Technical Report

UCAM-CL-TR-693
ISSN 1476-2986

Number 693

UNIVERSITY OF CAMBRIDGE

Computer Laboratory

Automatic classification of eventual failure detectors

Piotr Zieliński

July 2007

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

http://www.cl.cam.ac.uk/

Technical reports published by the University of Cambridge
Computer Laboratory are freely available via the Internet:

http://www.cl.cam.ac.uk/techreports/

ISSN 1476-2986

Automatic classification of eventual failure detectors

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

Abstract

Eventual failure detectors, such as Ω or $\Diamond$P, can make arbitrarily many mistakes before they start providing correct information. This paper shows that any detector implementable in a purely asynchronous system can be implemented as a function of only the order of most-recently heard-from processes. The finiteness of this representation means that eventual failure detectors can be enumerated and their relative strengths tested automatically. The results for systems with two and three processes are presented.

Implementability can also be modelled as a game between Prover and Disprover. This approach not only speeds up automatic implementability testing, but also results in shorter and more intuitive proofs. I use this technique to identify the new weakest failure detector *anti*-Ω and prove its properties. Anti-Ω outputs process ids and, while not necessarily stabilizing, it ensures that some correct process is eventually never output.

1 Introduction

In purely asynchronous systems, messages between processes can take arbitrarily long to reach their destinations. It is therefore impossible to distinguish a faulty process from a very slow one [8], which causes many practical agreement problems, such as consensus or atomic commit, to be unsolvable [5].

One method of dealing with this impossibility is by equipping the system with failure detectors [3, 11]. A failure detector is an abstract distributed object that processes can query to get information about failures in the system. Different kinds of failure detectors provide different sorts of information, with different reliability guarantees. For example, the eventually perfect detector ($\Diamond$P) returns a set of "suspected" processes, and guarantees that *eventually* it will equal the set of faulty processes. The eventual leader detector (Ω) returns a single process, and guarantees that eventually it will keep returning the same correct process.

Both $\Diamond$P and Ω are reliable only *eventually*. They can make mistakes for an arbitrarily long but finite period of time, which is unknown to the application. Such detectors are attractive because algorithms using them are *indulgent*; they never fully "trust" the

detector, therefore they never violate safety, even if the detector violates its specification [6]. This paper focuses exclusively on such detectors.

Different distributed tasks require different failure detectors. A detector is *implementable* if there is an algorithm that implements it, in a given model. A considerable amount of research has focused on determining the implementability relationships both between problems and failure detectors, and between failure detectors themselves (eg. [3, 4, 7, 9, 11]). For example, $\Diamond$P can implement Ω, by outputting the non-suspected process with the smallest id. As a result, every problem solvable with Ω is solvable with $\Diamond$P, but not vice versa [3].

Despite a number failure detectors identified in the literature, no comprehensive exploration of their design space has yet been attempted. As a result, identifying new failure detectors is difficult, and their properties must typically proved from scratch. This paper presents a method that greatly simplifies these tasks: an efficient and fully mechanical procedure for determining the implementability relationship between eventual failure detectors in a system with a given number of processes. The overall strategy to arrive at this result consists of the following steps:

- Section 2 shows that, under reasonable assumptions, all eventual failure detectors can be completely specified by the list of allowed sets of symbols output infinitely often. For Ω, this list consists of singleton sets, each containing a single correct process.
- Section 3 shows that, assuming immediate reliable broadcast, any implementable failure detector can be implemented as a function operating solely on the sequence of past process steps.
- Section 4 shows that only the order of last occurrences of processes in the above sequence matters. With finitely many possible such orderings, this opens the door to automatic enumeration of failure detectors.
- Section 5 shows that any failure detector implementable in the immediate reliable broadcast model remains so in the purely asynchronous model. In particular, all results from Section 3 and 4 still apply.
- Section 6 generalizes the above results to automatically comparing relative strengths of different failure detectors.
- Section 7 introduces a more intuitive, game-theoretic interpretation of the results from previous sections. It also identifies the weakest non-implementable failure detector *anti*-Ω, and proves its properties.
- Section 8 presents the results of automatic enumeration of failure detectors and their relative implementability in a three-process system. Game-solving techniques are used to speed up the search.

2 System model and failure detector specifications

The system consists of a fixed set $P = \{1, 2, \ldots, n\}$ of processes, which communicate using asynchronous reliable channels: messages between correct processes eventually get

delivered, but there is no bound on message transmission delay.

Processes can fail by crashing. In any run, the *failure pattern* is a function $alive(t)$, which returns the set of non-crashed processes at any given time $t \in \mathbb{N}$. Crashed processes do not recover, therefore $alive(t) \supseteq alive(t+1)$. Processes that never crash ($C = \bigcap_t alive(t) \neq \emptyset$) are called *correct*, the others are *faulty*. Runs are *fair*: correct processes perform infinitely many steps.

The system may be equipped with a failure detector. When queried, the detector returns a *symbol*, for example, a process id (Ω) or a set of processes ($\Diamond$P). The *detector history* is a function $hist(q, t)$, which gives the symbol returned by the detector at process q at time t. A *failure detector specification* $\mathcal{H}$ is a function that maps each pattern failure *alive* into a set of allowed functions *hist*. For example, for Ω, we have

$$\mathcal{H}_\Omega(alive) = \{\, hist \mid \exists_{t\in\mathbb{N}}\, \exists_{p\in(\bigcap_t alive(t))}\, \forall_{p'\in P}\forall_{t'>t}\, hist(p', t') = p \,\}. \tag{1}$$

2.1 Failure detector assumptions

The standard failure detector specification method [3] described above is very general, but this results in complicated specifications (1). This section simplifies this specification method by making the following assumptions:

1. The detector can behave arbitrarily for any finite amount of time.

2. The set of possible symbols output by the detector is finite.

3. The detector cannot distinguish otherwise indistinguishable runs.

For example, a detector that "eventually keeps outputting the process that crashed first" violates Assumption 3: even knowing the entire infinite sequence of system states in a given run is not enough to determine any upper bound on processes' crash times. This is because we cannot distinguish between a process that crashed and one that simply does not take steps.

On the other hand, the set of correct processes, provided by $\Diamond$P, is deducible from such an infinite sequence of states. As other detectors, $\Diamond$P is useful because it provides this information about the entire infinite run at a finite time.

This paper additionally assumes that the detector is *querier-independent*, that is, function *hist* depends only on time t, not on the querying process p. I do not list this with other assumptions, because detectors not satisfying this assumption can be emulated by ones that do (Section 3.1, (6)).

2.2 Failure detector specification

Assumptions 1–3 allow us to considerably reduce both the space of considered failure detectors as well as the complexity of their descriptions. First, Theorem 3 shows that $\mathcal{H}$ depends only on the set $C = \bigcap_t alive(t)$ of correct processes, not on the exact form of *alive*. This simplifies (1) to

$$\mathcal{H}_\Omega(C) = \{\, hist \mid \exists_{t\in\mathbb{N}}\, \exists_{p\in C}\, \forall_{t'>t}\, hist(t') = p \,\}. \tag{2}$$

	Ω	$\Diamond$S	$\Diamond$P	$\Diamond$?P
infset(1)	1	▖	▖	▌
infset(2)	2	▘	▘	▌
infset(12)	1,2	▖□, ▘□	□	□

▖: only process 1 is correct
▘: only process 2 is correct
▌: either 1 or 2 faulty
□: no failures

Figure 1: Specifications *infset*(C) for various failure detectors in a system with two processes 1 and 2 (left), and the interpretation of the output symbols (right).

Theorem 4 shows that whether "$hist \in \mathcal{H}(C)$" depends only on the set of values that $hist(t)$ takes infinitely often, not on the exact form of *hist*. Therefore, we can specify a failure detector as the set *infset*(C) of allowed sets of symbols output infinitely often. The description (2) simplifies to

$$infset_\Omega(C) = \{\, \{p\} \mid p \in C \,\}. \tag{3}$$

In general,

$$infset(C) \stackrel{\text{def}}{=} \{\, inf(s_1 \ldots) \mid s_k = hist(k),\ hist \in \mathcal{H}(C) \,\}. \tag{4}$$

where $inf(s_1 \ldots) = \bigcap_{k=1,2,\ldots} \{s_k, s_{k+1}, \ldots\}$ is the set of symbols s_i's that occur infinitely often in $s_1 \ldots$, for example, $inf(32413512212122 \ldots) = \{1, 2\}$. Since a failure detector can behave better than required, $S \subseteq T \in infset(C)$ implies $S \in infset(C)$ (Theorem 5). All free set variables in this paper, such as S, T, C in the previous sentence, are implicitly assumed to be non-empty.

Examples. Figure 1 shows the specifications of several known detectors, in a two-process system. Detectors Ω and $\Diamond$P have already been introduced. Anonymous $\Diamond$?P eventually detects whether all processes are correct (□) or not (▌), without revealing the identities of faulty processes. Detector $\Diamond$S is similar to $\Diamond$P: it also outputs a set of suspected processes, however, $\Diamond$S can forever suspect some, but not all, correct processes [3]. Figure 1 represents a set of suspected processes as a vertical bitmap (eg. ▘), with one entry per process; black entries mean "suspected", white entries "not suspected".

For each detector, Figure 1 on page 6 shows the value of *infset*(C) for $C = \{1\}$, $\{2\}$, $\{1, 2\}$. For brevity, sets $\{a, b, \ldots\}$ are abbreviated to $ab\ldots$, non-maximal elements of *infset*(C) removed (Theorem 5), and external braces omitted. For example,

$$infset_{\Diamond S}(\{1, 2\}) = \{\{▖\}, \{▘\}, \{□\}, \{▖, □\}, \{▘, □\}\} \quad \Longrightarrow \quad infset_{\Diamond S}(12) = ▖□, ▘□. \tag{5}$$

This de-cluttering convention is used throughout the paper.

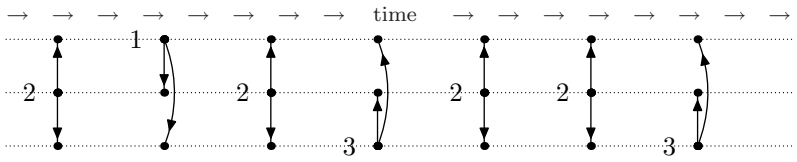
3 Implementability in the immediate broadcast model

Our goal is to determine whether a given failure detector, as specified by its *infset*, is implementable. Sections 3 and 4 will investigate this question in the *immediate broadcast model*. This model is significantly stronger than the purely asynchronous model, for example, its basic broadcast primitive of implements atomic broadcast, which in non-implementable in the asynchronous model [3, 5]. Surprisingly, however, as far as implementability of (eventual) failure detectors is concerned, these two models are equivalent (Section 5).

In the *immediate broadcast model*, all messages are transmitted instantaneously and reliably. Processes take steps in any fair order: correct processes take infinitely many steps, faulty ones finitely many steps. Processes never fail in the middle of a step.

3.1 Failure detector implementations

Immediate and reliable broadcast ensures that each process always knows the complete state of the system: the sequence $p_1 \dots p_k$ of processes that have taken steps until this moment. For example, the state at the end of



is $p_1 \dots p_7 = 2123223$. Assuming determinism, all other state information can be inferred from $p_1 \dots p_k$ (the initial state is fixed). Therefore, the complete state of any algorithm in this model depends only on $p_1 \dots p_k$. In particular, any failure detector implementation can be modelled as a function *output* from sequences of processes $p_1 \dots p_k$ to output symbols s_k.

A failure detector sensitive to the identity of the querying process has n functions: $output_1$, …, $output_n$, one per process. However, these can be transformed into a single, querier-independent function outputting a composite symbol:

$$output(p_1 \dots p_k) = [s_{k1} \dots s_{kn}], \quad \text{where } s_{ki} = output_i(p_1 \dots p_k). \tag{6}$$

The original detector output at process i is the s_{ki} in the composite symbol $[s_{k1} \dots s_{kn}]$. Thus, for any failure detector implementation $output_1$, …, $output_n$, there is a querier-independent detector implementation *output* that can emulate it. For this reason, this paper focuses on querier-independent detectors.

3.2 Failure detector specifications

From (4), an implementation *output* is consistent with a specification *infset* iff, for any infinite sequence $p_1 \dots$ of processes, we have:

$$inf(s_1 \dots) \in infset(C), \quad \text{where } s_k = output(p_1 \dots p_k) \text{ and } C = inf(p_1 \dots). \tag{7}$$

For example, consider a trivial failure detector:

$$infset_{\text{trivial}}(C) = \{\, X \mid X \subseteq C \,\} \quad \text{for all } C \subseteq P, \tag{8}$$

which eventually outputs only correct processes. It can be implemented by returning the most recent process to take a step, that is, $output(p_1 \dots p_k) = p_k$. Similarly, returning the least recent process, for example, $output(2123223) = 1$, will eventually keep outputting one stable *faulty* leader, if it exists:

$$infset_{\text{faulty}}(C) = \{\, \{p\} \mid p \notin C \,\} \quad \text{for all } C \subset P. \tag{9}$$

(Compare with (3).) By convention, the undefined case $C = P$ allows arbitrary behaviour.

For any set X, let $perms(X)$ be the set of all permutations of elements of X. Let $order(p_1 \dots p_k) \in perms(12 \dots n)$ be obtained from $p_1 \dots p_k$ by retaining only the last occurrence of each process (eg. $order(312233143433131) = 2431$)[1]. The implementations of failure detectors (8) and (9) can be succinctly written as

$$\begin{aligned} output_{\text{trivial}}(p_1 \dots p_k) &= \text{last element of } order(p_1 \dots p_k) \\ output_{\text{faulty}}(p_1 \dots p_k) &= \text{first element of } order(p_1 \dots p_k) \end{aligned} \tag{10}$$

Note that both implementations above ignore all information in $p_1 \dots p_k$, except for $order(p_1 \dots p_k)$. Theorem 1 shows that all implementable failure detectors can be implemented this way, with $s_k = output(p_1 \dots p_k) = map(order(p_1 \dots p_k))$ for some function map from $perms(12 \dots n)$ to output symbols. For example,

$$map_{\text{trivial}}(q_1 \dots q_n) = q_n, \qquad map_{\text{faulty}}(q_1 \dots q_n) = q_1.$$

With a fixed number n of processes, the number such functions map is finite, which enables us to automate implementability testing (Section 8).

In any run, as the sequence of steps $p_1 \dots p_k$ grows, $order(p_1 \dots p_k)$ keeps changing. Since faulty processes take finitely many steps, eventually the prefix of $order(p_1 \dots p_k)$ consisting of all faulty processes will stabilize, while the rest, consisting of correct processes, will keep changing. Therefore, the implementation map is consistent (7) with the specification $infset$ iff for any order $q_1 \dots q_k$ of faulty processes

$$\bigcup \{\, map(q_1 \dots q_k r_1 \dots r_{n-k}) \mid r_1 \dots r_{n-k} \in perms(C) \,\} \in infset(C), \tag{11}$$

where $C = P \setminus \{q_1 \dots q_k\}$.

For example, we can show that Ω is not implementable. To obtain contradiction, assume that it is. By Theorem 1, there is an implementation

$$output_\Omega(p_1 \dots p_k) = map_\Omega(order(p_1 \dots p_k)).$$

For any order $q_1 \dots q_n$, we must have $map_\Omega(q_1 \dots q_n) = q_n$ because q_n might be the only correct process (11). In other words, this implementation of Ω always outputs the last process to take a step. However, if more than one process is correct, the output may never stabilize, violating the properties of Ω.

4 Order map theorem

Section 3 used the fact that any implementable failure detector can be implemented using some function map acting solely on the order of recent process steps. This section proves this theorem. It is important because it restricts the originally infinite number of possible functions $output$ to those induced by one of the functions map, whose number is finite.

Theorem 1. *Any implementable failure detector has an implementation of the form* $output(p_1 \dots p_k) = map(order(p_1 \dots p_k))$ *for some function map.*

[1] To ensure that $order(p_1 \dots p_k)$ always contains all processes, even if some do not occur in $p_1 \dots p_k$, I implicitly prefix each $p_1 \dots p_k$ with $12 \dots n$.

```
1   function update(q_1 ... q_n) is
2     simulate q_n taking a step
3     set map(q_1 ... q_n) ← failure detector output in the simulation
4   function update(q_1 ... q_{k<n}) is
5     repeat
6       for each q ∉ q_1 ... q_k do
7         update(q_1 ... q_k q)
8     until (11) holds for q_1 ... q_k
9   task construct map is
10    update(ε), where ε is the empty sequence
```

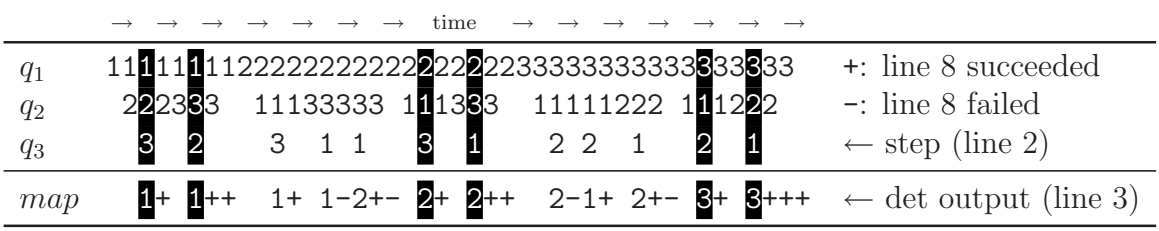
Figure 2: Generating a *map* for a given failure detector implementation using failure detector outputs in a specially constructed simulated run.

Proof. Figure 2 presents an algorithm that, for any implementable failure detector, constructs a *map* that implements it, that is, is consistent (11) with the detector's *infset*. It takes the algorithm implementing the failure detector, and collects its outputs in a simulated run. This run is constructed by function $update(q_1 \dots q_i)$, which updates *map* so that (11) holds for all $q_1 \dots q_k$ starting with $q_1 \dots q_i$. Therefore, $update(\varepsilon)$ in line 10 produces a *map* that satisfies (11) for all $q_1 \dots q_k$.

The implementation of $update(q_1 \dots q_i)$ covers two cases. For $q_1 \dots q_n$ consisting of all processes, *update* makes the last process step, queries the detector, and sets $map(q_1 \dots q_n)$ to its output (lines 1–3). It trivially satisfies (11), because no valid sequence of faulty processes can contain all processes.

For shorter $q_1 \dots q_k$, function *update* recursively ensures that (11) holds for all extensions of $q_1 \dots q_k$, and then tests whether (11) holds for $q_1 \dots q_k$ itself. If not, the process is repeated **until** success (lines 5–8). This cannot go on forever, because $update(q_1 \dots q_k)$ makes only processes in $C = P \setminus \{q_1 \dots q_k\}$ take steps. Therefore, any implementable detector will eventually start outputting symbols from some $S \in infset(C)$, passing the test in line 8. □

Example. Consider the faulty-leader detector (9) implemented by returning the process that took least steps (not the least recent one to step), favouring lower ids to break ties. This results in the following run of the algorithm in Figure 2 on page 9:



Function *update*() calls *update*(1), *update*(2), *update*(3). The recursion in *update*(1) eventually makes processes 2 and 3 step. In both cases, the detector outputs 1, which results in mappings $map(123) = map(132) = 1$, which pass the line 8 test in *update*(12), *update*(13), and then *update*(1).

```
1  maxstep[i] ← 0 for all processes i
2  when process i takes its k-th step do
3    reliably broadcast "process i, step k"
4  when reliably receive "process i, step k" do
5    maxstep[i] ← max {k, maxstep[i]}
6  when queried do
7    return the list of all processes i, ordered wrt increasing maxstep[i]
8      (ties broken deterministically)
```

Figure 3: An implementation of Order Oracle in an asynchronous system.

Function *update*(2) encounters more problems. It first calls *update*(21), which produces $map(213) = 1$, and then *update*(23). Function *update*(23) calls *update*(231), which produces $map(231) = 1$. Since $1 infset(1)$, line 8 in *update*(23) fails and *update*(231) is called again. It sets $map(231) = 2$, which passes the test in *update*(23), but (together with $map(213) = 1$) fails the test in *update*(2) because $12 infset(13)$. Calling *update*(231) and *update*(213) again results in $map(213) = map(231) = 2$, which passes the test in *update*(2).

Similarly, *update*(3) results in $map(312) = map(321) = 3$. The algorithm in Figure 2 on page 9 has therefore transformed the original least-often-stepping implementation of (9), into the least-recent-to-step implementation *map*, highlighted above.

5 Implementability in the asynchronous model

This section shows that any failure detector implementable in the immediate broadcast model (Sections 3 and 4) remains so in the purely asynchronous model. (The opposite implication is obvious.) This result implies, for example, that for any implementable failure detector, there is a querier-independent, implementable failure detector that can emulate it (Section 3.1).

Consider a failure detector implementable in the immediate broadcast model. Section 4 showed that there is a *map* consistent with it (11), which acts on the process order $q_1 \dots q_n$. This process order must satisfy (11): (i) faulty processes precede correct ones, and (ii) the order of faulty processes is fixed. Let *Order Oracle* be an abstraction that, when queried, outputs an order that eventually satisfies (i) and (ii). It is sufficient to show that Order Oracle is implementable in purely asynchronous settings.

As an example, consider a four-process system with only processes 3 and 4 correct. Order Oracle can keep switching between 1234 and 1243 or between 2134 and 2143 in the same run. However, outputting both 1234 and 2134 infinitely often in the same run would violate (ii), and 2314 would violate (i).

In the algorithm in Figure 3 on page 10, processes reliably broadcast a message whenever they take a step. Each process keeps track of steps taken by others by storing the highest-numbered step for each process in the vector *maxstep*. When the algorithm is asked for an order on processes, it returns them in the increasing order of *maxstep*.

This simple algorithm is similar to the heartbeat failure detector [1], with one important difference: it uses reliable broadcast [10] rather than ordinary broadcast. This ensures that not only *maxstep*s of correct processes keep increasing without limit (i), but also that eventually *maxstep*s corresponding to faulty processes will be the same at all correct processes (ii). Note that the agreement on the order of faulty processes is only "eventual" in the same sense as reliable broadcast makes correct processes agree on the set of broadcast messages. In particular, it does not contradict FLP [5].

Conclusion. By taking the results from Section 4 and this section together, we can conclude that a failure detector is implementable in the purely asynchronous system iff there is a *map* consistent (11) with its specification *infset*.

6 Comparing relative strengths of failure detectors

This section shows that the theory developed in previous sections allows us not only to mechanically test implementability, but also to compare relative strength of failure detectors. In other words, we can test whether a given failure detector (eg. ◇P) is implementable in the asynchronous system equipped with another detector (eg. Ω).

For any failure detector S, consider a *purely asynchronous* system consisting of real processes P and virtual processes R_S, one for each possible output of S (its *range*). For example, with a two-process ◇P, we have processes $P = \{1, 2\}$ and $R_{\Diamond\mathrm{P}} = \{\blacksquare, \blacksquare, \square\}$. In general, the set of processes is the *disjoint union* $[P, R_S]$, in which members of P and R_S keep separate identities[2], even if they have identical names (eg. $R_\Omega = P$).

The scheduler ensures that virtual processes behave according to the detector specification, that is, the set $[C, S] \subseteq [P, R_S]$ of correct processes satisfies $S \in \mathit{infset}(C)$. (A process is correct iff it takes infinitely many steps.) Given this assumption, real processes $p \in P$ can emulate the failure detector by always outputting the most-recently heard-from virtual process $s \in R_S$ (Theorem 6).

To check whether a failure detector S can implement another detector T, we need to test whether T is implementable in the system $[P, R_S]$. The specification of T in this system is

$$\mathit{infset}_T([C, S]) = \mathit{infset}_T(C) \qquad \text{for all } S \in \mathit{infset}_S(C). \tag{12}$$

By convention (9), the undefined cases $[C, S]$ with $S \notin \mathit{infset}_S(C)$ allow arbitrary behaviour: $\mathit{infset}_T([C, S]) = \{\, X \mid X \subseteq R_T \,\}$.

Example 0. Consider the eventually anonymously perfect detector ◇?P, which eventually consistently outputs ■ or □, depending on the actual state (Figure 1 on page 6). Let us first show that, in a two-process system, ◇P implements ◇?P. In a system equipped with ◇P, detector ◇?P is defined by (5) (12):

$$\mathit{infset}([1, \blacksquare]) = \blacksquare, \quad \mathit{infset}([2, \blacksquare]) = \blacksquare, \quad \mathit{infset}([12, \square]) = \square.$$

This specification can be easily implemented by looking just at the last output of ◇P: if it is □, output □, otherwise output ■. The *map* function, defined for all orders $s_1 \ldots s_5 \in \mathit{perms}(12\blacksquare\blacksquare\square)$, returns □ if □ is the last non-digit in $s_1 \ldots s_5$, and ■ otherwise.

[2]Formally, $[A_1 \ldots A_k] = \{\, (a, i) \mid a \in A_i \,\}$. Then, $[A, B] \subseteq [A', B'] \Leftrightarrow A \subseteq A' \wedge B \subseteq B'$.

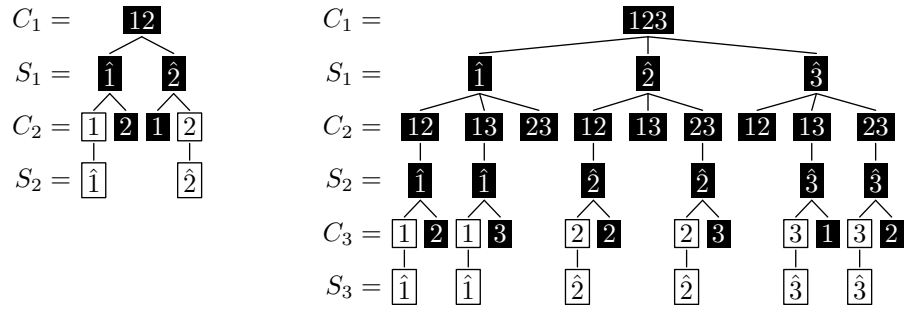


Figure 4: Game trees for Ω with two processes (left), and three process (right).

Example 1. Consider a two-process system equipped with $\Diamond$?P (Figure 1 on page 6). To show that $\Diamond$P is implementable in such a system, consider the requirements (12):

$$\mathit{infset}_{\Diamond\mathrm{P}}([1,\blacksquare]) = \blacksquare, \quad \mathit{infset}_{\Diamond\mathrm{P}}([2,\blacksquare]) = \blacksquare, \quad \mathit{infset}_{\Diamond\mathrm{P}}([12,\square]) = \square.$$

To implement $\Diamond$P, output $\square$ if $\Diamond$?P outputs $\square$. Otherwise output $\blacksquare$ or $\blacksquare$, depending whether the most recently heard-from process is 1 or 2. This strategy corresponds to the following *map*, which satisfies (11):

$$\square 21\blacksquare, \square 2\blacksquare 1, \square\blacksquare 21, 2\square\blacksquare 1, 2\square 1\blacksquare, 21\square\blacksquare \mapsto \blacksquare \qquad \square 12\blacksquare, \square 1\blacksquare 2, \square\blacksquare 12, 1\square\blacksquare 2, 1\square 2\blacksquare, 12\square\blacksquare \mapsto \blacksquare$$
$$\blacksquare 12\square, \blacksquare 1\square 2, \blacksquare 21\square, \blacksquare 2\square 1, \blacksquare\square 12, \blacksquare\square 21 \mapsto \square \qquad 12\blacksquare\square, 1\blacksquare 2\square, 1\blacksquare\square 2, 21\blacksquare\square, 2\blacksquare 1\square, 2\blacksquare\square 1 \mapsto \square$$

Example 2. To show that Ω cannot implement $\Diamond$?P, consider the requirements

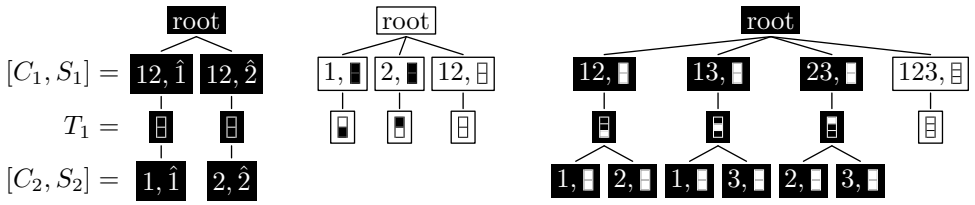
$$\mathit{infset}_{\Diamond?\mathrm{P}}([1,\hat{1}]) = \blacksquare, \quad \mathit{infset}_{\Diamond?\mathrm{P}}([2,\hat{2}]) = \blacksquare,$$
$$\mathit{infset}_{\Diamond?\mathrm{P}}([12,\hat{1}]) = \mathit{infset}_{\Diamond?\mathrm{P}}([12,\hat{2}]) = \square.$$

(I use $\hat{1},\hat{2}$ for Ω outputs to avoid name collisions with processes $1,2$.) First, $\mathit{infset}([12,\hat{2}]) = \square$ and (11) imply that $map(\hat{1}12\hat{2}) = \square$. However, $\mathit{infset}([2,\hat{2}]) = \blacksquare$ implies $map(\hat{1}12\hat{2}) = \blacksquare$, which contradicts $map(\hat{1}12\hat{2}) = \square$.

7 Game-theoretic interpretation of implementability

Two players, YES and NO, play the following game. In the k-th turn, NO chooses a set $C_k \subseteq P$, and YES chooses $S_k \in \mathit{infset}(C_k)$. The sets must satisfy $C_1 \supseteq C_2 \supseteq \cdots \neq \emptyset$, and $S_1 \supseteq S_2 \supseteq \cdots \neq \emptyset$. The first player unable to make a move loses. Theorem 7 shows that YES has a winning strategy iff the failure detector is implementable.

Figure 4 (left) shows the game tree for the two-process Ω. Each path C_1, S_1, ..., starting at the root, represents a sequence of moves. For example, "12, $\hat{1}$, 1, $\hat{1}$" is a victory for YES, and "12, $\hat{2}$, 1" for NO. White nodes are wins for YES, black ones for NO. The colour of a node can be easily computed using the minimax algorithm [12]: C_k (resp. S_k) nodes are black iff all (resp. some) of their children are black. Since $C_1 = 12$ is black, NO has a winning strategy, so the two-process Ω is not implementable. As Figure 4 on page 12 (right) suggests, similar reasoning works for general $n > 2$ (Theorem 9).



(a) Ω, ◇?P, 2 (b) ◇?P, ◇P, 2 (c) ◇?P, ◇P, 3

Figure 5: Game trees corresponding to implementing detector T in a system equipped with detector S in an n-process systems, for three different (S, T, n).

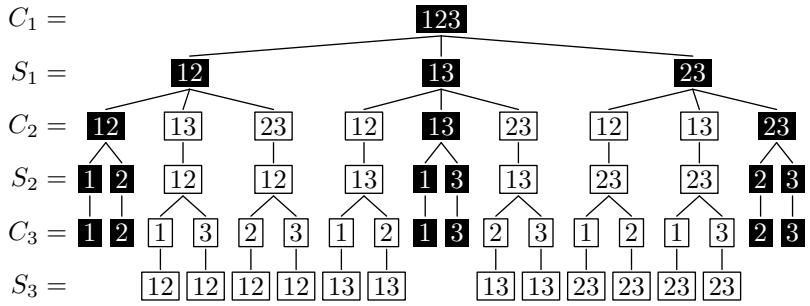


Figure 6: Game tree for the three-process anti-Ω.

7.1 Comparing relative detector strengths using game theory

With the modifications described in Section 6, the game-theory approach can also be used to check whether one failure detector S can implement another detector T. Since the system is now equipped with S, player NO chooses $[C_1, S_1] \supset [C_2, S_2] \supset \cdots \neq \emptyset$. YES chooses $T_1 \supseteq T_2 \supseteq \cdots \neq \emptyset$ with $T_k \in \mathit{infset}_T([C_k, S_k])$.

We can assume that $S_k \in \mathit{infset}_S(C_k)$ and $C_{k-1} \supset C_k$, because otherwise YES could always repeat its previous move, which cannot benefit NO (Lemma 8). With (12), this implies $T_k \in \mathit{infset}_T(C_k)$.

Figure 5 shows game trees corresponding to implementing detector T in a system equipped with detector S in an n-process systems, for three different (S, T, n). Case (a) shows that Ω cannot implement ◇?P in a two-process system. Detector ◇?P can implement ◇P with two processes (b), but not with three (c).

7.2 Anti-Ω: the weakest failure detector

The anti-Ω failure detector is specified as

$$\mathit{infset}_{\text{anti-}\Omega}(C) = \{\, S \mid C \not\subseteq S \subseteq P \,\}. \tag{13}$$

It outputs process ids, and ensures that some correct process id will eventually *never* be output. Note that the classic Ω ensures that such an id will eventually *always* be output.

Theorem 10 shows that anti-Ω is not implementable: NO can win by playing $C_1 = P$ and then always copying YES's last move $C_{k+1} = S_k$. This strategy corresponds to the black nodes in the three-process anti-Ω game-tree shown in Figure 6 on page 13. In this tree, each S_k-node has exactly one black child; the minimax rule therefore implies that

whitening any black node would make the game winnable by YES. In a sense, anti-Ω is therefore a "locally weakest detector".

Theorem 12 uses the method from Section 7.1 to prove a stronger result: anti-Ω is the (globally) weakest non-implementable eventual failure detector in the sense that it can be implemented by any non-implementable detector. In particular, anti-Ω is strictly weaker than Υ, the weakest *stable* detector [9]:

$$infset_{\Upsilon}(C) = \{\, \{T\} \mid C \neq T \subseteq P \,\}. \tag{14}$$

(A detector is stable iff it eventually outputs the same symbol, that is, all $infset(C)$'s consist of singleton sets only.) As a by-product, this shows that some failure detectors, such as anti-Ω, have no stable equivalents.

Anti-Ω is also the weakest detector that solves set agreement [13].

8 Automatic failure-detector discovery results

Section 6 introduced a mechanical procedure for comparing failure detector strength in a system with a given number of processes. The game-theoretic approach of Section 7 dramatically improved the efficiency by using standard game solving techniques (eg. alpha-beta cutting [12], proof-number search [2]). This section gives a glimpse at the failure detector specification space by enumerating eventual failure detectors and their relationships in systems with two and three processes.

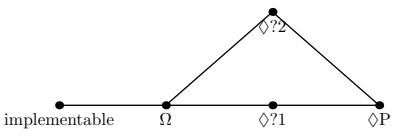
8.1 Two processes, all detectors

This section enumerates and compares all failure detectors with two processes and at most three outputs. The sets $infset(1)$, $infset(2)$, and $infset(12)$ can each take 18 possible values, giving the total of $18^3 = 5832$ failure detectors. Computer testing shows that they all fall into 5 equivalence classes, shown below (left) with several members (right).

	implementable	Ω	◊?1	◊?2	◊P
infset(1)	1	1	□	■	◪
infset(2)	2	2	■	□	◪
infset(12)	12	1,2	□	□	□
		equivalent to			⟶

◊S	?Ω	◊?12	◊?21	◊?P
◪	1■	■□	■	■
◪	2■	■	■□	■
■□, ■□	1□, 2□	□	□	□
Ω	Ω	?1	?2	◊P

The implementability relationship between these classes is



implementable < Ω < ◊?1 < ◊ P
implementable < Ω < ◊?2 < ◊ P

Detectors ◊?1 and ◊?2, which eventually detect whether process 1 (resp. 2) is correct, are of incomparable strength.

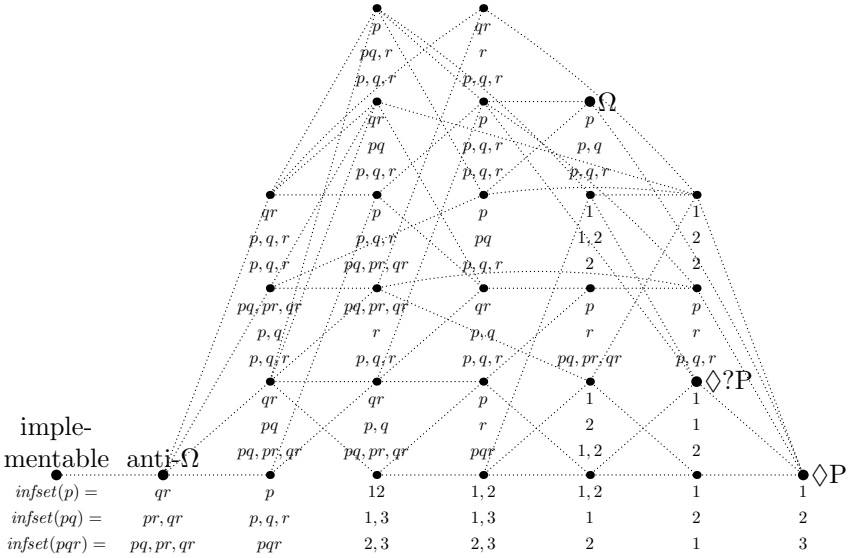


Figure 7: Three-process failure detectors with three outputs.

8.2 Three processes, symmetric detectors

The number of three-process failure detectors with three outputs is $18^7 \approx 6 \times 10^8$. For this reason, this section considers only *symmetric* failure detectors, which treat all processes equally, that is, do not favour any particular permutation of processes or group of such permutations. Such detectors fall into two categories: (i) those that output process-independent symbols, such as ◊?P, and (ii) those that output process ids, such as Ω. There are 6024 such detectors, grouped into 28 equivalence classes shown in Figure 7.

Figure 7 contains several known failure detectors, such as Ω, anti-Ω, and ◊?P. The strongest detector in Figure 7 on page 15 eventually outputs the number k of correct processes. It is equivalent to ◊P, which it can emulate by suspecting the $n - k$ least recently heard-from processes.

The 28 equivalence classes in Figure 7 on page 15 do not contain all symmetric detectors. Detectors that behave as class (i) or (ii), depending on the number of correct processes, form 654 such classes. Allowing non-symmetric detectors and/or more output symbols might increase this number even more. Based on the relatively few failure detectors identified in the literature, such a high number is rather unexpected (and we are only considering systems with three processes here!).

9 Conclusion

This paper investigated the space of eventual failure detectors. The key result is Theorem 1: every implementable detector is a function of the order of recently heard-from processes. By emulating failure detectors with virtual processes corresponding to their outputs, we can use the same technique to compare the strengths of different detectors.

Implementability is also equivalent to a winning strategy in a particular two-player game. The advantage of this approach is that it has more structure and a more intuitive visual representation. This makes failure detectors easier to analyse, and leads to more succinct, intuitive, and elegant proofs, using existing results from game theory. As an

example, this paper identified the weakest eventual failure detector *anti*-Ω. Every query returns a single process; the detector might not stabilize, but there is a correct process that eventually will never be output.

Both approaches produce a finite number of failure detectors, thereby making comprehensive computer search possible. Such a search, applied to three-process detectors with three outputs, generated many known detectors, but also revealed an unexpected richness of non-equivalent failure detector classes. I hope that a similar methodology can be used to explore the space of distributed problems such as consensus, renaming, etc.

The benefits of computer search extend to theoretical results as well, many of which would have been difficult to derive without it. For example, the ability of quick implementability verification was very valuable in identifying anti-Ω and proving its properties. I believe that using computer search as a tool for developing and testing one's intuition about a problem is a useful and productive technique that should become more popular in distributed-computing research.

References

[1] M. K. Aguilera, W. Chen, and S. Toueg. Heartbeat: A timeout-free failure detector for quiescent reliable communication. In *11th WDAG*, pages 126–140, Saarbrücken, Germany, September, 1997.

[2] L. V. Allis. *Searching for Solutions in Games and Artificial Intelligence*. PhD thesis, University of Limburg, the Netherlands, September 1994.

[3] T. D. Chandra, V. Hadzilacos, and S. Toueg. The weakest failure detector for solving Consensus. *Journal of the ACM*, 43(4):685–722, 1996.

[4] C. Delporte-Gallet, H. Fauconnier, R. Guerraoui, V. Hadzilacos, P. Kouznetsov, and S. Toueg. The weakest failure detectors to solve certain fundamental problems in distributed computing. In *23rd PODC*, pages 338–346. St. John's, Newfoundland, Canada, 2004.

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

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

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

[8] R. Guerraoui, M. Hurfin, A. Mostéfaoui, R. Oliveira, M. Raynal, and A. Schiper. Consensus in asynchronous distributed systems: A concise guided tour. In *Advances in Distributed Systems*, number 1752 in Lecture Notes in Computer Science, pages 33–47. Springer, 2000.

[9] R. Guerraoui, M. Herlihy, P. Kouznetsov, N. Lynch, and C. Newport. On the weakest failure detector ever. In *26th PODC*, Portland, OR, US, August 2007.

[10] V. Hadzilacos and S. Toueg. Fault-tolerant broadcast and related problems. In S. Mullender, editor, *Distributed Systems*, chapter 5, pages 97–146. ACM Press, New York, 2nd edition, 1993.

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

[12] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 1995.

[13] P. Zieliński. Anti-Ω: the weakest failure detector for set agreement. Technical Report UCAM-CL-TR-694, Computer Laboratory, University of Cambridge, July 2007.

A Proofs

A.1 Failure detector specifications *infset*(C)

Lemma 2. *Let warp be a strictly increasing function* $\mathbb{N} \to \mathbb{N}$. *For any history hist, let* $hist'(i) = hist(warp(i))$. *If* $\bigcap_t alive(t) = \bigcap_t alive'(t)$, *then* $hist \in \mathcal{H}(alive) \implies hist' \in \mathcal{H}(alive')$.

Proof. Consider two runs r and r', in which steps are taken only by correct processes (C). In r, all processes in C take steps at times $t_i = i$, whereas in r' all processes in C take steps at times $t'_i = warp(i)$. Both r and r' are fair and consistent with the failure pattern *alive*. Without failure detector output, these runs are indistinguishable.

Run r' with a valid history $hist' \in \mathcal{H}(alive)$ produces a sequence of outputs $s'_i = hist(t'_i)$ at each process. By Assumption 3, the same sequence of outputs $s_i = s'_i$ must be possible in run r. Therefore, the history uniquely determined by $hist(i) = s_i = s'_i = hist'(warp(i))$, must be also be valid, which proves the assertion ($hist \in \mathcal{H}(alive)$). □

Theorem 3.

$$\bigcap_t alive(t) = \bigcap_t alive'(t) \implies \mathcal{H}(alive) = \mathcal{H}(alive').$$

Proof. Lemma 2 applied to function $warp(i) = i$ gives us $hist' = hist$, so $hist \in \mathcal{H}(alive) \implies hist \in \mathcal{H}(alive')$. Similarly, $hist \in \mathcal{H}(alive') \implies hist \in \mathcal{H}(alive)$, which implies the assertion. □

Let

$$inf(hist) \stackrel{\text{def}}{=} inf(s_1 \ldots), \quad \text{where } s_i = hist(i).$$

This allows us to rewrite (4) as

$$infset(C) = \{\, inf(hist) \mid hist \in \mathcal{H}(C) \,\}.$$

Theorem 4.

$$inf(hist) = inf(hist'') \quad \implies \quad \big(hist \in \mathcal{H}(C) \iff hist'' \in \mathcal{H}(C)\big).$$

Proof. We shall prove that $hist \in \mathcal{H}(C) \implies hist'' \in \mathcal{H}(C)$. Let $I = inf(hist) = inf(hist'')$. Consider a history $hist'$ which is the same as $hist''$ but with all symbols $\notin I$ is replaced by some symbol $\in I$. Since $hist'(t)$ and $hist''(t)$ differ only for finitely many t, Assumption 1 implies that $hist' \in H(C) \iff hist'' \in \mathcal{H}(C)$. Therefore, we only need to show that $hist \in \mathcal{H}(C) \implies hist' \in \mathcal{H}(C)$.

We will construct a strictly increasing function $warp : \mathbb{N} \to \mathbb{N}$ that satisfies $hist'(i) = hist(warp(i))$. Let us start by introducing an artificial $warp(0) = 0$. Then, let $warp(i)$ be the smallest value larger than $warp(i-1)$ such that $hist'(i) = hist(warp(i))$. Such $warp(i)$ exist because $hist'(i) \in I$, so $hist(t) = hist'(i)$ for infinitely many t, some of them larger than $warp(i-1)$. Having constructed the function *warp*, the assertion follows from Lemma 2. □

Theorem 5. *If* $\emptyset \neq S' \subseteq S \in infset(C)$, *then* $S \in infset(C)$.

Proof. The assumption implies that there is a $hist \in \mathcal{H}(C)$ with $inf(hist) = S$. Consider a subsequence $hist'$ of $hist$, consisting only of the elements in S'. Since $S' \neq \emptyset$, $hist'$ has infinitely many elements. Thus, $hist'(i) = hist(warp(i))$ for some strictly increasing function $warp$. Lemma 2 implies $hist' \in \mathcal{H}(C)$, and $inf(hist') = S'$ implies the assertion. □

Theorem 6. *The virtual-processes and failure-detector models can emulate each other.*

Proof. On the one hand, virtual processes can emulate the failure detector. If each query outputs the id of the most-recently heard-from virtual process, eventually only ids s_i of correct virtual processes S will be output. In other words, $inf(s_1 \ldots) \subseteq S \in infset(C)$, which implies $inf(s_1 \ldots) \in infset(C)$ (Lemma 8).

On the other hand, the detector can emulate virtual processes. When the detector returns a symbol s, the enquiring process should broadcast an "I'm alive" message pretending to be from the virtual process s. □

A.2 Game-theoretic approach

A.2.1 Non-dominated strategies.

In the game described in Section 7, YES never benefits from choosing S_k over another valid move $S'_k \supset S_k$. The same applies to NO and $C'_k \supset C_k$. Such dominated moves S_k and C_k can be eliminated from the game analysis [12]. In particular, we can assume that $C_1 \supset C_2 \supset \cdots$ can be obtained by removing one process at a time from $C_1 = P$. In other words, $C_{k+1} = P \setminus \{q_1 \ldots q_k\}$, for some order $q_1 \ldots q_n \in perms(P)$. Note that, when choosing S_{k+1}, YES knows only the prefix $q_1 \ldots q_k$.

Theorem 7. *YES has a winning strategy iff the failure detector is implementable.*

Proof. ($\Longleftarrow$). As explained above, assume that $C_{k+1} = P \setminus \{q_1 \ldots q_k\}$ for some sequence $q_1 \ldots q_n$. YES has the following winning strategy:

$$S_{k+1} = \bigcup \{\, map(q_1 \ldots q_k r_1 \ldots r_{n-k}) \mid r_1 \ldots r_{n-k} \in perms(C_{k+1}) \,\}.$$

This implies $S_1 \supseteq S_2 \supseteq \cdots \neq \emptyset$, and $S_{k+1} \in infset(C_{k+1})$ from (11).

($\Longrightarrow$). If YES has a winning strategy, let $S_{k+1}(q_1 \ldots q_k)$ be YES's response to NO playing $C_{k+1} = P \setminus \{q_1 \ldots q_k\}$. Define $map(q_1 \ldots q_n)$ as any element of $S_n(q_1 \ldots q_{n-1})$. For $k < n$, we have

$$map(q_1 \ldots q_k r_1 \ldots r_{n-k}) \in S_n(q_1 \ldots q_k r_1 \ldots r_{n-k-1}) \subseteq S_{k+1}(q_1 \ldots q_k) \in infset(C_{k+1}),$$

which, by Theorem 5, implies (11). □

Lemma 8. *In the game from Section 7.1, assuming $S_k \in infset(C_k)$ and $C_{k-1} \supset C_k$ does not change the winner of the game.*

Proof. We will prove that, if the above assumptions do not hold, YES can always repeat its previous move T_{k-1} (assume $T_0 = R_T$). Playing $T_k = T_{k-1}$ does not put any new restrictions on YES's future play, which implies the assertion.

If $S_k \notin \mathit{infset}(C_k)$, then (12) implies $T_k = T_{k-1} \in \mathit{infset}_T([C_k, S_k])$, so the conclusion follows. Therefore, assume $S_k \in \mathit{infset}_S(C_k)$ for all moves $[C_k, S_k]$. We have $[C_{k-1}, S_{k-1}] \supseteq [C_k, S_k] \implies C_{k-1} \supseteq C_k$. Thus, $C_{k-1} \not\supset C_k$ implies $C_{k-1} = C_k$. Therefore, $S_{k-1} \in \mathit{infset}(C_{k-1})$ implies that (12):

$$T_k = T_{k-1} \in \mathit{infset}_T([C_{k-1}, S_{k-1}]) = \mathit{infset}_T(C_{k-1}) = \mathit{infset}_T(C_k) = \mathit{infset}_T([C_k, S_k]).$$

□

A.3 Anti-Ω results

Theorem 9. *Ω is not implementable.*

Proof. After NO's $C_1 = P$, YES has to choose $S_1 = \{p\}$ with $p \in P$. NO wins the game by $C_2 = P \setminus \{p\}$. This forces YES to pick $S_2 = \{q\} \subseteq S_1 = \{p\}$ with $q \neq p$ (impossible). □

Theorem 10. *Anti-Ω is not implementable.*

Proof. NO has the following winning strategy: start with $C_1 = P$ and then always copy YES's last move $C_{k+1} = S_k$. To show $C_1 \supset C_2 \supset \cdots$, we need to prove that $S_k \subset C_k$. This is equivalent to $S_k \subseteq C_k$, as $S_k \neq C_k$ (13). Obviously, $S_1 \subseteq P = C_1$. Then, $S_{k+1} \subseteq S_k = C_{k+1}$. □

Theorem 11. *If $n > 2$, anti-Ω cannot implement ϒ [9].*

Proof. Consider a game in which YES plays C_k and $S_k \in \mathit{infset}_{\text{anti-}\Omega}(C_k)$ (13), and NO plays $T_k \in \mathit{infset}_{\Upsilon}(C_k)$ (14). We need to prove that NO has a winning strategy.

NO starts with $C_1 = P$ and $S_1 = P \setminus \{p\}$ for some $p \in P$. YES has to choose $T_1 = \{T\}$ with $T \neq C_1 = P$. Then, NO plays $C_2 = T \subset C_1$ and $S_2 = P \setminus \{p, t\} \not\supseteq C_2$, for some $t \in T$. Since $C_2 = T$, YES cannot choose $T_2 = T_1 = \{T\}$, and loses.

This proof assumes that $S_2 = P \setminus \{p, t\} \neq \emptyset$, which requires $n = |P| > 2$. In two-process systems, detectors Ω, ϒ, and anti-Ω are all equivalent. □

Theorem 12. *Any non-implementable failure detector S can implement anti-Ω.*

Proof. Consider two games:

- G. Implementing S in the purely asynchronous system. In the k-th turn, NO chooses C_k and YES chooses $S_k \in \mathit{infset}_S(C_k)$. Since S is not implementable, NO has a non-dominated winning strategy.

- $\hat{G}$. Implementing anti-Ω using S. In the k-th turn, $\widehat{\text{NO}}$ chooses $\hat{C}_k$ and $\hat{S}_k \in \mathit{infset}_S(\hat{C}_k)$, $\widehat{\text{YES}}$ chooses $\hat{T}_k \in \mathit{infset}_{\text{anti-}\Omega}(\hat{C}_k)$ (13).

We need to show that $\widehat{\text{YES}}$ has a winning strategy in $\hat{G}$. It consists of two stages. In Stage 1, he copies NO's winning strategy in G while he is forced to output $\hat{T}_k \subseteq \hat{C}_k$. As soon as $\widehat{\text{YES}}$ can play $\hat{T}_k \not\subseteq \hat{C}_k$, he moves to Stage 2, in which he just outputs the same $\hat{T} = \hat{T}_k \setminus \hat{C}_k \neq \emptyset$ for the rest of $\hat{G}$.

Stage 1. In the k-th turn, $\widehat{\text{NO}}$ chooses $\hat{C}_k$ and $\hat{S}_k \in \mathit{infset}_S(\hat{C}_k)$. $\widehat{\text{YES}}$ checks whether $\hat{C}_k = C_k$, NO's optimal move in G, and proceeds to Stage 2 if not. If $\hat{C}_k = C_k$, $\widehat{\text{YES}}$ replies with $\hat{T}_k$ being the optimum move C_{k+1} for NO in G after YES's $S_k = \hat{S}_k$. Such a move exists because NO has a winning strategy in G. Since $\hat{T}_k = C_{k+1} \subset C_k = \hat{C}_k$, we have $\hat{T}_k \in \mathit{infset}_{\text{anti-}\Omega}(\hat{C}_k)$ and $\hat{T}_1 \supset \hat{T}_2 \supset \cdots \neq \emptyset$.

Stage 2. NO's strategy in G is not dominated, so $|C_k| \geq |\hat{C}_k|$. Therefore, $C_k \neq \hat{C}_k$ implies $C_k \setminus \hat{C}_k \neq \emptyset$. As soon as game $\hat{G}$ reaches this state, $\widehat{\text{YES}}$ can then output the same $\hat{T} = C_k \setminus \hat{C}_k \neq \emptyset$ for the rest of the game, because for any $i \geq k$:

$$\big(x \in \hat{C}_i \subseteq \hat{C}_k \implies x \notin \hat{T}\big) \implies \hat{C}_i \nsubseteq \hat{T} \implies \hat{T} \in \mathit{infset}_{\text{anti-}\Omega}(\hat{C}_i).$$

□