps.

5 Set agreement requires anti-Ω

Section 3 showed that the anti-Ω failure detector is *sufficient* to implement set agreement. This section will show that anti-Ω is also *necessary*. In other words, any failure detector Δ sufficient to implement a non-wait-free-implementable abstraction, such as set agreement, can implement anti-Ω.

More precisely, consider any abstraction Problem with the following properties: (i) no algorithm can implement Problem in all fair runs in the purely asynchronous system, (ii) there is an algorithm $\mathsf{Algorithm}_\Delta$ that implements Problem in all fair runs of a system equipped with a failure detector Δ, (iii) the termination condition of Problem is a function of the state of the system (eg. it does not require knowing which processes are correct, see below). This section proves that such a detector Δ can implement anti-Ω.

The proof consists of three parts:

1. Collecting Δ samples from the current run $\mathcal{A}$ (Section 5.1).

2. Using this information to safely simulate Δ in any run $\mathcal{B}$ (Section 5.2).

3. Simulating all possible runs $\mathcal{B}$ in order to emulate anti-Ω (Section 5.3).

```
initially dom $prec_i = \emptyset$, dom $det_i = \emptyset$                    { empty mappings }
function $localcomp_{\text{info}}([(prec_1, det_1), \ldots, (prec_n, det_n)], external)$ at step $s_i$ is
  $prec_i(s_i) \leftarrow \mathrm{dom}\, prec_1 \cup \cdots \cup \mathrm{dom}\, prec_n$
  $det_i(s_i) \leftarrow external$
  return $(prec_i, det_i)$
```

Figure 6: Computing mappings $prec_i$ and det_i at process p_i.

Correctness-independent termination condition for set agreement

The original Termination condition of set agreement refers to process correctness, violating property (iii) of Problem. However, any set agreement protocol can be transformed in the following way: (i) when a process p_i decides, it writes its decision to a special decision register dec_i (emulated as part of reg_i), initially empty; and (ii) each process repeatedly scans all registers dec_i and, if any of them is non-empty, decides on its contents. This modification obviously preserves all properties of set agreement.

In fair runs, standard Termination ("eventually all correct processes decide") is equivalent to 1-Register Termination ("eventually at least one dec_i is non-empty", p_i possibly faulty). This is because, if a correct process p_i decides, then it will write its decision to dec_i. On the other hand, if some dec_i is set, then all correct processes will eventually read it and decide. (This reasoning requires at least one correct process.)

5.1 Collecting failure detector samples

The algorithm in Figure 6, defined by the function $localcomp_{\text{info}}$ (Figure 4, line 12), gathers information about the behaviour of the detector Δ, and the causality relation $\overset{*}{\leadsto}$ between steps of the current run. Each register reg_i maintains two mappings, $prec_i$ and det_i, both mapping only the steps s_i made so far by process p_i. For each such step, $det_i(s_i)$ is the the output of Δ during s_i, and $prec_i(s_i)$ is the set of all steps s_j (by any process p_j) that causally precede s_i:

$$det_i(s_i) = external(s_i), \qquad prec_i(s_i) = \{\, s_j \mid s_j \overset{*}{\leadsto} s_i \,\}.$$

For example, in the run in Figure 5, we have

$$prec(s_1^1) = prec(s_2^1) = prec(s_3^1) = \emptyset, \quad prec(s_1^2) = prec(s_2^2) = \{s_1^1, s_2^1\},$$

where $prec(s_i) \overset{\text{def}}{=} prec_i(s_i)$.

Both $prec_i$ and det_i start empty. At each step s_i, they are updated by adding a new entry corresponding to s_i: $det_i(s_i)$ becomes the current output of Δ, whereas $prec_i(s_i)$ becomes the union of the domains of all mappings $prec_1$, ..., $prec_n$. Lemma 5.1 shows that each $\mathrm{dom}\, prec_j = \{\, s_j \mid s_j \overset{*}{\leadsto} s_i \,\}$. As a result, in line 3,

$$prec_i(s_i) \leftarrow \bigcup_{p_j} \mathrm{dom}\, prec_j = \bigcup_{p_j} \{\, s_j \mid s_j \overset{*}{\leadsto} s_i \,\} = \{\, s_j \mid s_j \overset{*}{\leadsto} s_i \,\}.$$

Both $prec_i$ and det_i are *growing mappings*. In general, a growing mapping X_i is a mapping that (i) starts empty at all processes, (ii) each step s_i adds a new entry $X_i(s_i)$.

```
1  **initially** $\operatorname{dom} map_i = \emptyset$                         { *empty mapping* }
2  **function** $successor_i(used)$ **is**                          { *steps* $used \subseteq actions(A)$ }
3    $detector_{\text{anti-}\Omega} \leftarrow p_i$
4    **for** $k = 1, 2, \ldots$ **do**
5      **wait until** $s_i^k \in \operatorname{dom} prec_{\mathcal{A}}$           { $s_i^k$ *is the k-th step of process* $p_i$ *in* $\mathcal{A}$ }
6      **if** $used \subseteq prec_{\mathcal{A}}(s_i^k)$ **return** $s_i^k$
7  **function** $localcomp_{\text{async}}([(map_1, data_1), \ldots, (map_n, data_n)])$ at step $s_i$ **is**
8    $map_i(s) \leftarrow successor_i(\operatorname{range} map_1 \cup \cdots \cup \operatorname{range} map_n)$
9    $data_i \leftarrow localcomp_{\Delta}\big([data_1, \ldots, data_n], det_{\mathcal{A}}(map_i(s))\big)$
10   **return** $(map_i, data_i)$
```

Figure 7: Emulating failure detector Δ at process p_i.

The mappings $X_1, \ldots, X_n$ can be thought of as fragments of a single composite mapping X defined as $X \overset{\text{def}}{=} \bigcup_i X_i$, that is, $X(s_i) \overset{\text{def}}{=} X_i(s_i)$. The symbol $\operatorname{dom}_s X_i$ denotes the value of $\operatorname{dom} X_i$ passed to *localcomp* at step s.

Lemma 5.1. *If X is a* growing mapping, *then* $\operatorname{dom}_{s_i} X_j = \{\, s_j \mid s_j \overset{*}{\rightsquigarrow} s_i \,\}$.

5.2 Simulating the failure detector in a given run

Section 5.1 showed how to collect failure detector Δ samples $det(s)$ and the causal precedence relation $prec(s)$ for a given run $\mathcal{A}$. This section shows how to use this information to simulate Δ in a purely asynchronous run $\mathcal{B}$. This simulation is indistinguishable from some run $\mathcal{C}$ with a real Δ. FLP and similar results are not violated because $\mathcal{C}$ is not necessarily fair, and a simulated Δ query might not terminate. One possible interpretation is that the simulation is safe but not necessarily live, for example, the Ω detector can be simulated by always returning a fixed process.

The algorithm in Figure 7 defines a local computation function $localcomp_{\text{async}}$ that allows us to run any algorithm $localcomp_{\Delta}$ that uses Δ. The detector Δ is simulated using a function *map*, which maps steps in the current run $\mathcal{B}$, in which Δ is emulated, to steps in the run $\mathcal{A}$, in which samples of Δ were collected.

The code in Figure 7 computes *map* and executes the algorithm $localcomp_{\Delta}$ at the same time. At every point in time, each reg_i stores two pieces of data: the algorithm data $data_i$, and a growing mapping map_i, the fragment of $map = map_1 \cup \cdots \cup map_n$. At each step s_i, process p_i first uses the current values of $map_1, \ldots, map_n$ to compute $map_i(s_i)$. The real Δ output at $map(s)$ in run $\mathcal{A}$, that is $det_{\mathcal{A}}(map(s_i))$, is then used as the simulated Δ output for the algorithm $localcomp_{\Delta}$ in the current run $\mathcal{B}$ (line 9).

The computation of $map_i(s)$ is performed by the function $successor_i$, which takes the values of the mappings map_i ($\bigcup_j \operatorname{range} map_j$), and returns the earliest step of process p_i that causally follows all of them in $\mathcal{A}$. To achieve this, $successor_i$ considers all steps s_i^1, s_i^2, ... in this order. For each s_i^k, it tests whether all steps in $used = \bigcup_j \operatorname{range} map_j$ causally precede it in $\mathcal{A}$, and if so return it as the value for $map(s)$ (line 6). Line 3 is part of the anti-Ω emulation algorithm, and will be explained in Section 5.3.

The **wait** instruction in line 5 ensures that the $prec_{\mathcal{A}}(s_i^k)$ information is available in line 6, which otherwise might not be if p_i performed only finitely many steps in $\mathcal{A}$. It is also intended to cover the possibility that the entries of $prec_{\mathcal{A}}$ are supplied to process p_i one by one from some external source, rather than given to it all at once at the beginning of the algorithm. Section 5.3 contains more details.

Function $successor_i$ might not terminate, for two reasons. First, because of the **wait** instruction in line 5. Second, because of the infinite loop in lines 4–6 and the exit condition in line 6 holding for no k. Again, Section 5.3 will explain why this is not a problem.

Lemma 5.2. *Mapping map preserves causality:* $s_j \overset{*}{\rightsquigarrow}_{\mathcal{B}} s_i \implies map(s_j) \overset{*}{\rightsquigarrow}_{\mathcal{A}} map(s_i)$.

Proof. By Lemma 5.1, $s_j \overset{*}{\rightsquigarrow}_{\mathcal{B}} s_i$ implies $s_j \in \mathrm{dom}_{s_i} map_j$. Therefore, at the beginning of step s_i, we have $map(s_j) = map_j(s_j) \in \mathrm{range}\, map_j \subseteq \bigcup_j \mathrm{range}\, map_j = used$, so $successor_i$ ensures that $map(s_j) \overset{*}{\rightsquigarrow}_{\mathcal{A}} map(s_i)$ (line 8). □

Theorem 5.3. *Let map be a function $actions(\mathcal{B}) \to actions(\mathcal{A})$ satisfying Lemma 5.2. Then, the run $\mathcal{B}$ with Δ simulated by the algorithm in Figure 7 is indistinguishable from some run $\mathcal{C}$ with a real Δ, with $inf(\mathcal{C}) = inf(\mathcal{B})$ and $correct(\mathcal{C}) = correct(\mathcal{A})$. Run $\mathcal{C}$ may be unfair.*

5.3 Emulating anti-Ω

Recall, from the beginning of Section 5, that Problem is an abstraction that is not wait-free-implementable in a purely asynchronous system, but is wait-free-implementable with a detector Δ using an algorithm $\mathsf{Algorithm}_{\Delta}$ (both assuming fair runs). This section will show that the detector emulation techniques from previous sections can be used implement anti-Ω using Δ.

The anti-Ω emulation process consists of two concurrent tasks (Figure 8). The first task uses the code from Figure 6 to continuously gather information about the behaviour of failure detector Δ in the current run $\mathcal{A}$, by updating $prec_{\mathcal{A}}$ and $det_{\mathcal{A}}$ (Section 5.1).

The second task simulates $\mathsf{Algorithm}_{\Delta}$ in all possible asynchronous runs $\mathcal{B}$, with Δ simulated using the technique from Section 5.2, and the information $prec_{\mathcal{A}}$ and $det_{\mathcal{A}}$ being collected by the first task. Since each simulated run $\mathcal{B}$ is purely asynchronous, at least one them does not decide (Lemma 5.6). Since at least one correct process eventually does not participate in this non-deciding run (Lemma 5.7), outputting only processes taking steps processes emulates anti-Ω (Theorem 5.8).

Each purely asynchronous run $\mathcal{B}$ is equivalent to some block run $B_1, B_2, \ldots$ (Section 4.3). Function *explore* in Figure 8 recursively enumerates and examines all such runs. More precisely, $explore(S_k, P_k)$ examines all runs, starting in the global system state S_k, in which only processes in P_k take steps. To generate all runs $\mathcal{B}$, we call $explore(S_0, P_0)$, where S_0 is the initial state of the simulated system, and $P_0 = P$ is the set of all processes (lines 11–14).

Function $explore(S_k, P_k)$ first checks whether a decision has been made in S_k (1-Register Termination). If so, it returns, because we are searching for a non-deciding run. Otherwise, we consider all non-empty $P_{k+1} \subseteq P_k$ in any deterministic order consistent with "$\subseteq$", for example, in the order of increasing $|P_{k+1}|$. For each such P_{k+1}, we recursively examine all blocks B_{k+1} with $B_{k+1} \subseteq P_{k+1}$. The reason for considering small P_{k+1}'s first is to prefer runs with a small number of correct processes (see below).

```
1  function $explore(S_k, P_k)$ is
2    if no decision in state $S_k$ then                    { test 1-Register Termination }
3      for all non-empty $P_{k+1} \subseteq P_k$ in an order consistent with "$\subseteq$" do
4        for all non-empty $B_{k+1} \subseteq P_{k+1}$ do
5          simulate $\mathsf{Algorithm}_\Delta$ from state $S_k$ and record the new state as $S_{k+1}$
6            for all $p_i \in B_{k+1}$ do simulate $step(p_i)$
7            simulate $update(B_{k+1})$
8          $explore(S_{k+1}, P_{k+1})$
9  task information gathering is
10   continuously update $prec_{\mathcal{A}}$ and $det_{\mathcal{A}}$ using the algorithm in Figure 6
11 task anti-$\Omega$ emulation is
12   $P_0 \leftarrow P$, where $P = \{p_1, \ldots, p_n\}$
13   $S_0 \leftarrow$ the initial state of the simulated system with $\mathsf{Algorithm}_\Delta$
14   $explore(S_0, P_0)$
```

Figure 8: Emulating anti-Ω using Δ and simulated runs of $\mathsf{Algorithm}_\Delta$.

For each choice of P_{k+1} and B_{k+1}, the algorithm simulates block B_{k+1} in state S_k by first making all processes $p \in B_{k+1}$ take a step, and then simulating $update(B_{k+1})$ itself. We call the new state S_{k+1}, and repeat the procedure recursively. Some runs $\mathcal{B}$ may be generated more than once, but this is not a problem.

Each invocation of $step(p_i)$ in line 6 invokes the detector simulation from Figure 7, which sets the current output of the emulated anti-Ω to p_i (line 3). The next section shows that this algorithm correctly emulates anti-Ω.

Correctness of the anti-Ω emulation

Lemma 5.4. *In fair runs, the executions of algorithm in Figure 8 are the same at all correct processes.*

Proof. The only process-dependent action in the algorithm is step simulation from Figure 7, which uses process-and-time-dependent values $\operatorname{dom} prec_{\mathcal{A}} = \operatorname{dom} det_{\mathcal{A}}$. Therefore, the only difference can be caused by line 5 in Figure 7 terminating at some correct processes but not on others.

We therefore need to show that if $s_i^k \in \operatorname{dom} prec_{\mathcal{A}}$ holds at some correct process, then it will eventually hold at all correct processes. Note that $s_i^k \in \operatorname{dom}_s prec_{\mathcal{A}}$ at step s of a correct process p means $s_i^k \overset{*}{\rightsquigarrow} s$ (Lemma 5.1). By Lemma 4.2, for any correct process p', there is a step s' with $s_i^k \overset{*}{\rightsquigarrow} s \overset{*}{\rightsquigarrow} s'$. Again by Lemma 5.1, this implies $s_i^k \in \operatorname{dom}_{s'} prec_{\mathcal{A}}$, which implies the assertion. □

As a consequence, it is sufficient to focus on the algorithm behaviour at a single correct process. First, note that the simulation might get stuck in function $successor_i$ in Figure 7. This can happen only if process p_i crashes in run $\mathcal{A}$ (Lemma 5.5). In this case, line 3 in Figure 7 ensures that the detector will eventually keep outputting a faulty process (p_i), which satisfies the definition of anti-Ω.

Lemma 5.5. *If $\mathcal{A}$ is fair and p_i is correct in $\mathcal{A}$, then $successor_i$ always terminates.*

Proof. Let p' be the current correct process. We need to prove that (i) line 5 in Figure 7 always terminates, and (ii) there is an s_i^k that satisfies line 6, terminating the loop.

(i) Since p_i is correct and $\mathcal{A}$ is fair, all steps s_i^k for $k = 1, 2, \ldots$ will eventually occur in $\mathcal{A}$. For each such k, there is a step s' by p', such that $s_i^k \overset{*}{\rightsquigarrow} s'$ (Lemma 4.2). Lemma 5.1 implies $s_i^k \in \mathrm{dom}_{s'}\, prec$.

(ii) By Lemma 4.2, for each step $s \in used$, there is a step $s_i^{k(s)}$ such that $s \overset{*}{\rightsquigarrow} s_i^{k(s)}$. The latest such step, s_i^k with $k = \max_{s \in used} k(s)$, satisfies $s \overset{*}{\rightsquigarrow} s_i^k$ for all $s \in used$, which implies $used \subseteq \mathrm{dom}_{s_i^k}\, prec_{\mathcal{A}}$ (Lemma 5.1). Then, line 3 in Figure 6 ensures that $used \subseteq prec_{\mathcal{A}}(s_i^k)$. □

For the rest of this proof, we can therefore assume that lines 5–7 in Figure 8 always terminate.

Lemma 5.6. *There is an infinite run $\mathcal{B}$ which does not decide.*

Proof. Consider the tree of all runs $\mathcal{B} = B_1, \ldots, B_k$ that do not decide. Each node in the tree correspond to some finite run $B_1, \ldots, B_k$, with children $B_1, \ldots, B_{k+1}$, one per each possible value of B_{k+1}. To obtain a contradiction, assume that this tree is finite; otherwise König's infinite-path lemma implies the assertion. This implies that there is a finite upper bound on the number of steps in any non-deciding run $\mathcal{B}$.

Since the number of non-deciding runs $B_1 \ldots B_k$ is finite, the algorithm in Figure 8 will terminate in a finite number of steps (we assumed that lines 5–7 always terminate), using only finite parts of $prec_{\mathcal{A}}$ and $det_{\mathcal{A}}$. Therefore, we can construct an algorithm $\mathsf{Algorithm}_{\text{async}}$ that uses these finite parts of $prec_{\mathcal{A}}$ and $det_{\mathcal{A}}$ to simulate $\mathsf{Algorithm}_{\Delta}$ in any asynchronous run (Figure 7).

Theorem 5.3 states that this run, with a simulated Δ, is indistinguishable from some, possibly unfair, run with a real Δ. This means that no safety property of Problem will be violated in the simulation. We have also shown that there is a bound on the number of steps required to decide. As a result, $\mathsf{Algorithm}_{\text{async}}$ is a purely asynchronous algorithm for Problem. This contradicts the non-wait-free-implementability of Problem, and proves the assertion. □

Lemma 5.7. *Each infinite non-deciding run $\mathcal{B}$ satisfies $inf(\mathcal{B}) \subset correct(\mathcal{A})$.*

Proof. To simulate all steps of any $p_i \in inf(\mathcal{B})$ without getting stuck in lines 4–6 in Figure 7, we need $prec_{\mathcal{A}}(s_i^k)$ for infinitely many steps $s_i^k \in actions(\mathcal{A})$. This implies $p_i \in correct(\mathcal{A})$, which means $inf(\mathcal{B}) \subseteq correct(\mathcal{A})$.

By Theorem 5.3, the system cannot distinguish run $\mathcal{B}$, with a simulated Δ, from some (possibly unfair) run $\mathcal{C}$, with the real Δ. To show that $inf(\mathcal{B}) \subset correct(\mathcal{A})$, we have to rule out the only remaining possibility $inf(\mathcal{B}) = correct(\mathcal{A})$. In this case, however, $correct(\mathcal{C}) = correct(\mathcal{A}) = inf(\mathcal{B}) = inf(\mathcal{C})$, so $\mathcal{C}$ is fair. As a result, $\mathsf{Algorithm}_{\Delta}$ must decide in $\mathcal{C}$, so also in $\mathcal{B}$, contradicting the assumption. □

Theorem 5.8. *The algorithm in Figure 8 implements anti-Ω.*

Proof. Consider the recursion tree of $explore$, in which each node is $explore(S_k, P_k)$, where k is the depth of that node. It contains each possible run $\mathcal{B}$ at least once, for example, by setting $P_k = P$ for all k. By Lemma 5.6, the tree contains an infinite path corresponding to some non-deciding run $\mathcal{B}$.

Let $\mathcal{B} = B_1, B_2, \ldots$ be such a run, the one encountered first by the algorithm in Figure 8. The ordering "$\subseteq$" in line 3 implies that $P_1, P_2, \ldots$ satisfies $P_k = \bigcup_{k' \geq k} B_{k'}$. The sequence $P_1 \supseteq P_2 \supseteq \cdots$ stabilizes at some $P_k = \bigcup_{k' \geq k} B_{k'} = \mathit{inf}(\mathcal{B})$. Since $\mathcal{B}$ does not decide, $explore(S_k, P_k)$ will not terminate, and will execute line 6 in Figure 8 only for $p_i \in P_{k' \geq k} = P_k = \mathit{inf}(\mathcal{B})$. As a result, only $p_i \in \mathit{inf}(\mathcal{B})$ will be output by the anti-Ω emulated in line 3 in Figure 7. By Lemma 5.7, $\mathit{inf}(\mathcal{B}) \subset \mathit{correct}(\mathcal{A})$, so there is a process $p' \in \mathit{correct}(\mathcal{A}) \setminus \mathit{inf}(\mathcal{B}) \subseteq \mathit{correct}(\mathcal{A})$ that will eventually never be output. □

6 Discussion and open questions

This section comments on the proof from Section 5, and suggests some open questions and directions of future work.

Comments on the proof

First, note that the proof is not specific to set agreement, but works with (almost) any non-wait-free-implementable abstraction Problem. This provides a confirmation that anti-Ω is the weakest non-implementable eventual failure detector [26]. Equivalently, any failure detector that implements Problem also implements set agreement. This does not mean, however, that set agreement is the weakest non-implementable abstraction [12].

The famous result that Ω is required for Consensus [4] uses the decision of the Consensus algorithm to construct a bivalent run. On the other hand, the proof presented in this paper does not use the decision value at all, only the fact that set agreement is not implementable [2, 18, 24]. I consider this a benefit, because we do not need to reprove non-implementablitity of set agreement, which would arguably be the most difficult part. In contrast, large parts of [4] are devoted to essentially reproving a version of FLP [9].

As a result, the proof presented in this paper is fairly generic, in the sense that only Section 5.3 depends on our assumptions about Problem and the anti-Ω detector being emulated. One could envision that, by changing the algorithm in Section 5.3, we could prove other minimal failure detector results. One way to do it would be to restrict the set of runs $\mathcal{B}$ to consider. For example, in Consensus, there is a non-deciding run $\mathcal{B}$ containing just two processes [9]. It is possible that this technique will allow us to prove, for example, that k-vector-Ω is the weakest failure detector to implement k-set agreement (Section 3.3).

Comments on failure detectors

The simulation in Section 5.2 implies that any detector can be simulated in a way that preserves all its safety properties. As a result, the weakest detector to implement any problem is an *eventual* failure detector, which offers only liveness properties. This is significant, because eventual failure detectors are fairly well understood in the sense that there is a mechanical procedure for testing their relative strength [26] (assuming a finite number of possible detector outputs).

This paper considered the shared memory model. Since the message-passing model does not have registers, it is plausible that the weakest failure detector necessary for implementing set agreement in this model is anti-Ω + Σ [7]. However, as anti-Ω is the weakest eventual failure detector, anti-Ω + Σ is equivalent to Σ alone. In other words, if this conjecture is true, both the register and set agreement would require the same failure detector in the message passing model.

Finally, note that anti-Ω is the weakest eventual failure detector in both shared-memory and message-passing models, because the relative strength of eventual failure detectors is largely model-independent [26].

Limitations of this work and open questions

The approach presented in this paper has two limitations. First, the implementation of vector-Ω using anti-Ω given in Figure 1 is not *anonymous* [12], that is, the given algorithm is not symmetric with respect to process identifiers. The culprit is line 9, "ties broken deterministically," which requires a pre-agreed order on processes. The question remains whether anti-Ω can implement set agreement in an anonymous way? Failing this, can it implement weaker abstractions such as renaming or weak symmetry breaking [12]?

The other limitation is the assumption (iii) made in the proof in Section 5 that "the termination condition of Problem is a function of the state of the system". Can we remove this restriction from the proof? Is this condition really limiting? We have seen that it is not for set agreement, but can, for example, renaming or weak symmetry breaking be transformed in a similar way?

7 Related work

The study of failure detectors was initiated by Chandra et al. [4], who introduced the concept and several detectors, including P, $\Diamond$P, $\Diamond$S, and Ω. They also showed that, in a system with a majority of correct processes, Ω is both sufficient and required to implement Consensus. Their proof technique, based on simulation, was used for analogous proofs for other abstractions [16], and inspired the approach presented in this paper.

Failure detectors have been extensively investigated in a number of other publications, for example [4, 8, 14, 16, 20, 22].

The set agreement problem was introduced by Chaudhuri [5]. The efforts to show the impossibility of a wait-free solution led to discovering deep connections between wait-free implementability and combinatorial topology, eventually succeeding in proving non-implementability of set agreement [2, 18, 24]. Achieving this result was challenging because, set agreement is one of the weakest non-wait-free-implementable problems.

Establishing the non-implementability started the quest for the weakest detector for set agreement. Previous attempts led to the Ω_k detector [23], the Υ failure detector [14], the $\Pi\Omega_k$ family [6], and anti-Ω [26]. Other recent work related to set agreement can be found in [10, 12, 13, 20, 21, 25]

8 Conclusion

The anti-Ω detector [26] outputs process ids and ensures that some correct process id is eventually never output. This paper showed that anti-Ω is both sufficient and necessary to implement set agreement in a shared-memory system equipped with registers.

The sufficiency is established by presenting an algorithm that uses anti-Ω to implement set agreement. It first implements an equivalent detector vector-Ω, a vector of $n-1$ detectors, at least one of which behaves as Ω. Vector-Ω is then used to implement set agreement. More generally, k-vector-Ω can implement k-set agreement.

The necessity is proved by showing that anti-Ω information can be extracted from any detector that can implement set agreement. The method collects samples of the detector outputs in the current run, and then uses them to simulate all possible runs of a implementing algorithm in order to find a non-deciding one. At least one correct process eventually does not participate in that run, which is sufficient to emulate anti-Ω. Interestingly, this method does not use the decision value of the set agreement algorithm, only the fact it does not terminate in some asynchronous run.

References

[1] Hagit Attiya. A direct lower bound for k-set consensus. In *PODC*, page 314, 1998. URL `http://doi.acm.org/10.1145/277697.277770`.

[2] E. Borowsky and E. Gafni. Generalized FLP impossibility result for t-resilient asynchronous computations. In Alok Aggarwal, editor, *Proceedings of the 25th Annual ACM Symposium on the Theory of Computing*, pages 91–100, San Diego, CA, USA, May 1993. ACM Press. ISBN 0-89791-591-7.

[3] Elizabeth Borowsky and Eli Gafni. Immediate atomic snapshots and fast renaming (extended abstract). In *PODC*, pages 41–51, 1993.

[4] Tushar Deepak Chandra, Vassos Hadzilacos, and Sam Toueg. The weakest failure detector for solving Consensus. *Journal of the ACM*, 43(4):685–722, 1996.

[5] Chaudhuri. More choices allow more faults: Set Consensus problems in totally asynchronous systems. *INFCTRL: Information and Computation*, 105, 1993.

[6] Wei Chen, Yu Chen, and Jialin Zhang. On failure detectors weaker than ever. Technical Report MSR-TR-2007-50, Microsoft Research, may 2007.

[7] Carole Delporte-Gallet, Hugues Fauconnier, and Rachid Guerraoui. Shared memory vs message passing. Technical report, LPD, December 2003.

[8] Carole Delporte-Gallet, Hugues Fauconnier, Rachid Guerraoui, Vassos Hadzilacos, Petr Kouznetsov, and Sam Toueg. The weakest failure detectors to solve certain fundamental problems in distributed computing. In *Proceedings of the 23rd Annual ACM Symposium on Principles of Distributed Computing*, pages 338–346. ACM Press, 2004.

[9] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed Consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, April 1985.

[10] Eli Gafni. Read-write reductions. In Soma Chaudhuri, Samir R. Das, Himadri S. Paul, and Srikanta Tirthapura, editors, *ICDCN*, volume 4308 of *Lecture Notes in Computer Science*, pages 349–354. Springer, 2006. ISBN 3-540-68139-6.

[11] Eli Gafni and Leslie Lamport. Disk Paxos. In *International Symposium on Distributed Computing*, pages 330–344, 2000.

[12] Eli Gafni, Sergio Rajsbaum, and Maurice Herlihy. Subconsensus tasks: Renaming is weaker than set agreement. In Shlomi Dolev, editor, *DISC*, volume 4167 of *Lecture Notes in Computer Science*, pages 329–338. Springer, 2006. ISBN 3-540-44624-9.

[13] Eli Gafni, Michel Raynal, and Corentin Travers. Test&set, adaptive renaming and set agreement: a guided visit to asynchronous computability. Technical report, IRISA, France, 2007.

[14] R. Guerraoui, M. Herlihy, P. Kouznetsov, N. Lynch, and C. Newport. On the weakest failure detector ever. Technical Report 1, Max Planck Institute for Software Systems, 2007.

[15] Rachid Guerraoui. Indulgent algorithms. In *Proceedings of the 19th Annual ACM Symposium on Principles of Distributed Computing*, pages 289–298, NY, July 2000. ACM Press.

[16] Rachid Guerraoui and Petr Kouznetsov. Finally the weakest failure detector for Non-Blocking Atomic Commit. Technical Report LPD-2003-005, EPFL, Lausanne, Switzerland, December 2003.

[17] Rachid Guerraoui and Michel Raynal. The Alpha of indulgent Consensus. *Comput. J*, 50(1):53–67, 2007.

[18] Herlihy and Shavit. The topological structure of asynchronous computability. *JACM: Journal of the ACM*, 46, 1999.

[19] Wai-Kau Lo and Vassos Hadzilacos. Using failure detectors to solve consensus in asynchronous shared-memory systems. In *Proceedings of the 8th International Workshop on Distributed Algorithms (WDAG)*, volume 857 of *LNCS*, pages 280–295, Terschelling, The Netherlands, 29 September–1 October 1994. Springer.

[20] Achour Mostefaoui, Sergio Rajsbaum, Michel Raynal, and Corentin Travers. Irreducibility and additivity of set agreement-oriented failure detector classes. In *PODC '06: Proceedings of the twenty-fifth annual ACM symposium on Principles of distributed computing*, pages 153–162, New York, NY, USA, 2006. ACM Press. ISBN 1-59593-384-0. doi: http://doi.acm.org/10.1145/1146381.1146406.

[21] Achour Mostéfaoui, Michel Raynal, and Corentin Travers. Exploring Gafni's reduction land: From Ω_k to wait-free adaptive $(2p - \lfloor p/k \rfloor)$-renaming via k-set agreement. In *Proceedings of the 20th International Symposium on Distributed Computing (DISC)*, volume 4167 of *LNCS*, pages 1–15, Stockholm, Sweden, September 2006. Springer.

[22] Michel Raynal. A short introduction to failure detectors for asynchronous distributed systems. *ACM SIGACT News*, 35(1):53–70, 2005.

[23] Michel Raynal and Corentin Travers. In search of the holy grail: Looking for the weakest failure detector for wait-free set agreement. In *Proceedings of the 10th International Conference on Principles of Distributed Systems (OPODIS)*, volume 4305 of *LNCS*, pages 3–19, Bordeaux, France, December 2006. Springer.

[24] Saks and Zaharoglou. Wait-free k-set agreement is impossible: The topology of public knowledge. *SICOMP: SIAM Journal on Computing*, 29, 2000.

[25] Corentin Travers Sergio Rajsbaum, Michel Raynal. Failure detectors as schedulers (an algorithmically-reasoned characterization). Technical report, IRISA, France, 2007.

[26] Piotr Zieliński. Automatic classification of eventual failure detectors. In *Proceedings of the 21st International Symposium on Distributed Computing (DISC)*, Lemesos, Cyprus, September 2007. (to appear).

A Proofs

Theorem 3.1. *The algorithm in Figure 1 implements vector-Ω.*

Lemma 4.1. *Any runs $\mathcal{A}$, with the same $actions(\mathcal{A})$, relation $\overset{*}{\rightsquigarrow}_{\mathcal{A}}$, and mapping $external_{\mathcal{A}}$ are indistinguishable.*

Proof. For any action a and each state variable $var \in view(a)$, let $in(a).var$ and $out(a).var$ denote the value of var immediately before and after action a. Two runs are indistinguishable if they have the same $in(a).var$ and $out(a).var$ for all actions a and variables $var \in view(a)$. I will show that the relation $\overset{*}{\rightsquigarrow}_{\mathcal{A}}$ and function $external_{\mathcal{A}}$ are sufficient to determine these uniquely.

For each state variable var consider the set of all actions that access var:

$$V = \{\, a \mid var \in view(a) \,\}$$

The time ordering of actions in V can be determined from $\overset{*}{\rightsquigarrow}$ because for any $a, a' \in V$:

$$view(a') \cap view(a) \supseteq \{var\} \supset \emptyset \quad \Longrightarrow \quad \big(time(a') < time(a) \iff a' \rightsquigarrow a\big).$$

The main proof uses structural induction on $\rightsquigarrow$. Consider any action a and $var \in view(a)$. The value var before a is the same as after the latest action $a' \in V$ that precedes a. In other words, $in(a).var = out(a').var$, and we have already determined $out(a').var$ by induction, because $a' \overset{*}{\rightsquigarrow} a$.

Using this procedure, we can determine $in(a).var$ for all $var \in view(a)$, which together with $external(a)$ is sufficient to determine $out(a).var$ for all $var \in view(a)$, thereby completing the inductive step. □

Lemma 4.2. *In a fair run, for any step s_j and any correct p_i, there is a step s_i such that $s_j \overset{*}{\rightsquigarrow} s_i$.*

Proof. Step s_j enables $update(\{p_j\})$. Memory fairness demands that this action cannot be enabled forever, so eventually $u_j = update(B)$ with $p_j \in B$ will be executed. Then, $step(p_i)$ cannot be enabled forever, so eventually $token_i = \mathsf{memory}$, enabling $update(\{p_i\})$. Therefore, some $u_i = update(B')$ with $p_i \in B'$ will eventually take place, resulting in $token_i = \mathsf{process}$. As a consequence, $s_i = step(p_i)$ will eventually occur. The assertion follows from $s_j \overset{*}{\rightsquigarrow} u_j \overset{0*}{\rightsquigarrow} u_i \overset{*}{\rightsquigarrow} s_i$. □

Lemma 4.3. *The block sequence $B_1, B_2, \ldots$ determines $\overset{*}{\rightsquigarrow}$.*

Proof. I will show that $B_1, B_2, \ldots$ determines $\rightsquigarrow$, and as a consequence $\overset{*}{\rightsquigarrow}$. Let $u^k = update(B_k)$ be the k-th update action, and u_i^k be the k-th action $update(B)$ with $p_i \in B$. To determine $\rightsquigarrow$, we need to show that each of the three groups of conflicting actions from Section 4.3 can be uniquely time-ordered given the information in $B_1, B_2, \ldots$.

(i) For any two updates u^k and $u^{k'}$, we have $time(u^k) < time(u^{k'}) \iff k < k'$.

(ii) We have $time(s_i^k) < time(u_i^{k+1}) < time(s_i^{k+1})$ because actions $step(p)$ and $update(B)$ with $p \in B$ alternate. Therefore, $time(s_i^k) < time(u_i^{k'}) \iff k < k'$, and $time(u_i^k) < time(s_i^{k'}) \iff k \leq k'$.

(iii) For any two steps s_i^k and $s_i^{k'}$, we have $time(s_i^k) < time(s_i^{k'}) \iff k < k'$. □

Lemma 5.1. *If X is a* growing mapping*, then* $\mathrm{dom}_{s_i} X_j = \{ s_j \mid s_j \overset{*}{\rightsquigarrow} s_i \}$.

Proof. First, I will show that any action causally succeeding a step action by some process, causally succeeds (or equals) the next update action involving that process. Formally, s_i^k be the k-th step by p_i, and u_i^k be the k-th update involving p_i. Note that $u_i^{k-1} \overset{*}{\rightsquigarrow} s_i^k \overset{*}{\rightsquigarrow} u_i^k$. We need to show that, for any action a,

$$s_i^k \overset{*}{\rightsquigarrow} a \iff u_i^k \overset{0*}{\rightsquigarrow} a, \quad \text{and} \quad a \overset{*}{\rightsquigarrow} s_i^k \iff a \overset{0*}{\rightsquigarrow} u_i^{k-1}. \tag{3}$$

For the purpose of this proof, $u_i^0 = u^0$ is the artificial $update(\{p_1, \ldots, p_n\})$ action that precedes all other actions, and behaves as if it wrote the initial values to $reg_1, \ldots, reg_n$ and took the snapshots $[reg_1, \ldots, reg_n]$ for all processes.

To show (3a), first observe that " $\Longleftarrow$ " follows from $s_i^k \overset{*}{\rightsquigarrow} u_i^k \overset{0*}{\rightsquigarrow} a$. For " $\Longrightarrow$ ", let $s_i^k \rightsquigarrow a' \overset{0*}{\rightsquigarrow} a$. We have two cases (i) $a' = s_i^{k'}$ for some $k' > k$, or (ii) $a' = u_i^{k'}$ with $k' \geq k$. In both cases $u_i^k \overset{0*}{\rightsquigarrow} a' \overset{0*}{\rightsquigarrow} a$. Equivalence (3b) can proved in the same way.

To avoid clutter, let us adopt the following shortcuts

$$s_j \overset{\text{def}}{=} s_j^{k_j}, \quad u_j \overset{\text{def}}{=} u_j^{k_j}, \quad s_i \overset{\text{def}}{=} s_i^{k_i} \quad u_i \overset{\text{def}}{=} u_i^{k_i - 1}.$$

Equivalences (3) imply

$$s_j \overset{*}{\rightsquigarrow} s_i \quad \overset{(3a)}{\iff} \quad u_j \overset{0*}{\rightsquigarrow} s_i \quad \overset{u_j \neq s_i}{\iff} \quad u_j \overset{*}{\rightsquigarrow} s_i \quad \overset{(3b)}{\iff} \quad u_j \overset{0*}{\rightsquigarrow} u_i. \tag{4}$$

Thus, two steps are causally dependent iff one reads (u_i) after the other writes (u_j).

We need to prove that $\mathrm{dom}_{s_i} X_j = Y_j$, where

$$Y_j \overset{\text{def}}{=} \{ s_j \mid s_j \overset{*}{\rightsquigarrow} s_i \} \overset{(4)}{=} \{ s_j \mid u_j \overset{0*}{\rightsquigarrow} u_i \}.$$

If Y is empty, then X_j available to s_i, as read by u_i, is the initial $\emptyset$, which implies $\mathrm{dom}_{s_i} X_j = \emptyset = Y_j$. Otherwise, X_j available to s_i, as read by u_i, was written by u_j corresponding to the latest step of p_j in Y_j, say s_j. By induction,

$$\mathrm{dom}_{s_i} X_j = \mathrm{dom}_{s_j} X_j \cup \{s_j\} \overset{\text{ind}}{=} \{ s'_j \mid s'_j \overset{*}{\rightsquigarrow} s_j \} \cup \{s_j\} = \{ s'_j \mid s'_j \overset{0*}{\rightsquigarrow} s_j \overset{*}{\rightsquigarrow} s_i \} \subseteq Y_j.$$

To show equality, assume there is an $s'_j \overset{*}{\rightsquigarrow} s_i$ but $s'_j \overset{0*}{\not\rightsquigarrow} s_j \overset{*}{\rightsquigarrow} s_i$. Since s'_j and s_j are both taken by p_j, we have $s'_j \overset{0*}{\not\rightsquigarrow} s_j \implies s_j \overset{*}{\rightsquigarrow} s'_j \overset{*}{\rightsquigarrow} s_i$, which means that s_j is not the latest step in Y_j, which contradicts the definition of s_j. □

Lemma 5.2. *Mapping map preserves causality:* $s_j \overset{*}{\rightsquigarrow}_{\mathcal{B}} s_i \implies map(s_j) \overset{*}{\rightsquigarrow}_{\mathcal{A}} map(s_i)$.

Theorem 5.3. *Let map be a function actions($\mathcal{B}$) $\rightarrow$ actions($\mathcal{A}$) satisfying Lemma 5.2. Then, the run $\mathcal{B}$ with Δ simulated by the algorithm in Figure 7 is indistinguishable from some run $\mathcal{C}$ with a real Δ, with inf($\mathcal{C}$) = inf($\mathcal{B}$) and correct($\mathcal{C}$) = correct($\mathcal{A}$). Run $\mathcal{C}$ may be unfair.*

Proof. I will construct such a run $\mathcal{C}$ with

$$\mathit{actions}(\mathcal{C}) \stackrel{\text{def}}{=} \mathit{actions}(\mathcal{B}), \quad \mathit{alive}_{\mathcal{C}} \stackrel{\text{def}}{=} \mathit{alive}_{\mathcal{A}}, \quad \mathit{history}_{\mathcal{C}} \stackrel{\text{def}}{=} \mathit{history}_{\mathcal{A}}.$$

Now, $\mathit{actions}(\mathcal{C}) = \mathit{actions}(\mathcal{B})$ implies $\mathit{inf}(\mathcal{C}) = \mathit{inf}(\mathcal{B})$, and $\mathit{alive}_{\mathcal{C}} = \mathit{alive}_{\mathcal{A}}$ implies $\mathit{correct}(C) = \mathit{correct}(A)$.

To prove the main indistinguishability claim, we need to specify the times of actions in $\mathit{actions}(\mathcal{C}) = \mathit{actions}(\mathcal{B})$. Let us start with step actions:

$$\mathit{time}_{\mathcal{C}}(s) \stackrel{\text{def}}{=} \mathit{time}_{\mathcal{A}}(\mathit{map}(s)).$$

Order let us order all updates $u^1, u^2, \ldots \in \mathit{actions}(\mathcal{C}) = \mathit{actions}(\mathcal{B})$ with respect to $\overset{*}{\rightsquigarrow}_{\mathcal{B}}$, and define

$$\begin{aligned} \mathit{time}_{\min}(u^k) &\stackrel{\text{def}}{=} \max\{\, \mathit{time}_{\mathcal{C}}(s\ ) \mid s \underset{\mathcal{B}}{\overset{*}{\rightsquigarrow}} u^k \,\} \\ \mathit{time}_{\max}(u^k) &\stackrel{\text{def}}{=} \min\{\, \mathit{time}_{\mathcal{C}}(s') \mid u^k \underset{\mathcal{B}}{\overset{*}{\rightsquigarrow}} s' \,\} \end{aligned} \tag{5}$$

Now, iteratively assign $\mathit{time}_{\mathcal{C}}$ to u^1, u^2, ..., so that

$$\mathit{time}_{\min}(u^k) < \mathit{time}_{\mathcal{C}}(u^k) < \mathit{time}_{\max}(u^k), \quad \mathit{time}_{\mathcal{C}}(u^{k-1}) < \mathit{time}_{\mathcal{C}}(u^k). \tag{6}$$

To show that such an assignment is possible, note that, for any $s \overset{*}{\rightsquigarrow}_{\mathcal{B}} u^k \overset{*}{\rightsquigarrow}_{\mathcal{B}} s'$, Lemma 5.2 implies $\mathit{map}(s) \overset{*}{\rightsquigarrow}_{\mathcal{A}} \mathit{map}(s')$. Therefore,

$$\mathit{time}_{\mathcal{C}}(s) = \mathit{time}_{\mathcal{A}}(\mathit{map}(s)) < \mathit{time}_{\mathcal{A}}(\mathit{map}(s')) = \mathit{time}_{\mathcal{C}}(s'),$$

hence,

$$\begin{aligned} \mathit{time}_{\min}(u^k) &< \mathit{time}_{\max}(u^k) && \text{by (5)}, \\ \mathit{time}_{\mathcal{C}}(u^{k-1}) < \mathit{time}_{\max}(u^{k-1}) &\le \mathit{time}_{\max}(u^k) && \text{by (5) and induction on } k, \end{aligned}$$

which makes assignments $\mathit{time}_{\mathcal{C}}(u^k)$ satisfying (6) possible.

To show that $\mathcal{B}$ and $\mathcal{C}$ are indistinguishable, we need $\overset{*}{\rightsquigarrow}_{\mathcal{B}} = \overset{*}{\rightsquigarrow}_{\mathcal{C}}$ and $\mathit{external}_{\mathcal{B}} = \mathit{external}_{\mathcal{C}}$ (Lemma 4.1). The former can be shown by proving $\rightsquigarrow_{\mathcal{B}} = \rightsquigarrow_{\mathcal{C}}$, that is, that all pairs of conflicting actions are executed in both $\mathcal{B}$ and $\mathcal{C}$ in the same order. In other words,

$$\mathit{time}_{\mathcal{B}}(a) < \mathit{time}_{\mathcal{B}}(a') \implies \mathit{time}_{\mathcal{C}}(a) < \mathit{time}_{\mathcal{C}}(a') \quad \text{for any conflicting } a \text{ and } a'. \tag{7}$$

Since $\mathit{time}_{\mathcal{B}}(a) \neq \mathit{time}_{\mathcal{B}}(a')$, the inverse implication follows automatically by exchanging a and a'. Using (5) and (6), we can show (7) by considering possible all pairs of conflicting actions (Section 4.3):

$$\begin{gathered} \mathit{time}_{\mathcal{B}}(u^k) < \mathit{time}_{\mathcal{B}}(u^{k'}) \implies k < k' \iff \mathit{time}_{\mathcal{C}}(u^k) < \mathit{time}_{\mathcal{C}}(u^{k'}), \\ \mathit{time}_{B}(u_i^k) < \mathit{time}_{B}(s_i) \implies u_i^k \rightsquigarrow_{\mathcal{B}} s_i \implies \mathit{time}_{\mathcal{C}}(u_i^k) < \mathit{time}_{\max}(u_i^k) \le \mathit{time}_{C}(s_i), \\ \mathit{time}_{B}(u_i^k) > \mathit{time}_{B}(s_i) \implies s_i \rightsquigarrow_{\mathcal{B}} u_i^k \implies \mathit{time}_{\mathcal{C}}(s_i) \le \mathit{time}_{\min}(u_i^k) < \mathit{time}_{C}(u_i^k), \\ \mathit{time}_{B}(s_i^k) < \mathit{time}_{B}(s_i^{k'}) \implies \mathit{time}_{B}(s_i^k) < \mathit{time}_{B}(u_i^k) < \mathit{time}_{B}(s_i^{k'}) \implies \\ \implies \mathit{time}_{C}(s_i^k) < \mathit{time}_{C}(u_i^k) < \mathit{time}_{C}(s_i^{k'}). \end{gathered}$$

We have shown that $\overset{*}{\rightsquigarrow}_{\mathcal{B}} = \overset{*}{\rightsquigarrow}_{\mathcal{C}}$. To conclude indistinguishability from Lemma 4.1, we need $\mathit{external}_{\mathcal{B}} = \mathit{external}_{\mathcal{C}}$. This forces us to define,

$$\mathit{external}_{\mathcal{C}}(u) \overset{\text{def}}{=} \bot = \mathit{external}_{\mathcal{B}}(u)$$
$$\mathit{external}_{\mathcal{C}}(s) \overset{\text{def}}{=} \mathit{det}_{\mathcal{C}}(s) \overset{\text{def}}{=} \mathit{det}_{\mathcal{A}}(\mathit{map}(s)) = \mathit{external}_{\mathcal{B}}(s).$$

We just need to show that $\det_{\mathcal{C}}$ is consistent with the failure history $\mathit{history}_C$, Eq. (2):

$$\mathit{det}_{\mathcal{C}}(s_i) = \mathit{det}_{\mathcal{A}}(\mathit{map}(s_i)) = \mathit{history}_{\mathcal{A}}(p_i, \mathit{time}_{\mathcal{A}}(\mathit{map}(s_i))) = \mathit{history}_{\mathcal{C}}(p_i, \mathit{time}_{\mathcal{C}}(s_i)),$$

and that $\mathit{history}_C$ is consistent with the failure pattern alive_C, Eq. (1):

$$p_i \in \mathit{alive}(\mathit{time}(\mathit{map}(s_i))) = \mathit{alive}(\mathit{time}(s_i)). \qquad \square$$

Lemma 5.4. *In fair runs, the executions of algorithm in Figure 8 are the same at all correct processes.*

Lemma 5.5. *If $\mathcal{A}$ is fair and p_i is correct in $\mathcal{A}$, then $\mathit{successor}_i$ always terminates.*

Lemma 5.6. *There is an infinite run $\mathcal{B}$ which does not decide.*

Lemma 5.7. *Each infinite non-deciding run $\mathcal{B}$ satisfies $\mathit{inf}(\mathcal{B}) \subset \mathit{correct}(\mathcal{A})$.*

Theorem 5.8. *The algorithm in Figure 8 implements anti-Ω.*