Perfect Failure Detection with Very Few Bits*

Pierre Fraigniaud[1], Sergio Rajsbaum**[2], Corentin Travers***[3], Petr Kuznetsov[4], and Thibault Rieutord[4]

[1] IRIF and U. Paris Diderot, France
[2] Instituto de Matemáticas, UNAM, Mexico
[3] LaBRI, U. Bordeaux, France
[4] Télécom ParisTech, France

Abstract. A *failure detector* is a distributed oracle that provides each process with a module that continuously outputs an estimate of which processes in the system have failed. The *perfect* failure detector provides accurate and eventually complete information about process failures. We show that, in asynchronous failure-prone message-passing systems, perfect failure detection can be achieved by an oracle that outputs at most $\lceil \log \alpha(n) \rceil + 1$ bits per process in n-process systems, where α denotes the inverse-Ackermann function. This result is essentially optimal, as we also show that, in the same environment, no failure detector outputting a constant number of bits per process can achieve perfect failure detection.

Keywords: failure detectors, well-quasi-order, Higman's lemma

1 Introduction

Failure detectors have influenced research and development of fault-tolerant distributed systems for over 20 years, since their introduction in two seminal papers [2,3]. A *failure detector* is an abstraction layer that provides each process with information about which other processes have crashed. The concept of failure detector provides a modular approach of distributed computing and an elegant framework which yields two orthogonal but interacting working projects: developing portable algorithms on top of failure detectors, and developing efficient failure detector implementations in various message passing and shared memory settings. This concept has been very successful in a wide variety of settings, including network communication protocols, group membership protocols, and algorithms for solving consensus, atomic commit, broadcast, mutual exclusion, leader election, as well as several other services (see Section 1.2). More generally,

* This research has been carried out within the framework of ECOS Nord (Project M12M01).
** Additional support from PAPIIT-UNAM grant IN107714.
*** Additional support from the French State, managed by the French National Research Agency (ANR) in the frame of the "Investments for the future" Programme IdEx Bordeaux - CPU (ANR-10-IDEX-03-02).

the failure detector abstraction has fostered the theoretical understanding of failures, and of their effects in distributed computing. Indeed, failure detectors abstract away details of the system (e.g., the message delivery times on each link, the process speeds, etc.), by focussing only on extracting process failure information. Given a failure detector, one can then investigate what are the distributed computing tasks that are solvable with the information provided by this failure detector layer (irrespective of the underlying network on top of which the failure detector is implemented).

In a nutshell, failure detectors provide a formal framework to tackle questions such as: How much, and what kind of information about failures is necessary to solve a given distributed computing task?

At the one end of the spectrum, the concern is the *minimum information about failures* needed to solve a given task, e.g., notably, the *consensus* problem. Various *weakest* failure detectors for consensus have been identified, that show that the question of *how much* information about failures is needed to solve a problem is subtle. The weakest failure detectors enabling to solve consensus are all equivalent, that is, given any of these failure detectors, one can build any other such failure detector. Yet, they seem to provide very different kind of information, presented in very different forms. For instance, the failure detector Ω outputs the identity of a single process at each process [2]. This identity is such that, eventually, all the correct processes are provided with the same identity, which is the identity of a correct process. In contrast, the failure detector $\Diamond S$ outputs a set of process identities at each process [3]. These identities are the ones of suspected processes, and are such that, eventually, they include all the processes that have crashed, and there is a correct process whose identity is included in none of the sets. These two failure detectors are equivalent (they both are weakest failure detectors enabling to solve consensus in an asynchronous message-passing system where a majority of processes are correct).

At the other end of the spectrum, the concern is failure detectors that provide *perfectly accurate information about failures*. Remarkably, also at this end of the spectrum, there are failure detectors that provide "the same information" about failures, but they do so in a very different way. This is raising the question of what is the *amount* of information provided by failure detectors. Consider for example the failure detector classes $\mathcal{P}$ and ψ^t. A failure detector $\mathcal{P}$ provides each process with a set of identities of processes that are suspected to have failed [3]. These sets are such that non-faulty processes are never suspected, and all faulty processes are eventually suspected by each process. In a system where at most t processes can crash, a failure detector ψ^t outputs an integer at each process [17]. These integers are such that they are at most the number of processes that have crashed, and, eventually, they are all equal to the number of processes that have actually crashed. While $\mathcal{P}$ and ψ^t are quite similar qualitatively as they both provide perfectly accurate information about failures, they are quantitatively quite different: one provides sets of identities, while the other provides integers in a bounded range of values.

In this paper, we initiate the study of how many bits should be provided to each process by a failure detector to ensure specific knowledge about the failure pattern, or to solve a given task. We start our investigation by tackling this question at the latter end of the spectrum, namely, the case of a failure detector that guarantees perfectly accurate information about failures. Specifically, in this paper, we tackle the following question: *how many bits should be provided to each process by a failure detector that guarantees perfectly accurate information about failures?*

1.1 Contributions

We describe a new failure detector, called *micro-perfect*, denoted μP, which outputs at most $\lceil \log \alpha(n) \rceil + 1$ bits at each process, in asynchronous failure-prone message-passing systems with n processes, where α denotes the inverse-Ackermann function. We show that μP is equivalent to the perfect failure detector. This result is essentially optimal, as we also show that, in asynchronous failure-prone message-passing systems, no failure detector outputting a constant number of bits at each process can achieve perfect failure detection.

For establishing both our lower and upper bounds, we use techniques from well-quasi-ordering theory [13]. This important tool in logic and computability has a wide variety of applications [15]. Here we proceed to explore the depth of the connection of well-quasi-orderings with distributed computing, stemming from an essential difficulty when dealing with processes that may crash. In fault-tolerant computing, when a process considers a list L of local states of other processes, it may well be the case that its view is incomplete, e.g., the actual global state is L' with $L \subset L'$, because it is possible that processes in $L' \setminus L$ are delayed. This bares resemblance to well-quasi-ordering theory, which studies words over alphabets, and the sub-words that can be obtained by deleting some symbols of each word. We show that fault-tolerant computing does not only bare resemblance to well-quasi-ordering theory, but that well-quasi-ordering theory is inherently present in some aspects of fault-tolerant computing.

More specifically, for the lower bound, a key ingredient is Higman's lemma [11], which essentially says that if $w^{(1)}, w^{(2)}, \ldots$ is an infinite sequence of words over some *finite* alphabet Σ, then there exist indices $i < j$ such that $w^{(i)}$ can be obtained from $w^{(j)}$ by deleting some of its letters. We show how to use Higman's lemma to prove that no failure detector outputting a constant number of bits at each process can achieve perfect failure detection.

For the upper bound, i.e., for the design of the micro-perfect failure detector μP, we use a combination of failure detector techniques, with the notion of *distributed encoding* of the integers recently introduced in [6]. A distributed encoding of the integers is a distributed structure that encodes each positive integer n by a word $w^{(n)} = w_1^{(n)}, \ldots, w_n^{(n)}$ over some (non-necessarily finite) alphabet Σ, such that no proper sub-words of $w^{(n)}$ can be interpreted as the distributed encoding of n' with $n' < n$. In [6], using well-quasi-ordering theory, it is proved that the first n integers can be distributedly encoded using words on an alphabet with letters on $\lceil \log \alpha(n) \rceil + 1$ bits, where α is a function growing at least

as slowly as the inverse-Ackerman function. We explain how to use this encoding to prove that there exists a failure detector μP outputting $\lceil \log \alpha(n) \rceil + 1$ bits at each process, which achieves perfect failure detection. A companion technical report [7] contains the proofs and some additional material.

1.2 Related Work

We refer to [8] for a recent survey on the failure detector abstraction. In this section, we just survey work closely related to our paper.

In [17], two failure detectors are introduced, which output an integer that approximates the number of crashed processes. More precisely, a query to a failure detector of the class ψ^y returns an integer that is always between $t - y$ and the number of processes that crashed during the execution (where t is the maximum of processes that can crash, and $0 \leq y \leq t$). More precisely, for any time τ, the output returned by a query issued at time τ is at most $\max(t - y, f^\tau)$ where f^τ is the number of processes that have crashed at time τ. Furthermore, there is a time τ' from which the output returned by any query issued at any time τ'' after time τ' is equal to $\max(t - y, f^{\tau''})$. The class $\Diamond\psi^y$ relaxes ψ^y by allowing the properties defining ψ^y to be satisfied only eventually. It is proved that the classes ψ^y and $\Diamond\psi^y$ are respectively equivalent[5] to the classes ϕ^y and $\Diamond\phi^y$ of [16]. A failure detector of the class ϕ^y provides the processes with a query primitive which has a set X of processes as parameter, and which returns a boolean answer. When $|X|$ is too small (or too big), the invocation of the query for X by a process returns systematically *true* (resp., $false$). Otherwise, namely, when $t - y < |X| \leq t$, $0 \leq y \leq t$, the query returns *true* only if all the processes in X have crashed. Moreover, if all the processes of X have crashed, and a process repeatedly issues the query, then it eventually obtains the answer *true*. Notice that ϕ^0 provides no information about failures, while ϕ^t is equivalent to a perfect failure detector. In the follow-up paper [17], the relation of the failure detector Ω_z to set agreement is studied, including relations with respect to the failure detector $\Diamond S_x$.

A failure detector class whose output is binary has been introduced in [9] to solve non-blocking atomic commit. This class, called *anonymously perfect failure detectors*, and denoted by $?\mathcal{P}$, is defined as follows. Each process has a flag (initially equal to $false$) that is eventually set to *true* if and only if a process has crashed (the identity of the crashed process is not necessarily known, hence the name "anonymous"). The definition of $?\mathcal{P}$ has been extended in [17] to take into account the fact that k processes have crashed (instead of just one). This class, denoted $?\mathcal{P}k$, provides each process with a flag that is eventually set to *true* if and only if at least k processes have crashed (observe that $?\mathcal{P}$ is $?\mathcal{P}1$). An interesting question raised in [17] is the issue of additivity of failure detectors. It is known that combining two failure detectors may enable solving consensus,

[5] We stress that, in the literature, by "equivalent" it is meant that, given any failure detector of one class, it is possible to build a failure detector of the other class, and it is understood that "both provide the same information on failures" (see, e.g., [17]).

while none of them is individually strong enough to enable solving consensus. In this paper, we aim at quantifying such phenomenon, by considering the number of bits provided to each process by the failure detector.

The notion of *well-quasi-ordering* (wqo) is a "frequently discovered concept", as already pointed out by Kruskal [13] in 1972. One important application of wqo is providing *termination arguments* in decidability results [1]. Indeed, thirteen years after publishing his undecidability result, Turing [21] proposed the now classic method of proving program termination using so-called "bad sequences", with respect to a wqo. In the setting of wqo, the problem of *bounding* the length of bad sequences is of utmost interest as it yields upper bounds on terminating program executions. Hence, the interest in *algorithmic aspects* of wqos has grown recently as witnessed by the amount of work collected in [19]. For more applications and related work on wqos, including rewriting systems, tree embeddings, lossy channel systems, and graph minors, see recent works [10,19]. The notion of distributed encoding of the integers was proposed in [6] to show that every one-shot system specification can be wait-free runtime monitored non-deterministically using only three opinions.

2 The model

Our results are stated in the classical model used for investigating failure detectors. Specifically, we consider an asynchronous crash-prone message-passing system consisting of n processes denoted by $p_1, \ldots, p_n$. Each process p_i has a unique identity $i = \mathsf{id}(p_i)$, and the total number n of processes is known to each of the processes. Each pair of processes $\{p_i, p_j\}$ is connected by a reliable, yet asynchronous channel. That is, any message sent by p_i to p_j is eventually received by p_j but there are no upper bounds on the time to transfer that message. Channels are reliable in the sense that they do not alter, duplicate or create messages. An arbitrary large number of processes can fail, by crashing, as long as at least one process remains correct. When a process crashes, it permanently stops functioning, that is, it does not execute any more steps of computation (including sending and receiving messages).

2.1 Failure detectors

For modeling failure detectors, we assume the existence of a global clock, with non-negative integer values. This clock is however not accessible to the processes. Let $\Pi = \{p_1, \ldots, p_n\}$. A *failure pattern* is a function $\mathcal{F} : \mathbb{N} \to 2^{\Pi}$ that specifies which are the processes that have crashed by time $\tau \in \mathbb{N}$. Let $\mathsf{faulty}(\mathcal{F}) = \bigcup_{\tau \in \mathbb{N}} \mathcal{F}(\tau)$ be the set of processes that fail in the failure pattern $\mathcal{F}$. The set of processes that do not fail is $\mathsf{correct}(\mathcal{F}) = \Pi \setminus \mathsf{faulty}(\mathcal{F})$. When there is no ambiguity on the underlying failure pattern $\mathcal{F}$, we say that a process p_i is *correct* if $p_i \in \mathsf{correct}(\mathcal{F})$ and *faulty* if $p_i \in \mathsf{faulty}(\mathcal{F})$. An *environment* is a set of failure patterns. In this paper, as specified before, all failure patterns in which at least one process is correct can occur.

A failure detector [3] is a distributed device that provides each process with some information on the failure pattern. Each process can *query* the failure detector, and each query returns a value in some (potentially infinite) range $\mathcal{R}$ that depends on the failure detector. The outputs of a failure detector during an execution is described by a failure detector *history*, which is a function $H : \Pi \times \mathbb{N} \to \mathcal{R}$ that maps each pair process-time to a value in $\mathcal{R}$. The value returned by the failure detector to process p_i at time τ is $H(p_i, \tau)$. A failure detector D with range $\mathcal{R}$ associates a non-empty set of histories with range $\mathcal{R}$ to every failure pattern. The set of histories corresponding to a failure pattern $\mathcal{F}$ is denoted by $D(\mathcal{F})$. That is, $D(\mathcal{F})$ is a collection of functions of the form $H : \Pi \times \mathbb{N} \to \mathcal{R}$, and when the failure pattern is $\mathcal{F}$, the behavior of the failure detector coincides with some history $H \in D(\mathcal{F})$. For instance, the failure detector Ω [2] has range $\{0, 1\}$, and guarantees that it eventually outputs 1 at a single correct process, and 0 at every other processes. That is, for every failure pattern $\mathcal{F}$, the history $H : \Pi \times \mathbb{N} \to \{0, 1\} \in \Omega(\mathcal{F})$ if and only if there exists $p_i \in \mathsf{correct}(\mathcal{F})$, and $\tau \in \mathbb{N}$ such that, for every $\tau' \geq \tau$, $H(p_i, \tau') = 1$ and $H(p_j, \tau') = 0$ for every $j \neq i$.

The so-called *perfect* failure detector P [3] provides a list of processes that have crashed to each process. The failure detector P does not make any mistake, in the sense that no process is declared crashed before it has failed. Moreover, it is eventually complete, in the sense that its output at every process eventually matches the set of faulty processes. More formally, the range of P is 2^{Π}, and, for every failure pattern $\mathcal{F}$, the history $H : \Pi \times \mathbb{N} \to 2^{\Pi}$ belongs to $P(\mathcal{F})$ if and only if the following two properties are satisfied: (Accuracy) for every time τ and process p_i, $H(p_i, \tau) \subseteq \mathcal{F}(\tau)$; and (Completeness) there exists a time τ such that, for every $\tau' \geq \tau$ and process p_i, $H(p_i, \tau') = \mathcal{F}(\tau')$.

Similarly, the *eventual perfect* failure detector $\Diamond P$ [3] is identical to the failure detector P except that the accuracy property only holds eventually. Formally, for every failure pattern $\mathcal{F}$, $H : \Pi \times \mathbb{N} \to 2^{\Pi}$ belongs to $\Diamond P(\mathcal{F})$ if and only if there exists $\tau \in \mathbb{N}$ such that, for every $\tau' \geq \tau$ and every process p_i, $H(p_i, \tau') = \mathcal{F}(\tau')$.

2.2 Protocols and executions

A *distributed protocol* $\mathcal{A}$ consists of n local algorithms $\mathcal{A}(p_1), \ldots, \mathcal{A}(p_n)$, one per process. An *execution* is a sequence of *steps*. During a step, every process p_i acts according to its local algorithm. First, it performs some local computation, and then it performs one of the following five actions: (1) sending a message to some process, (2) receiving a (possibly empty) set of messages, (3) querying the failure detector, (4) receiving an external input or, (5) sending an external output. In a reception step performed by process p_i, since the communication channels are asynchronous, the set of messages might be empty even if a message has been previously send to p_i and not yet received by it. Receiving an external input (resp., sending an external output) are actions enabling to specify protocols that implement, or emulate failure detectors. External inputs correspond to queries to the emulated failure detector. As a result of such a query, an external output is eventually sends, which corresponds to the result of the query.

An *execution* of a protocol $\mathcal{A}$ using failure detector D in environment $\mathcal{E}$ is a tuple $exec = (\mathcal{F}, H, S, T)$ where $\mathcal{F}$ is a failure pattern in $\mathcal{E}$, H is a failure detector history in $D(\mathcal{F})$, S is a sequence of steps of $\mathcal{A}$, and T is a strictly increasing sequence of clock ticks in $\mathbb{N}$. S is called a *schedule*, and the ith step $S[i]$ in S is taken at time $T[i]$. A tuple $exec = (\mathcal{F}, H, S, T)$ defines an execution of $\mathcal{A}$ if and only if the following conditions are satisfied: (1) every correct process takes infinitely many steps in S, (2) no processes take a step after they have crashed, (3) the sequence T and S are either both finite with the same length, or are both infinite, (4) no messages are loss, i.e., if a process performs infinitely many receive step, it eventually receives all messages that were sent to it, (5) if the ith step $S[i]$ is a failure detector query by process p_j that returns d, then $d = H(p_j, T[i])$, i.e., the failure detector queries return values that are consistent with the history H, and (6) the steps taken in S are consistent with the protocol $\mathcal{A}$. Formalizing the above conditions is straightforward but requires care and heavy notation. We refer to [4,12] for such a formalization.

2.3 Comparing failure detectors

A protocol $\mathcal{A}$ that implements a failure detector D receives queries as external inputs, and produces responses in the range of D. Since computing a response may entail sending/receiving messages as well as local computations, there might be some delay between the time τ at which $\mathcal{A}$ receives a query, and the time τ' at which $\mathcal{A}$ produces a response d to that query. The correctness condition taken from [12] requires that d must be a legal output for D at some point in time between τ and τ'. Hence the implementation $\mathcal{A}$ of D behaves as an atomic failure detector, for which responses to queries are given instantaneously. More precisely, a protocol $\mathcal{A}$ implements a failure detector D using a failure detector D', or, for short, *emulates* D using D', in environment $\mathcal{E}$ if, for every failure pattern $\mathcal{F} \in \mathcal{E}$, and for every execution $exec = (\mathcal{F}, H', S, T)$ of $\mathcal{A}$ where $H' \in D'(\mathcal{F})$ the following hold. Let H_Q and H_R be the histories of external inputs (queries) and outputs (responses to queries) in execution *exec*. A query occurs at process p_i at time τ if $H_Q(p_i, \tau) = query$. Similarly, a response occurs at time τ at process p_i if $H_R(p_i, \tau) = d$, where d is a value in the range of D. The following three properties must be fulfilled: (1) for every correct process p_i, and every integer $i \geq 1$, if the ith query at process p_i occurs at time τ, then a response occurs at p_i at some time $\tau' > \tau$ (the ith query and the ith response occurring at the same process p_i are said to be *matching*); (2) for every process p_i, and every integer $i \geq 1$, if the ith response occurs at time τ, then the ith query occurs at p_i at some time $\tau' < \tau$; (3) there exists a failure detector history $H \in D(\mathcal{F})$ such that, for every process p_i, and every times τ_1, τ_2, if $H_Q(p_i, \tau_1) = query$, $H_R(p_i, \tau_2) = d$, and this query/response pair is matching, then $d = H(p_i, \tau)$ for some time $\tau \in [\tau_1, \tau_2]$.

We are now ready to describe how to compare failure detectors. Let D and D' be two failure detectors. We say that D *is at least as weak as* D' in environment $\mathcal{E}$, denoted by $D \leq_{\mathcal{E}} D'$, if there is a protocol that implements D using D' in environment $\mathcal{E}$. Then D and D' are said *equivalent* in environment $\mathcal{E}$ if $D \leq_{\mathcal{E}} D'$

and $D' \leq_{\mathcal{E}} D$. These notions are motivated by the fact that if a failure detector D can be used to solve some task T in some environment $\mathcal{E}$ then every failure detector D' such that $D \leq_{\mathcal{E}} D'$ can be used as well to solve the task T. For example, consensus can be solved using Ω [18]. If $\Omega \leq_{\mathcal{E}} D$, then, in $\mathcal{E}$, one can compose a protocol $\mathcal{B}$ that implements Ω using D with a protocol $\mathcal{A}$ solving consensus using Ω. Finally, a failure detector D is said to be a *weakest* failure detector for a task T in environment $\mathcal{E}$ if and only if (1) there is a protocol that solves T using D in environment $\mathcal{E}$ and, (2) for every failure detector D' that can be used to solve T in $\mathcal{E}$, we have $D \leq_{\mathcal{E}} D'$. For example, it has been shown [2] that Ω is a weakest failure detector for consensus in the *majority* environment, i.e., the environment in which every failure pattern $\mathcal{F}$ satisfies $|\mathsf{faulty}(\mathcal{F})| < \frac{n}{2}$. Also, Ω is the weakest failure detector to implement eventual consistency [5].

3 Perfect failure detection requires $\omega(1)$ bits per process

In this section, we show that any failure detector emulating the perfect failure detector P must output values whose range depends on the size n of the system.

Theorem 1. *A failure detector that outputs a constant number of bits at each process cannot emulate the perfect failure detector P.*

Preliminaries. Key ingredients in the proof of Theorem 1 are *Ramsey's Theorem* and some elements of the *well-quasi-order* theory.

Ramsey's Theorem might be seen as a generalization of the pigeonhole principle. The statement of its finite version, which we are going to use in the proof is recalled below. An n-subset is a subset of size n and coloring α is a function that maps each n-subset to some element of a set of size c.

Theorem 2 (Ramsey's Theorem). *For all natural numebrs n, m, and c, there exists a natural number $g(n, m, c)$ with the following property. For every set S of size at least $g(n, m, c)$, and any coloring of the n-subsets of S with at most c colors, there is some subset C of S of size m that has all of its n-subsets colored the same color.*

We recall next some basic notions of well-quasi order theory. Let A be a (finite or infinite) set, and let $\preceq$ be a binary relation over A. A (finite or infinite) sequence $a_1, a_2, \ldots, a_\ell$ of elements of A is *good* if there exists two indices $i < j$ such that $a_i \preceq a_j$. Otherwise, if for every $i < j$, $a_i \not\preceq a_j$, the sequence is said to be *bad*. The pair $(A, \preceq)$ is a *well-quasi-order* (*wqo* for short), if (1) $\preceq$ is transitive and reflexive, and (2) every infinite sequence of elements of A is good.

A finite sequence $a_1, a_2, \ldots, a_k$ of elements of A is called a *word*. Let A^* denote the set of words, and let $\preceq_*$ be the sub-word relation over A^* induced by the relation $\preceq$. That is, $a = a_1, \ldots, a_k \preceq_* b = b_1, \ldots, b_\ell$ if and only if $k \leq \ell$ and there exists a strictly increasing mapping $m : [1, k] \to [1, \ell]$ such that $a_i \preceq b_{m(i)}$, for every i, $1 \leq i \leq k$. Higman's lemma essentially states that every bad sequence of words in $(A^*, \preceq_*)$ is finite whenever $(A, \preceq)$ is a wqo:

Lemma 1 (Higman's lemma [11]). *If $(A, \preceq)$ is a well-quasi-order, then so is $(A^*, \preceq_*)$.*

For the purpose of establishing Theorem 1, we are interested in $(\Sigma^*, =^*)$ where Σ is a *finite* set, and $=^*$ denotes the sub-word relation based on the equality relation. That is, for any two words $a = a_1, \ldots, a_k, b = b_1, \ldots, b_\ell$, $a =^* b \in \Sigma^*$ if and only if there exists a strictly increasing map $m : [1, k] \to [1, \ell]$ such that $a_i = b_{m(i)}$, for every $i, 1 \leq i \leq k$.

Since Σ is finite, $(\Sigma, =)$ is a wqo. It thus follows from Higman's lemma that $(\Sigma^*, =^*)$ is also a wqo. Hence, every bad sequence over $(\Sigma^*, =^*)$ is finite. We are interested in the maximal length of such bad sequences. Of course, if no further assumption is made, bad sequences of arbitrary lengths can be constructed. However, in the case of *controlled* bad sequences, (coarse) upper bounds on the length of bad sequence have been established:

Theorem 3 (Length function Theorem [20]). *For a finite set Σ and a given $d \in \mathbb{N}$, let $L_{\Sigma^*}(d)$ be the maximal length of bad sequences $x_0, x_1, x_2, \ldots$ over $(\Sigma^*, =^*)$ such that $|x_i| \leq f^i(d) = f(f(...f(d)))$ for $i = 0, 1, 2, \ldots$. If the* control function *f is primitive-recursive, then the* length function *$L_{\Sigma^*}(d)$ is bounded by a function in $\mathcal{F}_{\omega^{|\Sigma|-1}}$.*[6]

In the proof below, we will construct a sequences $x = x_1, x_2, \ldots$ over $(\Sigma^*, =^*)$ where $|x_i| = i$, for every $i = 1, 2, \ldots$, i.e., we will restrict our attention to sequences controlled by the successor function $f : n \to n + 1$, whose initial element has length 1. By the Length function Theorem, there is a bound on the length of every such sequence that is bad, depending solely on the cardinality of Σ:

Corollary 1. *Let Σ be a finite set. There exists an integer L_{Σ^*} with the following property: Every bad sequence $x = x_1, x_2, \ldots$ over $(\Sigma^*, =^*)$ such that $|x_i| = i$ for every $i = 1, 2, \ldots$ has length at most L_{Σ^*}.*

Overview of the Proof of Theorem 1. Let Σ be a finite set. We are going to show that there exists an integer N such that no failure detector with range Σ can emulate the perfect failure detector P in a N-process system.

For the sake of contradiction, let us assume that there is a failure detector X whose range is Σ and an algorithm $\mathcal{T}_{X \to P}$ that emulates P in a system $\Pi = \{p_1, \ldots, p_N\}$ consisting of N processes. We aim at constructing two executions $exec_1$ and $exec_2$ that are indistinguishable to a subset of the processes up to a certain point in time. However, the executions have different failure patterns and, by leveraging the indistinguishably of $exec_1$ and $exec_2$ from the perspective of some processes, we show that in one of these executions, a correct process is erroneously suspected by the emulated perfect failure detector.

[6] The function classes $\mathcal{F}_\alpha$ are the elementary-recursive closure of the functions F_α, which are the ordinal-indexed levels of the Fast-Growing Hierarchy [14]. Multiply-recursive complexity starts at level $\alpha = \omega$, i.e., Ackermannian complexity, and stops just before level $\alpha = \omega^\omega$, i.e., Hyper-Ackermannian complexity.

For a process p not to be able to distinguish between $exec_1$ and $exec_2$, it must in particular receive the same sequence of outputs from the underlying failure detector X in the two executions. Let $\mathcal{F}$ be a failure pattern in which p is correct. Since the range of X is finite, in any valid history $H \in X(\mathcal{F})$, there exists a symbol in the range Σ of X that is output infinitely often at process p. Hence, by appropriately scheduling the queries to the failure detector X, we can concentrate on executions in which the failure detector output is constant at each process. Furthermore, we only consider failure patterns in which all faulty processes fail initially, e.g., before taking any step in the emulation algorithm.

Each such execution $exec$ can be associated with a word $x_{exec} \in \Sigma^*$, namely the word formed by the failure detector constant outputs at each correct process. More precisely, the rth symbol of x_{exec} is the (constant) output of the failure detector X at the rth correct process, where processes are ordered by increasing ids. By considering executions with increasing sets of correct processes, we obtain a sequence $x = x_1, x_2, \ldots$ of words of Σ^*. If the system is sufficiently large and, thus, the induced sequence x of words in Σ^* is sufficiently long, x is good by The Length function Theorem. Hence, we are able to exhibit two words $x_{i_1}, x_{i_2}, i_1 < i_2$ where x_{i_1} is a sub-word of x_{i_2}. By construction, these words represent in fact the outputs of X at the correct processes in two executions $exec_1$ and $exec_2$ respectively with two different sets of correct processes.

Hence, in $exec_1$ and $exec_2$, two sets of correct processes of the same size (i_1) get the same output of X, although the failure patterns differ. This is, however, insufficient to conclude that $exec_1$ and $exec_2$ are indistinguishable from the perspective of some processes. Indeed, a common symbol in x_1 and x_2 may be output in $exec_1$ and $exec_2$ by processes with distinct ids. We resolve this issue by leveraging Ramsey's Theorem. We show that in a sufficiently large system, there is a subset S of processes of size strictly larger than L_{Σ^*} for which the outputs of X are essentially id-oblivious: sets of correct processes in S of the same size are provided with the same failure-detector outputs.

In more detail, for any set of processes C, let $\mathcal{F}_C$ denote the failure pattern in which the set of correct processes is C and every faulty process crashes at time 0. Let $L = L_{\Sigma^*} + 1$, where L_{Σ^*} is the bound in Corollary 1. Provided that the total number of processes is large enough, we show that there exists a sequence $x = x_1, \ldots, x_L$ of words of Σ^* such that $|x_i| = i$ for every $i, 1 \leq i \leq L$ with the following property. For every i-subset $C \subseteq S$, $1 \leq i \leq L$, there is a failure detector history in $X(\mathcal{F}_C)$ in which for each r, $1 \leq r \leq i$ the failure detector outputs infinitely often the rth symbol of x_i to the rth process (where the processes are ordered by increasing ids).

By construction, sequence x is good and, thus, in some executions $exex_1$ and $exec_2$ with failure patterns $\mathcal{F}_{C_1}$ and $\mathcal{F}_{C_2}$, respectively, where $C_1 \subsetneq C_2$, processes in C_1 obtain the same information about failures. But the given emulation of P in $exec_1$ should indicate eventually, at some time τ, that some process in $C_2 \setminus C_1$ is faulty. Now by delaying all messages of processes in $C_2 \setminus C_1$ in execution $exec_2$ until after τ, we get that $exec_1$ and $exec_2$ are indistinguishable to C_1 up to time τ and, thus, in $exec_2$, some correct process is suspected—a contradiction.

4 Perfect failure detection with quasi-constant #bits per process

In this section, we show that there exists a failure detector emulating the perfect failure detector P that outputs values whose range depends on the size of the system, but increases extremely slowly with that size.

Theorem 4. *There exists a failure detector equivalent to P that, in any n-process system, outputs $\lceil \log \alpha(n) \rceil + 1$ bits at each process, where $\alpha : \mathbb{N} \to \mathbb{N}$ is a function that grows as least as slowly as the inverse Ackermann function.*

The rest of the section is dedicated to the proof of Theorem 4. A key ingredient to achieve failure detection with such a small amount of output values at each process is the notion of *distributed encoding of the integers*, recently introduced in [6]. Let Σ denote a finite or infinite alphabet of symbols. Recall from Section 3 that a word w over Σ is a finite sequence of symbols of Σ, and the length of w, i.e., the number of symbols in w, is denoted by $|w|$. Σ^* denotes the set of words of Σ, and a word $u = u_1, \ldots, u_k$ is a sub-word of word $v = v_1, \ldots, v_\ell$, denoted by $u =_* v$ if and only if $k \leq \ell$ and there exists a strictly increasing mapping $m : [1, k] \to [1, \ell]$ such that $u_i = v_{m(i)}$, for every i, $1 \leq i \leq k$. u is said to be a *strict sub-word* of v if $u =_* v$ and $|u| < |v|$.

Definition 1 ([6]). *A* distributed encoding of the integers *is a pair (Σ, f) where Σ is a possibly infinite alphabet and $f : \Sigma^* \to \{\text{true}, \text{false}\}$ is a function such that, for every integer $n \geq 1$, there exists a word $w = w_1, \ldots, w_n \in \Sigma^n$ satisfying $f(w) = \text{true}$ and $f(w') = \text{false}$ for every strict sub-word $w' \in \Sigma^*$ of w. The word w is called the* code *of n, denoted by $\mathsf{code}(n)$.*

A trivial example of a distributed encoding consists in setting $\Sigma = \mathbb{N}$, and encoding every integer n with the word $n, \ldots, n$ of length n. For any $s \in \mathbb{N}^*$, the function f returns true on input s if $s = |s|, \ldots, |s|$, and false otherwise. To encode the first n integers, this encoding uses words in an alphabet of n symbols, each symbols being encoded on $O(\log n)$ bits. We are interested in parsimonious distributed encodings of the integers, i.e., encodings that use fewer than $\log n$ bits to encode the first n integers. Given a distributed encoding $E = (\Sigma, f)$, and $n \geq 1$, let $\Sigma_n \subseteq \Sigma$ denote the set of all symbols used in the code of at least one integer in $[1, n]$. More precisely, for every $u \in \Sigma$, $u \in \Sigma_n$ if and only if u is a symbol appearing in $\mathsf{code}(k)$ for some $1 \leq k \leq n$. We get that E uses symbols encoded on $O(\log |\Sigma_n|)$ bits to encode the first n integers.

Theorem 5 ([6]). *There exists a distributed encoding of the integers (Σ, f) such that, for every integer $n \geq 1$, $|\Sigma_n| \leq \alpha(n)$ where $\alpha : \mathbb{N} \to \mathbb{N}$ grows as least as slowly as the inverse-Ackermann function.*

We now show how to use distributed encoding of the integer to encode a perfect failure detector. Let (Σ, f) be a distributed encoding of the integers. We define a failure detector, called *micro-perfect*, and denoted by μP, induced by (Σ, f). We then show that μP is equivalent to the perfect failure detector P. That

is, for any distributed encoding (Σ, f), there is a protocol that emulates P in any environment whenever the failure detector μP induced by (Σ, f) is available, and, conversely, there is a protocol that emulates μP in any environment whenever the failure detector P is available. Combining these two results with Theorem 5 yields Theorem 4.

The failure detector μP. Let (Σ, f) be a distributed encoding of the integers. An instance of the failure detector μP is built on top of each such encoding. Given (Σ, f), the range of μP is Σ. We denote by w_i^τ the output $H(p_i, \tau)$ of a failure detector history for μP at time τ at process p_i. Let w^τ denotes the output sequence of the failure detector at time τ at the processes that have not crashed by time τ, ordered by processes IDs. More formally, $w^\tau = w_{j_1}^\tau, w_{j_2}^\tau, \ldots, w_{j_k}^\tau$ where $\{p_{j_1}, \ldots, p_{j_k}\} = \Pi \setminus \mathcal{F}(\tau)$, and $\mathsf{id}(p_{j_1}) < \ldots < \mathsf{id}(p_{j_k})$. For every failure pattern $\mathcal{F}$, a failure detector history H belongs to $\mu P(\mathcal{F})$ if and only if there exists $\ell \in [1, n]$ (recall that n is the number of processes) for which

- there exist $a_i \in \mathbb{N}$ for $i = 1, \ldots, \ell$, with $1 \leq a_\ell < a_{\ell-1} < \ldots < a_2 < a_1 \leq n$;
- there exist $\tau_i \in \mathbb{N}$ for $i = 0, \ldots, \ell$, with $0 = \tau_0 < \tau_1 < \ldots < \tau_\ell = +\infty$

such that, for every i with $1 \leq i \leq \ell$, and for every τ, τ' with $\tau_{i-1} \leq \tau, \tau' < \tau_i$, the four following conditions hold:

(C1) $w_j^\tau = w_j^{\tau'}$ for every p_j, i.e., the output of the failure detector does not change between τ_{i-1} and τ_i;
(C2) $w^\tau =_* \mathsf{code}(a_i)$, i.e., the word formed by the outputs of the failure detector at each process that has not crashed by time τ is a sub-word of the code of a_i;
(C3) $a_i \geq n - |\mathcal{F}(\tau)|$, i.e., a_i is an upper bound on the number of non-faulty processes during $[\tau_{i-1}, \tau_i)$;
(C4) $a_\ell = |\mathsf{correct}(\mathcal{F})|$, i.e., a_ℓ is the number of correct processes.

Let us consider some time τ, and let $k = |\Pi \setminus \mathcal{F}(\tau)|$ denote the number of processes that have not crashed by time τ. By concatenating the failure detector outputs of the non-crashed processes (ordered by process IDs), we obtain a word $w^\tau \in \Sigma^k$. The failure detector μP guarantees that this word is either the distributed code of the current number k of non-crashed processes, or a sub-word of the distributed code of some integer $k' > k$. Moreover, eventually, μP outputs the distributed code of the number of correct processes.

Failure detector μP can emulate the perfect failure detector P. Protocol 1 emulates the perfect failure detector P using μP, in any environment.

Each time a query to P occurs, the protocol strives to identify a set of processes that (1) contains every correct process and (2) does not contain the processes that have failed prior to the beginning of the query. Given such a set S, $\Pi \setminus S$ is a valid output for P (line 4), as it does not contain any correct process (Accuracy), and, if the query starts after every faulty process has failed, $\Pi \setminus S$ is exactly the set of faulty processes (Completeness).

Protocol 1 Emulation of P using μP induced by (Σ, f). Code of Process p_i.

```
 1: function P-QUERY()
 2:     for all S ⊆ Π : p_i ∈ S do launch thread th_S computing CHECK(S) end for
 3:     wait until ∃S : th_S terminates; stop all other threads th_S' for S' ≠ S
 4:     return Π \ S
 5: function CHECK(S)
 6:     r ← 0; count ← 0
 7:     repeat
 8:         r ← r + 1 ; send query(S, r) to every p_j ∈ S
 9:         wait until resp((S, r),w_j) has been received from every p_j ∈ S
10:         w ← w_{j_1}, ..., w_{j_s} where S = {p_{j_1}, ..., p_{j_s}} and id(p_{j_1}) < ... < id(p_{j_s})
11:         if f(w) = true then count ← count + 1 end if
12:     until count = n
13: when query(S, r) is received from p_j do
14:     w_i ← μP-QUERY(); send resp((S, r), w_i) to p_j
```

When P-QUERY() is invoked by some process p_i, 2^{n-1} threads are launched (line 2), one for each subset of Π containing p_i. Thread th_S associated to set S consists in a **repeat** loop (lines 7–12), each iteration of which aiming at collecting the outputs of the underlying failure detector μP at the processes of S. Each iteration is identified by a round number r[7]. In iteration r, *query* messages are first sent to every process in S (line 8), and then p_i waits for a matching *response* message[8] from each process in S (line 9). Iteration r may never ends if some processes of S fail. Nevertheless, for at least one set S, namely the set of correct processes, every iteration of the associated thread th_S terminates.

Each of the *response* messages received by p_i contains the output w_j of μP at its sender p_j when the message is sent (line 14). Assuming that *response* have been received from each process $p_j \in S$, let w be the word obtained by concatenating the outputs of μP in these messages, ordered by process id (line 10). Recall that a valid history of failure detector μP can be divided into ℓ *epochs* $e_1 = [0, \tau_1), e_2 = [\tau_1, \tau_2), \ldots, e_\ell = [\tau_{\ell-1}, +\infty)$, for some $\ell \leq n$. In each epoch $e_k, 1 \leq k \leq \ell$, the output of μP at each process does not change (cf. **(C1)**) and satisfy conditions **(C1)**–**(C4)** of the definition.

Let us assume that iteration r entirely fits within epoch e_k for some $k, 1 \leq k \leq \ell$. That is, every message *query* or *response* of that iteration is sent during e_k. Thus, by condition **(C2)**, w is a sub-word of $\mathsf{code}(a_k)$, i.e., the encoding of the integer a_k associated with epoch e_k by the distributed code (Σ, f). By using the function f, it can be determined whether $w = \mathsf{code}(a_k)$ or not. Indeed, for every proper sub-word w' of $\mathsf{code}(a_k)$, $f(w')$ = false and $f(\mathsf{code}(a_k))$ = true (cf. Definition 1). Moreover, integer a_k is an upper bound on the number of alive processes in epoch e_k (cf. **(C3)**). Therefore, if $f(w) =$ true, then $w = \mathsf{code}(a_k)$, and, since $|w| = |S|$, we get $|S| = a_k$. Since all processes in S have not failed

[7] Round numbers may be omitted, we keep them to simplify the proof of the protocol.

[8] *query* and *response* messages are implicitly tagged in order not to confuse messages sent during different invocations of P-QUERY().

at the beginning of e_k (as each of them has sent a *response* in that interval), it follows that every process not in S has failed. Furthermore, if $k = \ell$, then a_ℓ is the number of correct processes (cf. **(C4)**), and thus in that case $\Pi \setminus S$ is the complete set of faulty processes. To summarize, if the word w collecting during iteration r satisfies $f(w) = \text{true}$, and iteration r entirely fits within an epoch, then $\Pi \setminus S$ is a valid output of P.

Unfortunately, it may be the case that an iteration terminates while not fitting entirely within an epoch. The word w collected during that iteration may contain values output by the failure detector μP in distinct epochs. It is thus no longer guarantied that w is a sub-word of a valid code, and, from the fact that $f(w) = \text{true}$, it can no longer be concluded that $\Pi \setminus S$ is a valid output of P. Recall however that the ℓ epochs are consecutive, they span the whole time range (last epoch e_ℓ never ends), and there are at most n of them. Hence, if n iterations terminate, at least one of these iterations fits entirely in an epoch. In thread th_S, the variable *count* enumerate the number of iterations that terminate with an associated word w such that $f(w) = \text{true}$ (cf. line 11). When this counter reaches the value n, at least one successful iteration fitting entirely in an epoch has occurred, and $\Pi \setminus S$ can therefore be returned as a valid result of a query to P (cf. lines 3–4).

Failure detector P can emulate the failure detector μP. Failure detectors P and μP are in fact equivalent. Protocol 2 emulates μP using the perfect failure detector P, in any environment.

Protocol 2 Emulation of μP induced by (Σ, f) using P. Code of Process p_i.

```
 1: init alive ← {p_1, ..., p_n}; r ← 0
 2: function μP-QUERY()
 3:     repeat
 4:         r ← r + 1; S ← P-QUERY(); alive ← alive \ S
 5:         send query(r, alive) to all other processes
 6:         repeat S ← P-QUERY()
 7:         until response(r, a_j) has been received from every p_j ∈ Π \ S
 8:         rec ← set of all received sets a_j
 9:     until there exists a ⊆ Π such that rec = {a}
10:     k ← rank of id(p_i) in a; w_i ← kth symbol of code(|a|)
11:     return w_i
12: when query(r, a) is received from p_j do
13:     S ← P-QUERY(); alive ← (alive ∩ a) \ S; send response(r, alive) to p_j
```

Proof of Theorem 4. By Theorem 5 there exists a distributed encoding of the integers (Σ, f) where $|\Sigma_n| \leq \alpha(n)$, for some $\alpha : \mathbb{N} \to \mathbb{N}$ growing at least as slowly as the inverse Ackermann function. That is, for any n, each symbol in the code of n is encoded on $\lceil \log \alpha(n) \rceil + 1$ bits. Consider failure detector μP induced by the

distributed encoding (Σ, f). In an n-process system, any output of this failure detector is a symbol in Σ that is part of the code of some integer $n' \leq n$. Hence, the output of μP can be encoded on $\lceil \log \alpha(n) \rceil + 1$ bits at each process. Moreover it follows from the correctness of Protocols 1 and 2 that μP is equivalent to the perfect failure detector P. □

References

1. A. R. Byron Cook, A. Podelski. Proving program termination. *Communications of the ACM*, 54(5):88–98, 2011.
2. T. D. Chandra, V. Hadzilacos, and S. Toueg. The weakest failure detector for solving consensus. *J. ACM*, 43(4):685–722, 1996.
3. T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2):225–267, 1996.
4. C. Delporte-Gallet, H. Fauconnier, and S. Toueg. The minimum information about failures for solving non-local tasks in message-passing systems. *Distributed Computing*, 24(5):255–269, 2011.
5. S. Dubois, R. Guerraoui, P. Kuznetsov, F. Petit, and P. Sens. The weakest failure detector for eventual consistency. In *ACM Symposium on Principles of Distributed Computing (PODC)*, pages 375–384, 2015.
6. P. Fraigniaud, S. Rajsbaum, and C. Travers. Minimizing the number of opinions for fault-tolerant distributed decision using well-quasi orderings. In *12th Latin American Theoretical Informatics Symposium (LATIN)*, LNCS #9644 pp. 497–508, Springer, 2016.
7. P. Fraigniaud, S. Rajsbaum, C. Travers, P. Kuznetsov and T. Rieutord. Perfect failure detection with very few bits. Technical Report Hal-01365304, LaBRI, 2016. `https://hal.inria.fr/hal-01365304`
8. F.C. Freiling, R. Guerraoui, and P. Kuznetsov. The failure detector abstraction. *ACM Computing surveys*, 43(2):9, 2011.
9. R. Guerraoui. Non-blocking atomic commit in asynchronous distributed systems with failure detectors. *Distributed Computing*, 15(1):17–25, 2002.
10. C. Haase, S. Schmitz, and P. Schnoebelen. The power of priority channel systems. *Logical Methods in Computer Science*, 10(4), 2014.
11. G. Higman. Ordering by divisibility in abstract algebras. *Proceedings of the London Mathematical Society*, 3(2):326–336, 1952.
12. P. Jayanti and S. Toueg. Every problem has a weakest failure detector. In *27th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 75–84, 2008.
13. J.B. Kruskal. The theory of well-quasi-ordering: A frequently discovered concept. *Journal of Combinatorial Theory, Series A*, 13(3):297–305, 1972.
14. M.H. Löb and S.S. Wainer. Hierarchies of number theoretic functions. I *Archive for Mathematical Logic*, 13: 39–51, 1970.
15. E. Milner. Basic WQO- and BQO-theory. In *Graphs and Order, The Role of Graphs in the Theory of Ordered Sets and Its Applications*, NATO ASI Series, pages 487–502, 1985.
16. A. Mostéfaoui, S. Rajsbaum, M. Raynal, and C. Travers. The combined power of conditions and information on failures to solve asynchronous set agreement. *SIAM Journal of Computing*, 38(4):1574–1601, 2008.

17. A. Mostéfaoui, S. Rajsbaum, M. Raynal, and C. Travers. On the computability power and the robustness of set agreement-oriented failure detector classes. *Distributed Computing*, 21(3):201–222, 2008.
18. A. Mostéfaoui and M. Raynal. Leader-based consensus. *Parallel Processing Letters*, 11(1):95–107, 2001.
19. S. Schmitz and P. Schnoebelen. Algorithmic aspects of WQO theory. Technical Report Hal-00727025, HAL, 2013. `https://cel.archives-ouvertes.fr/cel-00727025`
20. S. Schmitz and P. Schnoebelen. Multiply-Recursive Upper Bounds with Higman's Lemma In *38th International Colloquium in Automata, Languages and Programming (ICALP)* pages 441–452, LNCS#6756, Springer 2011.
21. A. Turing. Checking a large routine. In *Conference on High Speed Automatic Calculating Machines*, pages 67–69, 1949.