HAL
open science

The Minimum Failure Detector For Non-Local Tasks In Message-Passing Systems

Carole Delporte-Gallet, Hugues Fauconnier, Sam Toueg

To cite this version:

Carole Delporte-Gallet, Hugues Fauconnier, Sam Toueg. The Minimum Failure Detector For Non-Local Tasks In Message-Passing Systems. 2009. hal-00401844

HAL Id: hal-00401844

https://hal.science/hal-00401844v1

Preprint submitted on 6 Jul 2009

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

The Minimum Failure Detector For Non-Local Tasks In Message-Passing Systems

Carole Delporte-Gallet
Université Paris Diderot
Paris, France

Hugues Fauconnier
Université Paris Diderot
Paris, France

Sam Toueg
University of Toronto
Toronto, ON, Canada

Abstract

Intuitively, a task is *local* if the output value of each process depends only on the process' own input value, not on the input values of the other processes; a task is *non-local* otherwise.

In this paper, we use the failure detector abstraction to determine the minimum information about failures that is necessary to solve non-local tasks in message-passing systems. More precisely, we show that there is a non-trivial failure detector, denoted $\mathcal{FS}^*$, that is necessary to solve non-local tasks, i.e., $\mathcal{FS}^*$ can be extracted from *any* failure detector that can be used to solve *any* non-local task in message-passing systems. We also show that $\mathcal{FS}^*$ is the strongest failure detector with this property. So, intuitively, $\mathcal{FS}^*$ is the greatest lower bound of the set of failure detectors that solve non-local tasks in message-passing systems.

Even though $\mathcal{FS}^*$ is quite weak, it is strong enough to solve a natural weakening of the well-known *set agreement* task, that we call *weak set agreement*. In fact, we show that $\mathcal{FS}^*$ is the weakest failure detector to solve the weak set agreement task.

Finally, we compare $\mathcal{FS}^*$ to two closely related failure detectors, namely, $\mathcal{L}$ and anti-Ω, which are the weakest failure detectors to solve set agreement in message-passing and shared memory systems, respectively. We prove that anti-Ω is strictly weaker than $\mathcal{FS}^*$ and $\mathcal{FS}^*$ is strictly weaker than $\mathcal{L}$, in message-passing systems.

1 Introduction

In this paper, we investigate the following question: *What is the minimum information about failures that is necessary to solve* any *non-local task in message-passing systems?*

To understand this question, we must first explain what we mean here by "non-local task". Roughly speaking, an (input/output) task is a relation between the input and the output values of processes [2, 14, 16]. In this paper, we consider *one-shot* tasks where each process has a single input value drawn from a finite number of possible input values, and each process outputs a single value. To classify a task as being local or non-local, we consider its input/output requirement in simple systems with no failures. Intuitively, a task is *local* if, in systems with no failures, every process can compute its output value locally by applying some function on its own input value. A task is *non-local* if it is not local.

To illustrate the concept of task locality, consider the trivial "identity" task which requires that every process simply outputs a copy of its input. Intuitively, this task is local: every process can compute its output locally, without any message exchange. Now consider the binary consensus task. This task is not local, in the sense that at least one process cannot compute its output from its individual input only (this holds even in a system where all processes are correct). So consensus is a non-local task.

To determine the minimum information about failures that is necessary to solve non-local tasks, we use the abstraction of failure detectors [4]. Failure detectors have been used to solve several basic problems of

fault-tolerant distributed computing and to capture the minimum information about failures that is necessary to solve these problems (e.g., consensus [4, 3], set agreement [18, 8], non-blocking atomic commit [6], mutual exclusion [7], uniform reliable broadcast [1, 13], boosting obstruction-freedom to wait-freedom [11], implementing an atomic register in a message-passing system [6], etc.).

In this paper, we show that there is a non-trivial failure detector, denoted $\mathcal{FS}^*$, that is necessary to solve non-local tasks in message-passing systems. By this we mean that $\mathcal{FS}^*$ can be extracted from *any* failure detector that can be used to solve *any* non-local task in such systems. We also show that $\mathcal{FS}^*$ is the strongest failure detector with this property. More precisely, we prove that:

1. NECESSITY: $\mathcal{FS}^*$ is necessary to solve non-local tasks, i.e., if a failure detector $\mathcal{D}$ can be used to solve a non-local task $\mathcal{T}$ then $\mathcal{FS}^*$ is weaker than $\mathcal{D}$,[1] and

2. MAXIMALITY: if a failure detector $\mathcal{D}^*$ is necessary to solve non-local tasks, then $\mathcal{D}^*$ is weaker than $\mathcal{FS}^*$.

So, intuitively, $\mathcal{FS}^*$ is the greatest lower bound of the set of failure detectors that solve non-local tasks, and it captures the minimum information about failures necessary for solving such tasks in message-passing systems.

$\mathcal{FS}^*$ is a very weak failure detector, so one may ask wether it is too weak to solve any interesting problem. We show that this is not the case: $\mathcal{FS}^*$ can be used to solve a natural weakening of the well-known *set agreement* task [5], that we call *weak set agreement (*WSA*)*. In fact, we prove that $\mathcal{FS}^*$ is the weakest failure detector to solve this task.[2] Our results imply that, in some precise sense, WSA is the weakest non-local task for message-passing systems: for *any* non-local task $\mathcal{T}$, if $\mathcal{T}$ is solvable using a failure detector $\mathcal{D}$, then WSA is also solvable with $\mathcal{D}$.

Finally, we compare $\mathcal{FS}^*$ to two closely related failure detectors, namely, $\mathcal{L}$ and anti-Ω, which are the weakest failure detectors to solve set agreement in message-passing and shared memory systems, respectively [18, 8]. We prove that anti-Ω is strictly weaker than $\mathcal{FS}^*$ and $\mathcal{FS}^*$ is strictly weaker than $\mathcal{L}$, in message-passing systems.

It is worth noting that the failure detector $\mathcal{FS}^*$ and the weak set agreement task WSA, introduced here, are both very simple. Intuitively, failure detector $\mathcal{FS}^*$ outputs GREEN or RED at each process such that (1) if *all* processes are correct, then $\mathcal{FS}^*$ outputs GREEN forever at some process, and (2) if *exactly one* process is correct, then there is a time after which $\mathcal{FS}^*$ outputs RED at this process. Weak set agreement is like set agreement, except that the condition that there are at most $n-1$ distinct decision values is required *only for failure-free runs*.

Roadmap. We informally describe our message-passing model, and define the concept of a local task, in Section 2. We define $\mathcal{FS}^*$ in Section 3 and prove that it is necessary to solve non-local tasks in Section 4. In Section 5, we show that $\mathcal{FS}^*$ is necessary to solve the weak set agreement task, and, in Section 6, we prove that $\mathcal{FS}^*$ is also sufficient to solve it. In Section 7, we show that $\mathcal{FS}^*$ is the minimum failure detector for solving non-local tasks. We compare $\mathcal{FS}^*$ to anti-Ω and $\mathcal{L}$ in Section 8, and conclude the paper with a brief description of related works in Section 9. We also include an optional Appendix that contains proofs omitted from the paper due to space limitations.

2 Model

Our model is based on the one for unreliable failure detectors described in [3]. We focus here on the main aspects of the model that are necessary to explain our results; details are left to the full version of the

[1] We say that $\mathcal{D}'$ is weaker than $\mathcal{D}$ if processes can use $\mathcal{D}$ to emulate $\mathcal{D}'$ [3], i.e., if $\mathcal{D}'$ can be extracted from $\mathcal{D}$.

[2] This means that (a) $\mathcal{FS}^*$ can be used to solve WSA and (b) $\mathcal{FS}^*$ is weaker than any failure detector that can be used to solve WSA [3].

paper. Henceforth, we assume the existence of a discrete global clock; the range of this clock's ticks is $\mathbb{N}$.

2.1 Asynchronous message-passing systems. We consider distributed message-passing systems with a set of $n \geq 2$ processes $\Pi = \{1, 2, \ldots, n\}$. Processes execute steps asynchronously and they communicate with each other by sending messages through reliable but asynchronous communication links. Each process has access to a failure detector module that provides some information about failures, as explained below.

2.2 Failures, failure patterns and environments. Processes are subject to *crash failures*, i.e., they may stop taking steps. A *failure pattern* is a function $F : \mathbb{N} \rightarrow 2^{\Pi}$, where $F(t)$ is the set of processes that have crashed through time t. Processes never recover from crashes, and so $F(t) \subseteq F(t+1)$. Let $faulty(F) = \bigcup_{t \in \mathbb{N}} F(t)$ be the set of faulty processes in a failure pattern F; and $correct(F) = \Pi - faulty(F)$ be the set of correct processes in F. When the failure pattern F is clear from the context, we say that process p is *correct* if $p \in correct(F)$, and p is *faulty* if $p \in faulty(F)$. The *failure-free failure pattern*, i.e., the pattern where all the processes are correct, is denoted F_{ff}.

An *environment* $\mathcal{E}$ is a non-empty set of failure patterns. Intuitively, an environment describes the number and timing of failures that can occur in the system. We denote by $\mathcal{E}^*$ the set of *all* failure patterns. Intuitively, in a system with environment $\mathcal{E}^*$ each process may crash, and it may do so any time.

2.3 Failure detectors. Each time a process queries its failure detector module, the response that it gets is a finite binary string from $\{0, 1\}^*$. A *failure detector history* describes the behavior of a failure detector during an execution. Formally, it is a function $H : \Pi \times \mathbb{N} \rightarrow \{0, 1\}^*$, where $H(p, t)$ is the value output by the failure detector module of process p at time t.

A *failure detector* $\mathcal{D}$ is a function that maps every failure pattern F to a nonempty set of failure detector histories. $\mathcal{D}(F)$ is the set of all possible failure detector histories that may be output by $\mathcal{D}$ when the failure pattern is F. Typically we specify a failure detector by stating the properties that its histories satisfy. The *trivial failure detector* $\mathcal{D}_{\perp}$ always outputs $\perp$ at all processes, forever: $\forall F \in \mathcal{E}^*, \forall p \in \Pi, \forall t \in \mathbb{N}, , \forall H \in \mathcal{D}(F) : H(p, t) = \perp$.

2.4 Message buffer. A *message buffer*, denoted M, contains all the messages that were sent but not yet received. When a process p attempts to receive a message, it either receives a message $m \in M$ addressed to p or the "empty" message $\perp$.

2.5 Input/Output variables. To model input/output tasks, we assume that each process p can read an *input variable*, denoted $IN(p)$, and write an *output variable*, denoted $OUT(p)$; both variables are external to p. We assume that $IN(p)$ is initialized to some input value in $\{0,1\}^*$, and that $IN(p)$ does not change after its initialization. Moreover, we assume that $OUT(p)$ is initialized to the special value $\perp \notin \{0,1\}^*$ (to denote that it was not yet written by p).

2.6 Algorithms. An *algorithm* $\mathcal{A}$ consists of n deterministic automata, one for each process; the automaton for process p is denoted $\mathcal{A}(p)$. The execution of an algorithm $\mathcal{A}$ proceeds as a sequence of *process steps*. In a step, a process p performs the following actions atomically: (1) receive a single message m from the message buffer M, or the empty message $\perp$; (2) read its input variable $IN(p)$; (3) query its local failure detector module and receive some value d; (4) change its state; (5) may write a value in its output variable $OUT(p)$; and (6) may send messages to other processes.

2.7 Runs of an algorithm. A *run of algorithm $\mathcal{A}$ using failure detector $\mathcal{D}$ in environment $\mathcal{E}$* is a tuple $R = (F, H, I, S, T)$ where F is a failure pattern in $\mathcal{E}$, H is a failure detector history in $\mathcal{D}(F)$, I is an initial input ($I(p)$ is the initial value of the input variable $IN(p)$, for each p), S is a sequence of steps of $\mathcal{A}$, and T is a list of times in $\mathbb{N}$ ($T[i]$ is the time when step $S[i]$ is taken) such that F, H, I, S, T satisfy some standard validity conditions. Specifying the above conditions formally is straightforward (e.g., see [3]) but tedious. Since this formalization is not necessary to present our results, we omit it from this extended abstract.

Since we focus on algorithms that solve *one-shot* input/output tasks, we restrict our attention to algorithms where each process writes its output variable at most once. That is, henceforth we consider only algorithms $\mathcal{A}$ that satisfy the following condition: For any failure detector $\mathcal{D}$, any environment $\mathcal{E}$, and any run R of $\mathcal{A}$ using $\mathcal{D}$ in $\mathcal{E}$, every process p writes $OUT(p)$ at most once.

2.8 The input/output of a run. Let $R = (F, H, I, S, T)$ be a run of an algorithm $\mathcal{A}$ (using some failure detector $\mathcal{D}$ in an environment $\mathcal{E}$). The processes' input/output behavior in run R is defined as follows. The *input of run R* is I; recall that for each process $p \in \Pi$, $I(p)$ is the initial value of the input variable $IN(p)$ in R. The *output of run R*, denoted O, is the vector of values written by processes in run R; more precisely, for each process $p \in \Pi$, $O(p)$ is the value that p writes in its variable $OUT(p)$ in R, and $O(p) = \perp$ if p never writes $OUT(p)$ in run R. It is also convenient to say that F *is the failure pattern of run R*.

2.9 Input/output tasks. We consider *one-shot input/output tasks*, i.e., problems where each process has an input value and writes an output value. To specify a task, we must give the set of possible input values, the set of possible output values, and the input/output behaviors that satisfy the task. For any task $\mathcal{T}$, each process p has a non-empty set $\mathcal{I}_p$ of possible input values, and a set $\mathcal{O}_p$ of possible output values that contains the special value $\perp$ (intuitively, $\perp$ denotes an empty output). Henceforth, $\mathcal{I} = \mathcal{I}_1 \times \mathcal{I}_2 \times \cdots \times \mathcal{I}_n$ and $\mathcal{O} = \mathcal{O}_1 \times \mathcal{O}_2 \times \cdots \times \mathcal{O}_n$; moreover, $I \in \mathcal{I}$ and $O \in \mathcal{O}$ denote vectors of input and output values, respectively: one value for each of the processes $\{1, 2, \ldots, n\}$ of Π.

To specify a task $\mathcal{T}$, we must specify the desired input/output behavior of processes under each possible failure pattern. We can do so by giving a set $\mathcal{T}_S$ of tuples of the form (F, I, O): intuitively, $(F, I, O) \in \mathcal{T}_S$ if and only if, when the failure pattern is F and the processes input is I, the processes output O is acceptable, i.e., it "satisfies" task $\mathcal{T}$. This definition of a task is a generalization of the ones given in [2, 14, 16] which are based solely on the input/output requirement in failure-free runs. This generalization allows us to capture tasks where the desired input/output behavior depends on the failure pattern. For example, in the *Atomic Commit* task, if *all* processes vote COMMIT (this is the input), then processes are allowed to output ABORT *if and only if a failure occurs*.

In summary, a task $\mathcal{T}$ is specified by giving (1) the sets $\mathcal{I}_p$ and $\mathcal{O}_p$ of possible input and output values of each process $p \in \Pi$, and (2) a set $\mathcal{T}_S$ of tuples of the form (F, I, O), where $F \in \mathcal{E}^*$, $I \in \mathcal{I} = \mathcal{I}_1 \times \mathcal{I}_2 \times \cdots \times \mathcal{I}_n$, and $O \in \mathcal{O} = \mathcal{O}_1 \times \mathcal{O}_2 \times \cdots \times \mathcal{O}_n$, such that:

1. ($\mathcal{T}$ *is well-defined*) For every possible failure pattern and every possible input, there is at least one output that satisfies $\mathcal{T}$:

   $\forall F \in \mathcal{E}^*, \forall I \in \mathcal{I}, \exists O \in \mathcal{O} : (F, I, O) \in \mathcal{T}_S$

2. ($\mathcal{T}$ *terminates*) Every correct process outputs some value:

   $\forall (F, I, O) \in \mathcal{T}_S, \forall p \in correct(F) : O(p) \neq \perp$

In this paper, we consider only tasks where each process $p \in \Pi$ has a *finite* set of possible inputs $\mathcal{I}_p$.

2.10 Solving an input/output task. Let $\mathcal{T}$ be a task defined by some sets $\mathcal{I}_p$ and $\mathcal{O}_p$ (for each process p), and $\mathcal{T}_S$. Let $\mathcal{A}$ be an algorithm, $\mathcal{D}$ a failure detector, and $\mathcal{E}$ an environment. We say that:

- A run $R = (F, H, I, S, T)$ of $\mathcal{A}$ using $\mathcal{D}$ in $\mathcal{E}$ *satisfies* $\mathcal{T}$ if and only if either $I \notin \mathcal{I}_1 \times \mathcal{I}_2 \times \cdots \times \mathcal{I}_n$, or $(F, I, O) \in \mathcal{T}_S$, where F, I and O are the failure pattern, the input and the output of run R, respectively.
- $\mathcal{A}$ *solves* $\mathcal{T}$ *using* $\mathcal{D}$ *in* $\mathcal{E}$ if and only if every run R of $\mathcal{A}$ using $\mathcal{D}$ in $\mathcal{E}$ satisfies $\mathcal{T}$.
- $\mathcal{D}$ *can be used to solve* $\mathcal{T}$ *in* $\mathcal{E}$ if and only if there is an algorithm that solves $\mathcal{T}$ using $\mathcal{D}$ in $\mathcal{E}$.

2.11 Local versus non-local tasks. To classify a task as being local or non-local we consider its input/output requirement in the simple case of systems with no failures. Intuitively, we say that a task $\mathcal{T}$ is local if, in a system with no failures, each process can determine its output value *based only on its input value*.

More precisely, let $\mathcal{T}$ be a task specified by some sets $\mathcal{I}_p$ and $\mathcal{O}_p$ of possible input and output values for each process p, and a set $\mathcal{T}_S$. We say that $\mathcal{T}$ *is local* if and only if there are functions $f_1, f_2, \ldots, f_n$ such that for each possible input $I = (i_1, i_2, \ldots, i_n) \in \mathcal{I}_1 \times \mathcal{I}_2 \times \cdots \times \mathcal{I}_n$ of $\mathcal{T}$, the output $O = (f_1(i_1), f_2(i_2), \ldots, f_n(i_n))$ satisfies the specification of $\mathcal{T}$ when all the processes are correct, i.e., $(F_{ff}, I, O) \in \mathcal{T}_S$. Note that for each process p, the output $f_p(i_p)$ of p depends only on the input i_p of p, and so p can compute its own output locally. We say that $\mathcal{T}$ *is non-local* if and only if $\mathcal{T}$ is not local.

2.12 Comparing failure detectors. To compare two failure detectors $\mathcal{D}$ and $\mathcal{D}'$ in some environment $\mathcal{E}$, we use the concept of failure detector transformation. Intuitively, an algorithm transforms $\mathcal{D}$ to $\mathcal{D}'$ if it can use $\mathcal{D}$ to emulate (the failure detector outputs of) $\mathcal{D}'$. Such an algorithm, denoted $\mathcal{T}_{\mathcal{D}\rightarrow\mathcal{D}'}$, uses $\mathcal{D}$ to maintain a local variable $out\text{-}\mathcal{D}'_p$ at every process p; $out\text{-}\mathcal{D}'_p$ functions as the output of the emulated failure detector module $\mathcal{D}'_p$ of $\mathcal{D}'$ at p. For each run R of $\mathcal{T}_{\mathcal{D}\rightarrow\mathcal{D}'}$, let H_{out} be the history of all the $out\text{-}\mathcal{D}'$ variables in R; i.e., $H_{out}(p, t)$ is the value of $out\text{-}\mathcal{D}'_p$ at time t in R. Algorithm $\mathcal{T}_{\mathcal{D}\rightarrow\mathcal{D}'}$ *transforms* $\mathcal{D}$ *to* $\mathcal{D}'$ *in environment* $\mathcal{E}$ if and only if for every run R of $\mathcal{T}_{\mathcal{D}\rightarrow\mathcal{D}'}$ using $\mathcal{D}$ in $\mathcal{E}$, $H_{out} \in \mathcal{D}'(F)$. We say that:

- $\mathcal{D}'$ *is weaker than* $\mathcal{D}$ *in* $\mathcal{E}$, if there is an algorithm $\mathcal{T}_{\mathcal{D}\rightarrow\mathcal{D}'}$ that transforms $\mathcal{D}$ to $\mathcal{D}'$ in $\mathcal{E}$.
- $\mathcal{D}'$ *is necessary to solve a task* $\mathcal{T}$ *in* $\mathcal{E}$, if, for every failure detector $\mathcal{D}$ that can be used to solve $\mathcal{T}$ in $\mathcal{E}$, $\mathcal{D}'$ is weaker than $\mathcal{D}$ in $\mathcal{E}$.
- $\mathcal{D}'$ *is necessary to solve non-local tasks in* $\mathcal{E}$ if, for every non-local task $\mathcal{T}$, $\mathcal{D}'$ is necessary to solve $\mathcal{T}$ in $\mathcal{E}$.

3 The $\mathcal{FS}^*$ failure detector

The failure detector $\mathcal{FS}^*$ outputs GREEN or RED at each process. If *all* processes are correct, then $\mathcal{FS}^*$ outputs GREEN forever at some process. If *exactly one* process is correct, then there is a time after which $\mathcal{FS}^*$ outputs RED at this process. (Note that since we consider systems with at least $n \geq 2$ processes, these two preconditions are mutually exclusive.) Formally, for every failure pattern $F \in \mathcal{E}^*$:

$$\begin{aligned}\mathcal{FS}^*(F) = \big\{H \mid &(\forall p \in \Pi, \forall t \in \mathbb{N}: \ H(p,t) = \textsc{Green} \vee H(p,t) = \textsc{Red}) \wedge \\ &(|\mathit{correct}(F)| = n \Rightarrow \exists p \in \Pi, \forall t \in \mathbb{N}: \ H(p,t) = \textsc{Green}) \wedge \\ &(|\mathit{correct}(F)| = 1 \Rightarrow \exists p \in \mathit{correct}(F), \exists t \in \mathbb{N}, \forall t' \geq t: \ H(p,t') = \textsc{Red})\big\}\end{aligned}$$

We observe that $\mathcal{FS}^*$ is non-trivial in the sense that it cannot be implemented "from scratch" in an asynchronous system where each process may crash at any time. Formally, we say that an algorithm implements $\mathcal{FS}^*$ in $\mathcal{E}^*$, if it transforms the trivial failure detector $\mathcal{D}_\perp$ into $\mathcal{FS}^*$ in environment $\mathcal{E}^*$. The following observation is obvious; its proof is in the optional Appendix A.

Observation 1 *No algorithm implements $\mathcal{FS}^*$ in $\mathcal{E}^*$.*

4 $\mathcal{FS}^*$ is necessary to solve non-local tasks

We now show that for every non-local task $\mathcal{T}$, $\mathcal{FS}^*$ is necessary to solve $\mathcal{T}$. We start with the following lemma that relates the ability to solve a task without exchanging messages to the definition of task locality given in Section 2.

Lemma 2 *Let $\mathcal{T}$ be any task and $\mathcal{E}$ be any environment that contains the failure-free failure pattern F_{ff}. Suppose that there is an algorithm $\mathcal{A}$, a failure detector $\mathcal{D}$, and a failure detector history $H \in \mathcal{D}(F_{ff})$ such that:*

1. *$\mathcal{A}$ solves $\mathcal{T}$ using $\mathcal{D}$ in $\mathcal{E}$, and*
2. *for every input $I \in \mathcal{I} = \mathcal{I}_1 \times \mathcal{I}_2 \times \cdots \times \mathcal{I}_n$, there is a run $R_I = (F_{ff}, H, I, S, T)$ of $\mathcal{A}$ using $\mathcal{D}$ in $\mathcal{E}$ such that every process outputs a value before receiving any message.*

Then $\mathcal{T}$ is a local task.

PROOF. Let $\mathcal{T}$ be any task, and let $\mathcal{I}_p$ and $\mathcal{O}_p$ be the corresponding sets of possible input and output values for each process p, and $\mathcal{T}_S$ be the corresponding set of acceptable (F, I, O) tuples. Let $\mathcal{E}$ be any environment that contains F_{ff}. Suppose that there is an algorithm $\mathcal{A}$, a failure detector $\mathcal{D}$, and a failure detector history $H \in \mathcal{D}(F_{ff})$, that satisfy conditions (1) and (2) of the lemma. In the proof below, we fix some input $I = (i_1, i_2, \ldots, i_n) \in \mathcal{I} = \mathcal{I}_1 \times \mathcal{I}_2 \times \cdots \times \mathcal{I}_n$ of task $\mathcal{T}$.

We now show that task $\mathcal{T}$ is local by first defining a function $f_p : \mathcal{I}_p \to \mathcal{O}_p$ for every process $p \in \Pi$, and then showing that these functions give correct outputs. For each process $p \in \Pi$, and every possible input $z \in \mathcal{I}_p$ of p, we start by defining a run R_p^z of $\mathcal{A}$, which in turn defines the value of $f_p(z)$.

To define R_p^z and $f_p(z)$, let $I_p^z \in \mathcal{I}$ be the input of $\mathcal{T}$ that is identical to I except that the input of process p is z; more precisely, $I_p^z(p) = z$ and, for all $q \neq p$, $I_p^z(q) = I(q)$. By our assumptions (1) and (2) on $\mathcal{A}$, $\mathcal{D}$, and $H \in \mathcal{D}(F_{ff})$, there is a run $R_p^z = (F_{ff}, H, I_p^z, S_p^z, T_p^z)$ of $\mathcal{A}$ using $\mathcal{D}$ in $\mathcal{E}$, such that every process outputs a value before receiving any message. We define $f_p(z)$ to be the output of process p in run R_p^z,[3] and t_p^z to be the time when this output occurs in run R_p^z.

[3]Note that since $\mathcal{A}$ solves $\mathcal{T}$ using $\mathcal{D}$ in $\mathcal{E}$, the output of p in run R_p^z must be in $\mathcal{O}_p$.

To prove that $\mathcal{T}$ is a local task, it now suffices to show that for every input $I = (x_1, x_2, \ldots, x_n) \in \mathcal{I}$, we have $(F_{ff}, I, O) \in \mathcal{T}_S$ where $O = (f_1(x_1), f_2(x_2), \ldots, f_n(x_n))$. To show this, we construct a run $R = (F_{ff}, H, I, S, T)$ of $\mathcal{A}$ such that, in the sequence of steps S and the corresponding times T, each process $p \in \Pi$ takes exactly the same sequence of steps at the same times as in the run $R_p^{x_p} = (F_{ff}, H, I_p^{x_p}, S_p^{x_p}, T_p^{x_p})$ up to time $t_p^{x_p}$.[4] Note that for every process $p \in \Pi$, runs R and $R_p^{x_p}$ are indistinguishable up to time $t_p^{x_p}$; this is because in both runs: (a) process p has the same input, namely x_p, (b) process p receives no messages up to time $t_p^{x_p}$, (c) the failure dectector history H is the same, and (d) process p takes the same sequence of steps, at the same times, up to time $t_p^{x_p}$. Thus, in run R, every process $p \in \Pi$ outputs at time $t_p^{x_p}$ the same value that it outputs in run $R_p^{x_p}$, which is, by definition, $f_p(x_p)$. So the output of run R is $O = (f_1(x_1), f_2(x_2), \ldots, f_n(x_n))$. Since R is a run of $\mathcal{A}$ using $\mathcal{D}$ in $\mathcal{E}$, and $\mathcal{A}$ solves $\mathcal{T}$ using $\mathcal{D}$ in $\mathcal{E}$, we have $(F_{ff}, I, O) \in \mathcal{T}_S$, as we wanted to show. □

Theorem 3 *For every environment $\mathcal{E}$, failure detector $\mathcal{FS}^*$ is necessary to solve non-local tasks in $\mathcal{E}$.*

PROOF. Let $\mathcal{E}$ be any environment. We must show that for every non-local task $\mathcal{T}$, if a failure detector $\mathcal{D}$ can be used to solve $\mathcal{T}$ in $\mathcal{E}$, then $\mathcal{FS}^*$ is weaker than $\mathcal{D}$ in $\mathcal{E}$, i.e., there is an algorithm that transforms $\mathcal{D}$ to $\mathcal{FS}^*$ in $\mathcal{E}$.

Let $\mathcal{T}$ be any non-local task, $\mathcal{D}$ be any failure detector that can be used to solve $\mathcal{T}$ in $\mathcal{E}$, and $\mathcal{A}$ be any algorithm that solves $\mathcal{T}$ using $\mathcal{D}$ in $\mathcal{E}$. We now describe an algorithm $\mathcal{T}_{\mathcal{D}\rightarrow\mathcal{FS}^*}$ that uses $\mathcal{A}$ and $\mathcal{D}$ to transform $\mathcal{D}$ to $\mathcal{FS}^*$ in $\mathcal{E}$. In the following, $\mathcal{I}_p = \{i_p^1, i_p^2, \ldots, i_p^{k_p}\}$ is the set of possible input values of process p in task $\mathcal{T}$ (recall that we consider only tasks with a finite number of inputs).

The transformation algorithm $\mathcal{T}_{\mathcal{D}\rightarrow\mathcal{FS}^*}$, shown in Figure 1, emulates the output of $\mathcal{FS}^*$ as follows. Each process $p \in \Pi$ has a local variable $out\text{-}\mathcal{FS}_p^*$ that represents the output of the failure detector module $\mathcal{FS}_p^*$ of $\mathcal{FS}^*$. This variable is initialized to GREEN at every process $p \in \Pi$. In $\mathcal{T}_{\mathcal{D}\rightarrow\mathcal{FS}^*}$, every process $p \in \Pi$ concurrently executes several independent instances of the algorithm $\mathcal{A}$ (these executions proceed in a round-robin way to ensure the progress of every execution). More precisely, for each input $i_p \in \mathcal{I}_p$, process p emulates an execution of the local code $\mathcal{A}_p$ of the algorithm $\mathcal{A}$ with input i_p, using the given failure detector $\mathcal{D}$. In each emulated execution, process p faithfully follows the "code" of $\mathcal{A}_p$, except that when a message is received, p discards the message and continues this execution of $\mathcal{A}_p$ *as if no message was actually received*. Note that by doing so, it is possible that in some (or all) of these emulated executions of $\mathcal{A}_p$, process p never outputs a value for task $\mathcal{T}$. If *every* emulated execution of $\mathcal{A}_p$ by process p actually outputs some value, then process p sets its local variable $out\text{-}\mathcal{FS}_p^*$ to RED; thereafter $out\text{-}\mathcal{FS}_p^* =$ RED forever.

CLAIM: *The algorithm in Figure 1 transforms $\mathcal{D}$ to $\mathcal{FS}^*$ in environment $\mathcal{E}$.*

PROOF: Consider an *arbitrary* run R of algorithm $\mathcal{T}_{\mathcal{D}\rightarrow\mathcal{FS}^*}$ using $\mathcal{D}$ in $\mathcal{E}$. Let $F \in \mathcal{E}$ be the failure pattern of R and $H \in \mathcal{D}(F)$ be the failure detector history of R. We must show that, in run R, the local variables $out\text{-}\mathcal{FS}_p^*$, which are maintained by $\mathcal{T}_{\mathcal{D}\rightarrow\mathcal{FS}^*}$, emulate the output of failure detector $\mathcal{FS}^*$; i.e., their values satisfy the two properties of $\mathcal{FS}^*$ stated in Section 3: (1) if exactly one process is correct in R, say it is process p, then there is a time after which $\mathcal{FS}_p^* =$ RED, and (2) if all processes are correct in R then, at some process q, we have $\mathcal{FS}_q^* =$ GREEN forever. So we only need to consider the following two cases.

Case 1: $|correct(F)| = 1$, i.e., there is exactly one correct process in run R. Let p be this correct process and $i_p \in \mathcal{I}_p$ be any input of p. Consider the run of $\mathcal{A}_p$ with input i_p that is emulated by process p in run R of $\mathcal{T}_{\mathcal{D}\rightarrow\mathcal{FS}^*}$. To process p, this emulated run is indistinguishable from a run $R' = (F, H, I', S', T')$

[4] After time $t_p^{x_p}$, the steps of process p in run R may diverge from those it takes in run $R_p^{x_p}$; in particular, after time $t_p^{x_p}$ in R, p receives all the messages that were previously sent to it in run R, *not* those that were sent in run $R_p^{x_p}$.

CODE FOR PROCESS p

Local Variables:

$out\text{-}\mathcal{FS}^*_p = \text{GREEN}$ { variable that emulates the output of $\mathcal{FS}^*_p$ }
$j = 1$ { p executes $\mathcal{A}_p$ with input i^j_p for every j, $1 \leq j \leq k_p$ }
$outputs = 0$ { number of executions of $\mathcal{A}_p$ that output a value so far }

Main Code:

1 **while** $outputs < k_p$ **do** { continue until all k_p executions of $\mathcal{A}_p$ output a value }
2 emulate the next step of $\mathcal{A}_p$ with input $i^j_p \in \mathcal{I}_p$ using $\mathcal{D}$, except that
any message received in this step is discarded as if no message was received
3 **if** this step of $\mathcal{A}_p$ outputs a value (for task $\mathcal{T}$)
4 **then** $outputs \leftarrow outputs + 1$
5 $j \leftarrow (j \bmod k_p) + 1$
6 $out\text{-}\mathcal{FS}^*_p = \text{RED}$

Figure 1: Transformation algorithm $\mathcal{T}_{\mathcal{D}\rightarrow\mathcal{FS}^*}$.

of $\mathcal{A}$ using $\mathcal{D}$ in $\mathcal{E}$ such that (1) p's input in $I'(p)$ is i_p, (2) p is the only correct process in R', and (3) all the other processes take no steps before they crash in R' (that's why p never receives any message in this run). Since $\mathcal{A}$ solves $\mathcal{T}$ in $\mathcal{E}$, by the termination requirement of tasks (namely, that each correct process must output a value), p eventually outputs a value in this emulated run of $\mathcal{A}_p$ with input i_p. Thus, each of the k_p runs of $\mathcal{A}_p$ (one run for each possible input $i_p \in \mathcal{I}_p$) that p emulates in run R eventually outputs some value. So eventually p exits the while loop of line 1, it sets its local variable $out\text{-}\mathcal{FS}^*_p$ to RED in line 6, and thereafter $out\text{-}\mathcal{FS}^*_p = \text{RED}$.

Case 2: $|correct(F)| = n$, i.e., $F = F_{ff}$ and all processes are correct in run R. We must show that there is some process q such that $out\text{-}\mathcal{FS}^*_q = \text{GREEN}$, forever. Suppose, for contradiction, that this does not hold. Then, it must be that every process $p \in \Pi$ eventually reaches line 6 and sets $out\text{-}\mathcal{FS}^*_p$ to RED in run R. So, for every process $p \in \Pi$ and every input $i_p \in \mathcal{I}_p$, the execution of $\mathcal{A}_p$ with input i_p that p emulates in run R eventually outputs some value (even though p never receives any message in this emulation). Let t_{out} be the time the last such output occurs (across all processes and all inputs) in run R.

Consider any input vector $I = (i_1, i_2, \ldots, i_n) \in \mathcal{I} = \mathcal{I}_1 \times \mathcal{I}_2 \times \cdots \times \mathcal{I}_n$ of task $\mathcal{T}$. We now show that there is a run $R_I = (F_{ff}, H, I, S', T')$ of $\mathcal{A}$ using $\mathcal{D}$ in $\mathcal{E}$ such that every process outputs a value before receiving any messages. We construct run $R_I = (F_{ff}, H, I, S', T')$ of $\mathcal{A}$ using, for each process p, the emulated execution of $\mathcal{A}_p$ with input i_p in run R, as follows. In run R_I, for every process $p \in \Pi$: (1) up to time t_{out}, p takes the same steps at the same times as in the emulated execution of $\mathcal{A}_p$ with input i_p in run R (in these steps p does not receive any message and it outputs some value), and (2) after time t_{out}, p continues its execution of $\mathcal{A}_p$ with input i_p, but now p starts receiving every message sent to it by every other process $q \in \Pi$ in its execution of $\mathcal{A}_q$ with input i_q (including messages that were discarded by p in its emulation of $\mathcal{A}_p$ with input i_p in run R). In other words, run $R_I = (F_{ff}, H, I, S', T')$ of $\mathcal{A}$ is built as follows: (1) up to time t_{out}, R_I is the merging of the n independent executions of $\mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_n$ with the local inputs $i_1, i_2, \ldots, i_n$, emulated by processes $1, 2, \ldots, n$, respectively, in run R (in these emulated executions no messages are received — it is as if all messages are delayed to after time t_{out} — and every process outputs some value by time t_{out}); and (2) after time t_{out}, in R_I processes $1, 2, \ldots, n$ continue their executions of $\mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_n$ with local inputs $i_1, i_2, \ldots, i_n$, respectively, such that they eventually receive every message that are sent to them in these executions, including all those sent before time t_{out}.

Note that $R_I = (F_{ff}, H, I, S', T')$ is a run of $\mathcal{A}$ using $\mathcal{D}$ in $\mathcal{E}$. Moreover, in run R_I every process outputs a value by time t_{out}, before receiving any message. Since I is an *arbitrary* input in $\mathcal{I}$, it is now clear that $\mathcal{A}$, $\mathcal{D}$ and $H \in \mathcal{D}(F_{ff})$ satisfy both conditions (1) and (2) of Lemma 2. Note also that $\mathcal{E}$ includes F_{ff}. So, by Lemma 2, $\mathcal{T}$ is a *local* task — a contradiction. We conclude that there is some process q such that $out\text{-}\mathcal{FS}^*_q =$ GREEN, forever.

Case 1 and Case 2 show that the local variables $out\text{-}\mathcal{FS}^*$ emulate the output of failure detector $\mathcal{FS}^*$ correctly, as we needed to show. □

5 Weak set agreement

Weak set agreement (WSA) is a weaker version of the well-known *set agreement* task [5]: the condition that there are at most $n-1$ distinct decision values is required *only for failure-free runs*. More precisely, each process can propose any value in $V = \{1, 2, \ldots, n\}$ and must decide some value such that:

- TERMINATION: Every correct process eventually decides some value.
- VALIDITY: Each decided value was proposed by some process.
- WEAK AGREEMENT: If all processes are correct then there are at most $n-1$ distinct decision values.

It is clear that WSA can be formally defined as an input/output task. The proof of the following theorem is in the optional Appendix B.

Theorem 4 *WSA is a non-local task.*

Corollary 5 *For every environment $\mathcal{E}$, $\mathcal{FS}^*$ is necessary to solve WSA in $\mathcal{E}$.*

PROOF. This is an immediate consequence of Theorems 3 and 4. □

6 Solving weak set agreement using $\mathcal{FS}^*$

We now prove that $\mathcal{FS}^*$ can be used to solve the weak set agreement task, and in fact it is the *weakest failure detector* to solve this task; intuitively, this means that $\mathcal{FS}^*$ is necessary and sufficient for this task [3]. More precisely: A failure detector $\mathcal{D}$ is the *weakest failure detector to solve a task $\mathcal{T}$ in $\mathcal{E}$* iff:[5]

- NECESSITY: $\mathcal{D}$ is necessary to solve $\mathcal{T}$ in $\mathcal{E}$, and
- SUFFICIENCY: $\mathcal{D}$ can be used to solve $\mathcal{T}$ in $\mathcal{E}$.

The algorithm in Figure 2 solves the weak set agreement task using $\mathcal{FS}^*$. It is identical to the one that solves the set agreement task using $\mathcal{L}$ given in [8]. In this algorithm, lines 3-5 and lines 6-8 are executed atomically. The proof of correctness is in the optional Appendix C.

Theorem 6 *For every environment $\mathcal{E}$, the algorithm in Figure 2 solves WSA using $\mathcal{FS}^*$ in $\mathcal{E}$.*

[5]There may be several distinct failure detectors that are the weakest to solve a task $\mathcal{T}$. It is easy to see, however, that they are all equivalent in the sense that each is weaker than the other. For this reason we speak of *the* weakest, rather than *a* weakest failure detector to solve $\mathcal{T}$.

CODE FOR PROCESS p:

```
to propose(v_p)                                   { v_p is p's proposal value }
    send v_p to every process q > p
{ p decides a value as follows: }
    upon receipt of a value v do
        send v to all
        decide v ; halt
    upon FS*_p = RED do
        send v_p to all
        decide v_p ; halt
```

Figure 2: Using $\mathcal{FS}^*$ to solve WSA in any environment $\mathcal{E}$.

Corollary 7 *For every environment $\mathcal{E}$, $\mathcal{FS}^*$ is the weakest failure detector to solve WSA in $\mathcal{E}$.*

PROOF. This is an immediate consequence of Corollary 5 and Theorem 6. □

We conclude that WSA is the weakest non-local task in message-passing systems in the following sense:

Corollary 8 *For every environment $\mathcal{E}$, any non-local task $\mathcal{T}$, and any failure detector $\mathcal{D}$, if $\mathcal{D}$ can be used to solve $\mathcal{T}$ in $\mathcal{E}$, then $\mathcal{D}$ can also be used to solve WSA in $\mathcal{E}$.*

PROOF (SKETCH). Since $\mathcal{D}$ can be used to solve $\mathcal{T}$ in $\mathcal{E}$, by Theorem 3, $\mathcal{FS}^*$ is weaker than $\mathcal{D}$ in $\mathcal{E}$. So processes can use $\mathcal{D}$ to emulate $\mathcal{FS}^*$ in $\mathcal{E}$, and then they can use this emulated $\mathcal{FS}^*$ to solve WSA with the algorithm in Figure 2. □

7 $\mathcal{FS}^*$ is the minimum failure detector for non-local tasks

We now show that $\mathcal{FS}^*$ is the strongest failure detector necessary to solve non-local tasks.

Theorem 9 *Let $\mathcal{E}$ be any environment.*

1. (NECESSITY:) *$\mathcal{FS}^*$ is necessary to solve non-local tasks in $\mathcal{E}$.*
2. (MAXIMALITY:) *Every failure detector $\mathcal{D}$ that is necessary to solve non-local tasks in $\mathcal{E}$ is weaker than $\mathcal{FS}^*$ in $\mathcal{E}$.*

PROOF. Part (1) was already shown in Theorem 3. We now prove Part (2). Let $\mathcal{E}$ be any environment. Let $\mathcal{D}$ be any failure detector that is necessary to solve non-local tasks in $\mathcal{E}$, i.e.: (*) for any non-local task $\mathcal{T}$, if a failure detector $\mathcal{D}'$ can be used to solve $\mathcal{T}$ in $\mathcal{E}$ then $\mathcal{D}$ is weaker than $\mathcal{D}'$ in $\mathcal{E}$. We must show that $\mathcal{D}$ is weaker than $\mathcal{FS}^*$ in $\mathcal{E}$. Consider the WSA task. By Theorem 4, WSA is a non-local task. By Theorem 6, $\mathcal{FS}^*$ can be used to solve WSA in $\mathcal{E}$. By "plugging in" $\mathcal{T} = WSA$ and $\mathcal{D}' = \mathcal{FS}^*$ in (*), we get that $\mathcal{D}$ is weaker than $\mathcal{FS}^*$ in $\mathcal{E}$. □

8 Task locality and solving a task without messages

In this section, we establish the equivalence of two alternate definitions of task locality: the one given in Section 2, based on the existence of "local" functions that processes can apply to obtain a correct output, and another one based on the ability to solve a task without messages. We prove this equivalence under a natural assumption on tasks, which, roughly speaking, says that the output of processes is not constrained by failures that may happen *in the future*. We first explain this assumption in Section **??**, and then prove the equivalence of the two alternate definitions of locality in Section **??**.

8.1 Omission-closed tasks

We now define a property that, to the best of our knowledge, is satisfied by all the one-shot input/output tasks that have been proposed and studied to date, including those defined in [2, 14, 16]. Roughly speaking, a task $\mathcal{T}$ is omission-closed if any input/output behavior that satisfies $\mathcal{T}$ *when all processes are correct*, can be modified into a input/output behavior that also satisfies $\mathcal{T}$ *when some processes are faulty*, by simply allowing any subset of the faulty processes to omit their output. In other words, if, for the input vector I, the output vector O satisfies $\mathcal{T}$ when there are *no* failures, then, when there *are* failures, $\mathcal{T}$ is also satisfied by any output vector O' that differs from O only in that some faulty processes may have the "empty" output $\perp$ in O'. Formally, *task $\mathcal{T}$ is omission-closed*, if and only if, for all inputs $I \in \mathcal{I} = \mathcal{I}_1 \times \mathcal{I}_2 \times \cdots \times \mathcal{I}_n$ and all outputs $O \in \mathcal{O} = \mathcal{O}_1 \times \mathcal{O}_2 \times \cdots \times \mathcal{O}_n$ of $\mathcal{T}$, we have:

$$(F_{ff}, I, O) \in \mathcal{T}_S \Rightarrow \forall F \in \mathcal{E}^*, \forall O' \in \mathcal{O} \text{ such that}$$
$$\forall p \in \Pi\big((O'[p] = O[p]) \vee (O'[p] = \perp \wedge p \in \mathit{faulty}(F))\big) : (F, I, O') \in \mathcal{T}_S$$

This property is quite natural: intuitively, it says that the output requirement of processes is not constrained by failures that may occur in the future. To illustrate this, suppose all processes are alive and execute correctly up to a time when they are just about to output some values, and these values satisfy the task in the failure-free case; but immediately after this time, some processes crash just before outputing, while the other processes output as planned. If the task is omission-closed, these outputs also satisfy the task.

8.2 Equivalence of two notions of locality

An alternate definition of task locality is based on the ability to solve a task without messages. Roughly speaking, we could say that a task is local if it can be solved by an algorithm $\mathcal{A}$ and a failure detector $\mathcal{D}$ such that in *all* the failure-free runs of $\mathcal{A}$ using $\mathcal{D}$ every process outputs a value before receiving any message. For omission-closed tasks, this definition of locality is equivalent to the one that we gave in Section 2 using local functions:

Theorem 15 *Let $\mathcal{T}$ be any omission-closed task, and $\mathcal{E}$ be any environment that contains the failure-free failure pattern F_{ff}. $\mathcal{T}$ is a local task if and only if there is an algorithm $\mathcal{A}$ and a failure detector $\mathcal{D}$ such that:*

1. *$\mathcal{A}$ solves $\mathcal{T}$ using $\mathcal{D}$ in $\mathcal{E}$, and*
2. *for every failure detector history $H \in \mathcal{D}(F_{ff})$, every input $I \in \mathcal{I}_1 \times \mathcal{I}_2 \times \cdots \times \mathcal{I}_n$, and every run $R_I = (F_{ff}, H, I, -, -)$ of $\mathcal{A}$ using $\mathcal{D}$ in $\mathcal{E}$, every process outputs a value before receiving any message.*

PROOF. Let $\mathcal{T}$ be any omission-closed task, and let $\mathcal{I}_p$ and $\mathcal{O}_p$ be the corresponding sets of possible input and output values for each process p, and $\mathcal{T}_S$ be the corresponding set of acceptable (F, I, O) tuples. Let $\mathcal{E}$ be any environment that contains F_{ff}.

Suppose task $\mathcal{T}$ is local. Then there are functions $f_1, f_2, \ldots, f_n$ such that for each possible input $I = (i_1, i_2, \ldots, i_n) \in \mathcal{I}_1 \times \mathcal{I}_2 \times \cdots \times \mathcal{I}_n$ of $\mathcal{T}$, the output $O = (f_1(i_1), f_2(i_2), \ldots, f_n(i_n))$ satisfies the specification of $\mathcal{T}$ when all the processes are correct, i.e., $(F_{ff}, I, O) \in \mathcal{T}_S$. Consider the following trivial algorithm $\mathcal{A}$ (which does not require the help of failure detectors): for any local input $i_p \in \mathcal{I}_p$, each process p outputs $f_p(i_p)$. From the properties of the functions f_p's above, and the assumption that $\mathcal{T}$ is omission-closed, it is clear that $\mathcal{A}$ solves $\mathcal{T}$ in $\mathcal{E}$ (using any trivial failure detector $\mathcal{D}$). To see this, note that $\forall I = (i_1, i_2, \ldots, i_n) \in \mathcal{I}_1 \times \mathcal{I}_2 \times \cdots \times \mathcal{I}_n$ of $\mathcal{T}$, $\forall F \in \mathcal{E}$, $(F, I, O') \in \mathcal{T}_S$, where $\forall p \in \Pi\big((O'[p] = f_p(i_p)) \vee (O'[p] = \perp \wedge p \in \mathit{faulty}(F))\big)$. So $\mathcal{A}$ and any trivial failure detector $\mathcal{D}$ satisfy conditions (1) and (2) of Theorem **??**.

Now suppose there exists an algorithm $\mathcal{A}$ and a failure detector $\mathcal{D}$ that satisfies conditions (1) and (2) of Theorem **??**. Let H be any failure detector history in $\mathcal{D}(F_{ff})$. Conditions (1) and (2) of Theorem **??** trivially imply conditions (1) and (2) of Lemma 2. So by Lemma 2, task $\mathcal{T}$ is local. □

9 Comparing $\mathcal{FS}^*$ to anti-Ω and $\mathcal{L}$ in message-passing systems

The following theorem compares $\mathcal{FS}^*$ to two closely related failure detectors, namely, $\mathcal{L}$ and anti-Ω, which are the weakest failure detectors to solve set agreement in message-passing and shared memory systems, respectively. Intuitively, failure detector $\mathcal{L}$ outputs GREEN or RED at each process such that (1) it outputs GREEN forever at some process, and (2) if exactly one process is correct, then there is a time after which $\mathcal{L}$ outputs RED at this process. Anti-Ω outputs a process id at each process such that, if there is a correct process, then there is a correct process c and a time after which anti-Ω never outputs c at any correct process. The proof of the following result is in the optional Appendix D.

Theorem 16 *Anti-Ω is strictly weaker than $\mathcal{FS}^*$ in $\mathcal{E}^*$, and $\mathcal{FS}^*$ is strictly weaker than $\mathcal{L}$ in $\mathcal{E}^*$.*[6]

10 Related Work

Failure detectors have been used to capture the minimum information about failures that is necessary to solve some basic problems in message-passing and shared-memory systems (e.g., [4, 3, 1, 13, 6, 7, 11, 18, 8]). Recently, failure detectors were also used to investigate the minimum information about failures that is necessary (but not necessarily sufficient) to solve some interesting *sets* of problems.

In particular, Guerraoui et al. consider the set of wait-free tasks $\mathcal{S}_{\text{wf}}$ that cannot be solved in asynchronous shared-memory systems with failures [10]. They introduce a failure detector denoted Υ and prove that (1) among the set of *"eventually stable failure detectors"*, Υ is necessary to solve any task in $\mathcal{S}_{\text{wf}}$, and (2) Υ is sufficient to solve set agreement. Zielinski generalizes this result in [18]: he introduces the failure detector anti-Ω and proves that (1) anti-Ω is necessary to solve any task in $\mathcal{S}_{\text{wf}}$, and (2) anti-Ω is the weakest failure detector to solve set agreement. In contrast to the results in [10, 18] which are in shared-memory systems, in [17] Zielinsky considers message-passing systems, and proves that anti-Ω is the weakest failure detector among the set of *eventual* failure detectors that are not implementable in such systems.

Delporte et al. also consider message-passing systems in [8]: they introduce failure detector $\mathcal{L}$ and show that it is the weakest failure detector to solve set agreement in such systems. The failure detector $\mathcal{FS}^*$ introduced here is a simple weakening of $\mathcal{L}$.

Our definition of weak set agreement was obtained by taking a well-known problem, namely set agreement, and weakening one of its property *in the case of failures*. This method of weakening a task was already proposed in the early 80's to obtain weaker versions of some classical problems such as consensus and reliable broadcast [9, 12]. For example, the validity property of consensus (which requires that any decision

[6] We say that $\mathcal{D}$ is strictly weaker than $\mathcal{D}'$ in $\mathcal{E}^*$, if $\mathcal{D}$ is weaker than $\mathcal{D}'$ in $\mathcal{E}^*$ but $\mathcal{D}'$ is not weaker than $\mathcal{D}$ in $\mathcal{E}^*$ (recall that $\mathcal{E}^*$ is the environment where any process can fail at any time).

value must be a proposed value) can be weakened to "if there are no failures, then the validity property must hold" (this property is called *weak unanimity* in [9]). The specification of such tasks can be captured by our definition of a task because it includes the failure pattern (to the best of our knowledge, such definition of a task first appeared in [15]). This is a generalization of the definitions of a task given in [2, 14, 16] which are based solely on the input/output requirement in *failure-free runs*.

Acknowledgement

We are grateful to Marcos K. Aguilera for valuable discussions on this work.

References

[1] Marcos K. Aguilera, Sam Toueg, and Boris Deianov. Revisiting the weakest failure detector for uniform reliable broadcast. In *DISC '99: Proceedings of the thirteenth International Symposium on Distributed Computing*, volume 1693 of *LNCS*, pages 13–33. Springer, September 1999.

[2] Ofer Biran, Shlomo Moran, and Shmuel Zaks. A combinatorial characterization of the distributed 1-solvable tasks. *J. Algorithms*, 11(3):420–440, 1990.

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

[4] Tushar D. Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, March 1996.

[5] Soma Chaudhuri. Agreement is harder than consensus: set consensus problems in totally asynchronous systems. In *PODC '90: Proceedings of the ninth annual ACM symposium on Principles of Distributed Computing*, pages 311–324. ACM Press, August 1990.

[6] 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 *PODC '04: Proceedings of the twenty-third annual ACM symposium on Principles of Distributed Computing*, pages 338–346. ACM Press, July 2004.

[7] Carole Delporte-Gallet, Hugues Fauconnier, Rachid Guerraoui, and Petr Kouznetsov. Mutual exclusion in asynchronous systems with failure detectors. *Journal of Parallel and Distributed Computing*, 65(4):492–505, April 2005.

[8] Carole Delporte-Gallet, Hugues Fauconnier, Rachid Guerraoui, and Andreas Tielmann. The weakest failure detector for message passing set-agreement. In *DISC '08: Proceedings of the twenty-second International Symposium on Distributed Computing*, volume 5218 of *LNCS*, pages 109–120. Springer, September 2008.

[9] Michael J. Fischer. The consensus problem in unreliable distributed systems (a brief survey). In *Proceedings of the 1983 International FCT-Conference on Fundamentals of Computation Theory*, volume 158 of *LNCS*, pages 127–140. Springer, 1983.

[10] Rachid Guerraoui, Maurice Herlihy, Petr Kouznetsov, Nancy A. Lynch, and Calvin C. Newport. On the weakest failure detector ever. In *PODC '07: Proceedings of the twenty-sixth Annual ACM Symposium on Principles of Distributed Computing*, pages 235–243. ACM Press, August 2007.

[11] Rachid Guerraoui, Michal Kapalka, and Petr Kouznetsov. The weakest failure detectors to boost obstruction-freedom. In *DISC '06: Proceedings of the twentieth International Symposium on Distributed Computing*, volume 4167 of *LNCS*, pages 399–412. Springer, September 2006.

[12] Vassos Hadzilacos. On the relationship between the atomic commitment and consensus problems. In *Fault-Tolerant Distributed Computing*, volume 448 of *LNCS*, pages 201–208. Springer, 1986.

[13] Joseph Y. Halpern and Aleta Ricciardi. A knowledge-theoretic analysis of uniform distributed coordination and failure detectors. In *PODC '99: Proceedings of the eighteenth annual ACM symposium on Principles of Distributed Computing*, pages 73–82. ACM Press, May 1999.

[14] Maurice Herlihy and Nir Shavit. The topological structure of asynchronous computability. *Journal of the ACM*, 46(6):858–923, November 1999.

[15] Prasad Jayanti and Sam Toueg. Every problem has a weakest failure detector. In *PODC '08: Proceedings of the twenty-seventh annual ACM Symposium on Principles of Distributed Computing*, pages 75–84. ACM Press, August 2008.

[16] Michael E. Saks and Fotios Zaharoglou. Wait-free k-set agreement is impossible: The topology of public knowledge. *SIAM J. Comput.*, 29(5):1449–1483, 2000.

[17] Piotr Zielinski. Automatic classification of eventual failure detectors. In *DISC '07: Proceedings of the twenty-first International Symposium on Distributed Computing*, volume 4731 of *LNCS*, pages 465–479. Springer, september 2007.

[18] Piotr Zielinski. Anti-Omega: the weakest failure detector for set agreement. In *PODC '08: Proceedings of the twenty-seventh annual ACM Symposium on Principles of Distributed Computing*, pages 55–64. ACM Press, August 2008.

Optional Appendix

A $\mathcal{FS}^*$ is non-trivial

Observation 1 *No algorithm implements $\mathcal{FS}^*$ in $\mathcal{E}^*$.*

PROOF. (Sketch) Suppose, for contradiction, that there is an algorithm $\mathcal{A}$ that implements $\mathcal{FS}^*$ in $\mathcal{E}^*$. We now show that $\mathcal{A}$ has a run R that violates the specification of $\mathcal{FS}^*$.

Let R_p be a run of $\mathcal{A}$ such that (1) process p is the only correct process in R_p, and (2) no other process takes any step in R_p. Since $\mathcal{A}$ satisfies the specification of $\mathcal{FS}^*$, there is a time t_p after which p outputs RED in R_p, forever. We now construct a run R of $\mathcal{A}$ where all processes are correct, but all the messages are delayed to a time $t > \max_{p\in\Pi}\{t_p\}$. For each process p, R is indistinguishable from run R_p up to time t_p, and so p outputs RED at time t_p in run R. After time t, we continue run R as follows: all the messages sent, including those that were previously delayed, are eventually received. Note that run R is a valid run of $\mathcal{A}$ where (1) all processes are correct, but (2) every process $p \in \Pi$ outputs RED at least once. Run R of $\mathcal{A}$ violates the specification of $\mathcal{FS}^*$. □

B Weak set agreement is a non-local task

Theorem 4 *WSA is a non-local task.*

PROOF. The proof is simple. Suppose, for contradiction, that WSA is local. Since WSA is local, there are functions $f_1, f_2, \ldots, f_n$ such that for each possible input $I = (i_1, i_2, \ldots, i_n) \in \{1, 2, \ldots, n\}^n$, the output $O = (f_1(i_1), f_2(i_2), \ldots, f_n(i_n))$ satisfies the specification of WSA when all processes are correct, i.e., $(F_{ff}, I, O) \in WSA_S$.

Consider input $I = (1, 2, \ldots, n)$, i.e., every process i, $1 \leq i \leq n$, proposes value i. Let $O = (f_1(1), f_2(2), \ldots, f_n(n))$; this is the output where every process i, $1 \leq i \leq n$, decides value $f_i(i)$. The output O must satisfy the specification of WSA when all processes are correct, i.e, $(F_{ff}, I, O) \in WSA_S$. Thus, by the validity property of WSA, for all i, $1 \leq i \leq n$, $f_i(i) \in \{1, 2, \ldots, n\}$. Furthermore, by the weak agreement property of WSA, there must be a j and a k such that $f_j(j) = k \neq j$.

Let $I' = (j, j, \ldots, j)$, i.e., every process proposes j. The output $O' = (f_1(j), f_2(j), \ldots, f_j(j), \ldots f_n(j))$ must satisfy the specification of WSA when all processes are correct, i.e, $(F_{ff}, I', O') \in WSA_S$. But process j outputs (i.e., decides) $f_j(j) = k$, a value that was not proposed by any process. This violates the validity property of WSA, so $(F_{ff}, I', O') \notin WSA_S$ — a contradiction that concludes the proof. □

C $\mathcal{FS}^*$ solves the weak set agreement task

The algorithm in Figure 2 solves the weak set agreement task using $\mathcal{FS}^*$. It is identical to the one that solves the set agreement task using $\mathcal{L}$ given in [8]. In this algorithm, lines 3-5 and lines 6-8 are executed atomically. The proof of correctness is given in the optional Appendix.

Theorem 6 *For every environment $\mathcal{E}$, the algorithm in Figure 2 solves WSA using $\mathcal{FS}^*$ in $\mathcal{E}$.*

PROOF. Consider an arbitrary run R of the algorithm using $\mathcal{FS}^*$ in some environment $\mathcal{E}$, and let F be the failure pattern of R (so, when we say "p is correct", we mean $p \in correct(F)$). We show that run R satisfies the three properties of WSA.

Validity. This property holds because each process decides on its own proposal value (line 8), or a proposal value sent by another process (line 5).

CODE FOR PROCESS p:

```
1  to propose(v_p)                                   { v_p is p's proposal value }
2      send v_p to every process q > p
   { p decides a value as follows: }
3      upon receipt of a value v do
4          send v to all
5          decide v ; halt
6      upon FS*_p = RED do
7          send v_p to all
8          decide v_p ; halt
```

Figure 2: Using $\mathcal{FS}^*$ to solve WSA in any environment $\mathcal{E}$.

Termination. There are two possible cases:

(1) There is exactly one correct process p in run R. Then, by the specification of $\mathcal{FS}^*$, there a time after which the failure detector module $\mathcal{FS}^*_p$ outputs RED at p. So either p decides in line 5, or it decides according to lines 6-8.

(2) There are at least two correct processes in run R. Let p and q be two correct processes such that $q > p$. Note that p sends v_p to q in line 2. So either q decides in line 8, or it decides according to lines 3-5. Therefore q sends its decision value to *all* processes in line 4 or in line 7. So every correct process that is still undecided receives this value in line 3 and decides in line 5.

Thus, in both cases, all correct processes decide.

Weak agreement. If processes propose at most $n-1$ different values in R, then by the validity property (shown above), there are at most $n-1$ different decision values, and so weak agreement holds. So assume that there are n distinct proposed values in R, and let v_p denote the value proposed by process p in R.

Note that if there is a process that is not correct in R, or there is a process that never decides in R, then the weak agreement property is trivially satisfied in R. Henceforth, we assume that all the processes are correct and they all decide in run R.

Since all processes are correct in R, there is a process r such that the failure detector module $\mathcal{FS}^*_r$ never outputs RED at r. Thus r does not decide in line 8, and so it decides in line 5. Let p be the process with the largest id that decides in line 5. So every process $q > p$ decides in line 8.

We claim that no process decides v_p. To see this, note that:

(1) No process $q > p$ decides v_p. This is because each process $q > p$ decides in line 8, and so it decides $v_q \neq v_p$.

(2) No process $q > p$ sends v_p. This follows from (1).

(3) No process $s \leq p$ receives v_p. This follows from the fact that p initially sends v_p only to processes $q > p$, and, by (2), these processes do not relay v_p to any process. So v_p is not sent to any process $s \leq p$.

(4) Process p does not decide v_p. This is because p decides in line 5 the value that it received in line 3, and, by (3), it does not receive v_p.

CODE FOR PROCESS p

Local Variables:

out-anti-$\Omega_p = p$ { variable that emulates the output of anti-Ω_p }
$\forall q \in \Pi\ heartbeat[q] = 0$
$RedSet = \emptyset$

Main Code: Task1 || Task2

Task1:

```
1   upon receipt of (RedSet', heartbeat')
2       for q from 1 to n do
3           heartbeat[q] ← max(heartbeat[q], heartbeat'[q])
4       RedSet ← RedSet ∪ RedSet'
```

Task2:

```
5   repeat forever
6       heartbeat[p] ← heartbeat[p] + 1
7       if (FS*_p = RED)
8           then RedSet ← RedSet ∪ {p}
9       send (RedSet, heartbeat) to all
10      if RedSet ≠ Π
11          then out-anti-Ω_p ← smallest process in Π − RedSet
12          else out-anti-Ω_p ←process q with the smallest heartbeat[q] { at least one process is faulty }
```

Figure 3: Transformation algorithm $\mathcal{T}_{\mathcal{FS}^* \rightarrow \text{anti-}\Omega}$.

(5) No process $s < p$ decides v_p. This is because either such a process s decides $v_s \neq v_p$ in line 8, or it decides in line 5 the value that it received in line 3, and, by (3), it does not receive v_p.

By (1), (4), and (5) no process decides v_p. By the validity property (shown above), this implies that there are at most $n-1$ distinct decision values. □

D Comparing $\mathcal{FS}^*$ to anti-Ω and $\mathcal{L}$ in message-passing systems

In this section we compare $\mathcal{FS}^*$ to anti-Ω and $\mathcal{L}$. We say that:

- *$\mathcal{D}$ is weaker than $\mathcal{D}'$ in $\mathcal{E}$*, if there is an algorithm $\mathcal{T}_{\mathcal{D}' \rightarrow \mathcal{D}}$ that transforms $\mathcal{D}'$ to $\mathcal{D}$ in $\mathcal{E}$.
- *$\mathcal{D}$ is strictly weaker than $\mathcal{D}'$ in $\mathcal{E}$*, if $\mathcal{D}$ is weaker than $\mathcal{D}'$ but $\mathcal{D}'$ is not weaker than $\mathcal{D}$ in $\mathcal{E}$.

D.1 Anti-Ω is strictly weaker than $\mathcal{FS}^*$

The failure detector anti-Ω [18] outputs a process id at every process. If there is at least one correct process, then there is a correct process c and a time after which anti-Ω does not output c at any correct process. Formally, for every failure pattern $F \in \mathcal{E}^*$:

$$\text{anti-}\Omega(F) = \big\{H \mid (\forall p \in \Pi, \forall t \in \mathbb{N}:\ H(p,t) \in \Pi) \wedge \\ correct(F) \neq \emptyset \Rightarrow \exists c \in correct(F), \exists t \in \mathbb{N}, \forall p \in correct(F), \forall t' \geq t:\ H(p,t') \neq c)\big\}$$

The algorithm in Figure 3 transforms $\mathcal{FS}^*$ to anti-Ω in $\mathcal{E}$; intuitively, it works as follows. Each process p maintains two variables: (1) $heartbeat$, an array containing the number of heartbeats of each process, and (2) $RedSet$, the set of processes q such that $\mathcal{FS}^*$ output RED at some time at q. Processes communicate these variables between themselves and update them accordingly: they merge the $RedSet$s and take the maximum of the number of heartbeats for each process. Each process p emulates the ouput of anti-Ω_p as follow. When p's $RedSet$ is not Π, p outputs the smallest process in $\Pi - RedSet$. When p's $RedSet$ is Π, p outputs the process with the smallest number of heartbeats (if several processes have the same smallest number of heartbeats, p outputs the process with the smallest id among this set).

Intuitively, this algorithm correctly emulates the output of failure detector anti-Ω for the following reasons. At each process, the set $RedSet$ is non-decreasing, and since it is bounded by Π, eventually it stops changing. Since correct processes keep exchanging and merging their $RedSet$s, there is a time after which all the correct processes have the *same* set $RedSet$. After this occurs, there are two possible cases:

$RedSet \neq \Pi$. In this case, every correct process outputs the smallest process q in $\Pi - RedSet$. Note that q cannot be the only correct process in the run: if it was, there would be a time after which $\mathcal{FS}^*$ outputs RED at q forever, so q would be in $RedSet$. So there is a correct process process $c \neq q$ and a time after which correct processes do not output c. This correctly emulates the output of anti-Ω.

$RedSet = \Pi$. In this case, every correct process p outputs the process q with the smallest $heartbeat_p[q]$. Note that some process must be faulty: if all processes were correct, $\mathcal{FS}^*$ would output GREEN forever at some process q, so $RedSet$ would not contain q. Since the number of hearbeats of every faulty process stops growing, and the number of hearbeats of every correct process keeps increasing, there is a time after which, at every correct process p, the process q with the smallest $heartbeat_p[q]$ is faulty. So there is a time after which correct processes output only faulty processes. This also correctly emulates the output of anti-Ω.

Lemma 11 *For every environment $\mathcal{E}$, anti-Ω is weaker than $\mathcal{FS}^*$ in $\mathcal{E}$.*

PROOF. Let $\mathcal{E}$ be any environment. We claim that the algorithm $\mathcal{T}_{\mathcal{FS}^* \rightarrow \text{anti-}\Omega}$ shown in Figure 3 transforms $\mathcal{FS}^*$ to anti-Ω in $\mathcal{E}$. To prove this, consider an *arbitrary* run R of algorithm $\mathcal{T}_{\mathcal{FS}^* \rightarrow \text{anti-}\Omega}$ using $\mathcal{FS}^*$ in $\mathcal{E}$. Let F be the failure pattern of R and $H \in \mathcal{FS}^*(F)$ be the failure detector history of R. We must show that, in run R, the local variables out-anti-Ω_p, which are maintained by $\mathcal{T}_{\mathcal{FS}^* \rightarrow \text{anti-}\Omega}$ emulate the output of failure detector anti-Ω.

We first note that if $correct(F) = \emptyset$ then the property of anti-Ω is trivially satisfied. Henceforth we assume that $correct(F) \neq \emptyset$, i.e., there is at least one correct process in run R.

From the way $RedSet$ is set in lines 4 and 8, $RedSet$ is non-decreasing and bounded by Π, i.e., for every process p and all times $t < t'$, $RedSet_p^t \subseteq RedSet_p^{t'}$ and $RedSet_p^t \subseteq \Pi$. Thus, for every processes p, there is a set of processes X_p and a time after which $RedSet_p = X_p$. So, since correct processes keep exchanging and merging their $RedSet$s forever, there is a set of processes X and a time after which $RedSet_p = X$ at every correct process p. There are two possible cases:

Case 1: $X \neq \Pi$. In this case, there is a time after which every correct process p has out-anti-$\Omega_p = q$ where q is the smallest process in $\Pi - X$. We claim that there is some correct process $c \neq q$ in run R. To see this, note that (a) if q is faulty, then this follows from our assumption that there is at least one correct process in run R, and (b) if q is correct, then it cannot be the only correct process in run R: it it was, then by the first property of $\mathcal{FS}^*$, there would be a time after which $\mathcal{FS}_q^* = \text{RED}$, and since q is correct, the set X would contain q (contradicting the fact that $q \in \Pi - X$). From the claim, there is a time after which every correct process p has out-anti-$\Omega_p \neq c$.

Case 2: $X = \Pi$. We first note that there is at least one faulty process in run R. To see this, suppose for contradiction that all the processes are correct in R. In this case, by the first property of $\mathcal{FS}^*$, there is

a correct process q such that $\mathcal{FS}^*$ outputs GREEN at q forever. So q is never added to $RedSet_q$, and this implies that $q \notin X$ — a contradiction to the assumption that $X = \Pi$.

Note that faulty processes eventually stop increasing their heartbeats while correct processes keep increasing them forever. Thus, from the way processes maintain their $heartbeat$ arrays, it is clear that there is a time after which for every correct process p, every faulty process f and every correct process c, $heartbeat_p[f] < heartbeat_p[c]$.

Since $X = \Pi$, there is a time after which every correct process p has out-anti-$\Omega_p = q$, where q is the process with the smallest $(heartbeat_p[q], q)$; from the above, it is clear that there is a time after which q is faulty. So, since there is at least one correct process in run R, there is a correct process c and a time after which every correct process p has out-anti-$\Omega_p \neq c$.

Thus, in both cases, the variables out-anti-Ω correctly emulate the output of anti-Ω. □

Lemma 12 *$\mathcal{FS}^*$ is not weaker than anti-Ω in $\mathcal{E}^*$.*

PROOF. (Sketch) Suppose, for contradiction, that there is an algorithm $\mathcal{T}_{\text{anti-}\Omega \rightarrow \mathcal{FS}^*}$ that transforms anti-Ω to $\mathcal{FS}^*$ in $\mathcal{E}^*$. We now show that this algorithm is not correct, i.e., it has a run R in $\mathcal{E}^*$ that does not emulate the output of $\mathcal{FS}^*$ correctly.

For every $p \in \Pi$, let R_p be a run of $\mathcal{T}_{\text{anti-}\Omega \rightarrow \mathcal{FS}^*}$ in $\mathcal{E}^*$ such that (1) process p is the only correct process in R_p, and (2) no other process takes any step in R_p, and (3) at all processes, anti-Ω outputs the smallest process in the set $\Pi - \{p\}$ (note that this output satisfies the specification of anti-Ω since process p is correct in run R_p). Since $\mathcal{T}_{\text{anti-}\Omega \rightarrow \mathcal{FS}^*}$ emulates the output of $\mathcal{FS}^*$, there is a time t_p after which p outputs RED in R_p, i.e., out-$\mathcal{FS}^*_p = \text{RED}$.

We now construct the "bad" run R of $\mathcal{T}_{\text{anti-}\Omega \rightarrow \mathcal{FS}^*}$ in $\mathcal{E}^*$ as follows. In run R, (1) all processes are correct, (2) all the messages are delayed to a time $t > \max_{p \in \Pi}\{t_p\}$, (3) each process p takes steps at the same times as in run R_p, and (4) at each process p, anti-Ω outputs the smallest process in the set $\Pi - \{p\}$. For every process p, run R is indistinguishable from run R_p up to time t_p, and so p has out-$\mathcal{FS}^*_p = \text{RED}$ at time t_p in run R.

To ensure that R is a valid run of $\mathcal{T}_{\text{anti-}\Omega \rightarrow \mathcal{FS}^*}$ in $\mathcal{E}^*$, we continue run R from time t as follows: (1) all the messages sent, including those that were previously delayed, are eventually received, and (2) anti-Ω outputs process 1 at all processes (note that this output satisfies the specification of anti-Ω because all the processes are correct in R). Thus, R is a valid run of $\mathcal{T}_{\text{anti-}\Omega \rightarrow \mathcal{FS}^*}$ in $\mathcal{E}^*$ where (1) all processes are correct, but (2) every process $p \in \Pi$ has out-$\mathcal{FS}^*_p = \text{RED}$ at least once. Note that this emulated output of $\mathcal{FS}^*$ does not satisfy the specification of $\mathcal{FS}^*$ — a contradiction that concludes the proof. □

D.2 $\mathcal{FS}^*$ is strictly weaker than $\mathcal{L}$

The failure detector $\mathcal{L}$ [8] outputs RED and GREEN at each process, such that: (a) $\mathcal{FS}^*$ outputs GREEN forever at some process, and (b) if *exactly one* process is correct, then there is a time after which $\mathcal{FS}^*$ outputs RED at this process.[7] Formally, for every failure pattern $F \in \mathcal{E}^*$:

$$\begin{aligned}\mathcal{L}(\mathcal{F}) = \big\{H \mid & (\forall p \in \Pi,\, \forall t \in \mathbb{N}:\ H(p,t) = \text{GREEN} \vee H(p,t) = \text{RED}) \wedge \\ & (\exists p \in \Pi,\, \forall t \in \mathbb{N}:\ H(p,t) = \text{GREEN}) \wedge \\ & (|correct(F)| = 1 \Rightarrow \exists p \in correct(F),\, \exists t \in \mathbb{N},\, \forall t' \geq t:\ H(p,t') = \text{RED})\big\}\end{aligned}$$

Lemma 13 *For every environment $\mathcal{E}$, $\mathcal{FS}^*$ is weaker than $\mathcal{L}$ in $\mathcal{E}$.*

[7] In [8], the outputs GREEN and RED were actually called $false$ or $true$, respectively.

PROOF. This is obvious from the definitions of $\mathcal{L}$ and $\mathcal{FS}^*$: for every failure pattern $F \in \mathcal{E}$, $\mathcal{L}(\mathcal{F}) \subseteq \mathcal{FS}^*(F)$. □

Lemma 14 *$\mathcal{L}$ is not weaker than $\mathcal{FS}^*$ in $\mathcal{E}^*$.*

PROOF. (Sketch) Suppose, for contradiction, that there is an algorithm $\mathcal{T}_{\mathcal{FS}^* \rightarrow \mathcal{L}}$ that transforms $\mathcal{FS}^*$ to $\mathcal{L}$ in $\mathcal{E}^*$. We now show that this algorithm is not correct, i.e., it has a run R in $\mathcal{E}^*$ that does not emulate the output of $\mathcal{L}$ correctly.

For every $p \in \Pi$, let R_p be a run of $\mathcal{T}_{\mathcal{FS}^* \rightarrow \mathcal{L}}$ in $\mathcal{E}^*$ such that (1) process p is the only correct process in R_p, and (2) no other process takes any step in R_p, and (3) $\mathcal{FS}^*_p$ always outputs RED at p (note that this output satisfies the specification of $\mathcal{FS}^*$ since process p is the only correct in run R_p). Since $\mathcal{T}_{\mathcal{FS}^* \rightarrow \mathcal{L}}$ emulates the output of $\mathcal{L}$, there is a time t_p after which p outputs RED in R_p, i.e., $out\text{-}\mathcal{L}_p = \text{RED}$.

We now construct the "bad" run R of $\mathcal{T}_{\mathcal{FS}^* \rightarrow \mathcal{L}}$ in $\mathcal{E}^*$ as follows. In run R, (1) all processes are correct except for process 1 which crashes at time $\max_{p \in \Pi}\{t_p\}$+1, (2) all the messages are delayed to a time $t > \max_{p \in \Pi}\{t_p\}$, (3) up to time t_p, each process p takes steps at the same times as in run R_p, and (4) $\mathcal{FS}^*$ outputs RED forever at every process (note that this output satisfies the specification of $\mathcal{FS}^*$ since not all processes are correct in run R). For every process p, run R is indistinguishable from run R_p up to time t_p, and so p has $out\text{-}\mathcal{L}_p = \text{RED}$ at time t_p in run R.

To ensure that R is a valid run of $\mathcal{T}_{\mathcal{FS}^* \rightarrow \mathcal{L}}$ in $\mathcal{E}^*$, we continue run R from time t as follows: all the messages sent to correct processes, including those that were previously delayed, are eventually received. Thus, R is a valid run of $\mathcal{T}_{\mathcal{FS}^* \rightarrow \mathcal{L}}$ in $\mathcal{E}^*$ where (1) all processes except process 1 are correct, but (2) every process $p \in \Pi$ has $out\text{-}\mathcal{L}_p = \text{RED}$ at least once. Note that this emulated output of $\mathcal{L}$ does not satisfy the specification of $\mathcal{L}$ — a contradiction that concludes the proof. □

D.3 Comparing $\mathcal{FS}^*$ to anti-Ω and $\mathcal{L}$

We can now conclude:

Theorem 10 *Anti-Ω is strictly weaker than $\mathcal{FS}^*$ in $\mathcal{E}^*$, and $\mathcal{FS}^*$ is strictly weaker than $\mathcal{L}$ in $\mathcal{E}^*$.*

PROOF. From Lemmas 11 and 12, we get that anti-Ω is strictly weaker than $\mathcal{FS}^*$ in $\mathcal{E}^*$. From Lemmas 13 and 14, we get that $\mathcal{FS}^*$ is strictly weaker than $\mathcal{L}$ in $\mathcal{E}^*$. □