Simple CHT:
A New Derivation of the Weakest Failure Detector for Consensus

Petr Kuznetsov

TU Belrin/Deutsche Telekom Laboratories

Abstract

The paper proposes an alternative proof that Ω, an oracle that outputs a process identifier and guarantees that eventually the same correct process identifier is output at all correct processes, provides minimal information about failures for solving consensus in read-write shared-memory systems: every oracle that gives enough failure information to solve consensus can be used to implement Ω.

Unlike the original proof by Chandra, Hadzilacos and Toueg (CHT), the proof presented in this paper builds upon the very fact that 2-process wait-free consensus is impossible. Also, since the oracle that is used to implement Ω can solve consensus, the implementation is allowed to directly access consensus objects. As a result, the proposed proof is shorter and conceptually simpler than the original one.

1 Introduction

The presence of faults and the lack of synchrony make distributed computing interesting. In an *asynchronous* system which assumes no bounds on communication delays and relative processing speeds, even a basic form of non-trivial synchronization (*consensus*) is impossible if just one process may fail by crashing [7, 13]. On the other hand, in a *synchronous* system, in the bounds exist and known a priori, almost any meaningful fault-tolerant synchronization problem becomes solvable. On the other hand, This gap suggests that the amount of synchrony is a crucial factor in reasoning about solvability of distributed computing problems in the presence of faults.

Chandra and Toueg proposed *failure detectors* as a convenient language to describe synchrony assumptions. Informally, a failure detector is a distributed oracle that provides processes with hints about failures [5]. The notion of a *weakest failure detector* [4] captures the exact amount of synchrony needed to solve a given problem: $\mathcal{D}$ is the weakest failure detector for solving $\mathcal{M}$ if (1) $\mathcal{D}$ is sufficient to solve $\mathcal{M}$, i.e., there exists an algorithm that solves $\mathcal{M}$ using $\mathcal{D}$, and (2) any failure detector $\mathcal{D}'$ that is sufficient to solve $\mathcal{M}$ provides at least as much information about failures as $\mathcal{D}$ does, i.e., there exists a *reduction* algorithm that extract the output of $\mathcal{D}$ using the failure information provided by $\mathcal{D}'$.

This paper considers a distributed system in which n crash-prone processes communicate using atomic reads and writes in shared memory. In the (binary) consensus problem [7], every process starts with a binary input and every correct (never-failing) process is supposed to output one of the inputs such that no two processes output different values. Asynchronous (wait-free) consensus is known to be impossible [7, 13], as long as at least one process may fail by crashing. Chandra et al. [4] showed that the "eventual leader" failure detector Ω is necessary and sufficient to solve consensus. The failure detector Ω outputs, when queried, a process identifier, such that, eventually, the same correct process identifier is output at all correct processes.

The reduction technique presented in [4] is very interesting in its own right, since it not only allows us to determine the weakest failure detector for consensus, but also establishes a framework for determining the weakest failure detector for any problem. Informally, the reduction algorithm of [4] works as follows. Let $\mathcal{D}$ be any failure detector that can be used to solve consensus. Processes periodically query their modules of $\mathcal{D}$, exchange the values returned by $\mathcal{D}$, and arrange the accumulated output of the failure detector in the form of ever-growing directed acyclic graphs (DAGs). Every process periodically uses its DAG as a stimulus for simulating multiple runs of the given consensus algorithm. It is shown in [4] that, eventually, the collection of simulated runs will include a *critical* run in which a single process p "hides" the decided value, and, thus, no extension of the run can reach a decision without cooperation of p. As long as a process performing the simulation observes a run that the process suspects to remain critical, it outputs the "hiding" process identifier of the "first" such run as the extracted output of Ω. The existence of a critical run and the fact that the correct processes agree on ever-growing prefixes of simulated runs imply that, eventually, the correct processes will always output the identifier of the same correct process.

Crucially, the existence of a critical run is established in [4] using the notion of *valence* [7]: a simulated finite run is called v-valent ($v \in \{0, 1\}$) if all simulated extensions of it decide v. If both decisions 0 and 1 are "reachable" from the finite run, then the run is called bivalent. Recall that in [7], the notion of valence is used to derive a critical run, and then it is shown that such a run cannot exist in an asynchronous system, implying the impossibility of consensus. In [4], a similar argument is used to extract the output of Ω in a partially synchronous system that allows for solving consensus. Thus, in a sense, the technique of [4] rehashes arguments of [7]. It is challenging to find a proof that Ω is necessary to solve consensus building upon the very fact that 2-process wait-free consensus is impossible.

This paper addresses this challenge. It is shown that Ω is necessary to solve consensus using the very impossibility of 2-process wait-free consensus, without "opening the box" and considering the problem semantics. The resulting proof is shorter and simpler that the original proof of [4].

On the technical side, the paper uses two fundamental results and one observation. First, the technique of Zieliński [15] that allows us to construct, given an algorithm $\mathcal{A}$ that uses a failure detector $\mathcal{D}$, an *asynchronous* algorithm $\mathcal{A}'$ that simulates runs of $\mathcal{A}$ using a static sample of $\mathcal{D}$'s output (captured in a DAG), instead of the "real" output of the failure detector. The live processor set that run $\mathcal{A}'$ may be different than the live set of processor implied by the sample of $\mathcal{D}$. Therefore, the asynchronous algorithm $\mathcal{A}'$ guarantees that every infinite simulated run is *safe* (every prefix of it is a finite run of $\mathcal{A}$), but not necessarily *live* (some correct process may not be able to make progress).

Second, the paper also makes use of the BG-simulation technique [1, 3] that allows $k + 1$ processes simulate, in a wait-free manner, a *k-resilient* (with at most k faulty processes) run of $\mathcal{A}'$. Using a series of consensus instances (provided by the algorithm $\mathcal{A}$ using $\mathcal{D}$), processes locally simulate the very same sequence of 1-resilient runs and eventually identify a "never-deciding" 1-resilient run of $\mathcal{A}'$. Since $\mathcal{A}'$ is an asynchronous simulation of $\mathcal{A}$, a 1-resilient run of $\mathcal{A}'$ that includes infinitely many steps of each correct process should be deciding. Thus, exactly one correct process appears in the 1-resilient never-deciding run only finitely often. To emulate Ω, it is thus sufficient to output the process that appears the least in that 1-resilient run. Eventually, all correct process agree on the same never-deciding 1-resilient run, and will always output the same correct process. The observation here is that a reduction algorithm may directly access consensus objects, since it is given a failure detector which can be used to solve consensus.

The rest of the paper is organized as follows. Section 2 describes the system model. Sections 3 presents the reduction algorithm. Section 4 overviews the related work and Section 5 concludes the paper by discussing implications of the presented results.

2 Model

The model of processes communicating through read-write shared objects and using failure detectors is based on [4, 9, 10, 11]. The details necessary for showing the results of this paper are described below.

2.1 Processes and objects

A distributed system is composed of a set Π of n processes $\{p_1, \dots, p_n\}$ ($n \geq 2$). Processes are subject to *crash* failures. A process that never fails is said to be *correct*. Processes that are not correct are called *faulty*. Process communicate through applying atomic operations on a collection of *shared objects*. In this paper, the shared objects are registers, i.e., they export only conventional atomic read-write operations.

2.2 Failure patterns and failure detectors

A *failure pattern* F is a function from the time range $\mathbb{T} = \{0\} \cup \mathbb{N}$ to 2^{Π}, where $F(t)$ denotes the set of processes that have crashed by time t. Once a process crashes, it does not recover, i.e., $\forall t : F(t) \subseteq F(t+1)$. The set of faulty processes in F, $\cup_{t \in \mathbb{T}} F(t)$, is denoted by *faulty*$(F)$. Respectively, *correct*$(F) = \Pi -$ *faulty*(F). A process $p \in F(t)$ is said to be *crashed* at time t. An *environment* is a set of failure patterns. This paper considers environments that consists of failure patterns in which at least one process is correct.

A *failure detector history H with range $\mathcal{R}$* is a function from $\Pi \times \mathbb{T}$ to $\mathcal{R}$. $H(p_i, t)$ is interpreted as the value output by the failure detector module of process p_i at time t. A *failure detector $\mathcal{D}$* with range $\mathcal{R}_{\mathcal{D}}$ is a function that maps each failure pattern to a (non-empty) set of failure detector histories with range $\mathcal{R}_{\mathcal{D}}$. $\mathcal{D}(F)$ denotes the set of possible failure detector histories permitted by $\mathcal{D}$ for failure pattern F. Possible ranges of failure detectors are not a priori restricted.

2.3 Algorithms

An *algorithm $\mathcal{A}$ using a failure detector $\mathcal{D}$* is a collection of deterministic automata, one for each process in the system. $\mathcal{A}_i$ denotes the automaton on which process p_i runs the algorithm $\mathcal{A}$. Computation proceeds in atomic *steps* of $\mathcal{A}$. In each step of $\mathcal{A}$, process p_i

(i) invokes an atomic operation (read or write) on a shared object and receives a response *or* queries its failure detector module $\mathcal{D}_i$ and receives a value from $\mathcal{D}$, and

(ii) applies its current state, the response received from the shared object or the value output by $\mathcal{D}$ to the automaton $\mathcal{A}_i$ to obtain a new state.

A step of $\mathcal{A}$ is thus identified by a tuple (p_i, d), where d is the failure detector value output at p_i during that step if $\mathcal{D}$ was queried, and $\perp$ otherwise.

If the state transitions of the automata $\mathcal{A}_i$ do not depend on the failure detector values, the algorithm $\mathcal{A}$ is called *asynchronous*. Thus, for an asynchronous algorithm, a step is uniquely identified by the process id.

2.4 Runs

A *state* of $\mathcal{A}$ defines the state of each process and each object in the system. An *initial state* I of $\mathcal{A}$ specifies an initial state for every automaton $\mathcal{A}_i$ and every shared object.

A *run of algorithm* $\mathcal{A}$ *using a failure detector* $\mathcal{D}$ in an environment $\mathcal{E}$ is a tuple $R = \langle F, H, I, S, T \rangle$ where $F \in \mathcal{E}$ is a failure pattern, $H \in \mathcal{D}(F)$ is a failure detector history, I is an initial state of $\mathcal{A}$, S is an *infinite* sequence of steps of $\mathcal{A}$ respecting the automata $\mathcal{A}$ and the sequential specification of shared objects, and T is an *infinite* list of increasing time values indicating when each step of S has occurred, such that for all $k \in \mathbb{N}$, if $S[k] = (p_i, d)$ with $d \neq \bot$, then $p_i \notin F(T[k])$ and $d = H(p_i, T[k])$.

A run $\langle F, H, I, S, T \rangle$ is *fair* if every process in $\mathit{correct}(F)$ takes infinitely many steps in S, and k*-resilient* if at least $n - k$ processes appear in S infinitely often. A *partial run* of an algorithm $\mathcal{A}$ is a finite prefix of a run of $\mathcal{A}$.

For two steps s and s' of processes p_i and p_j, respectively, in a (partial) run R of an algorithm $\mathcal{A}$, we say that s *causally precedes* s' if in R, and we write $s \rightarrow s'$, if (1) $p_i = p_j$, and s occurs before s' in R, or (2) s is a write step, s' is a read step, and s occurs before s' in R, or (3) there exists s'' in R, such that $s \rightarrow s''$ and $s'' \rightarrow s'$.

2.5 Consensus

In the binary consensus problem, every process starts the computation with an input value in $\{0, 1\}$ (we say the process *proposes* the value), and eventually reaches a distinct state associated with an output value in $\{0, 1\}$ (we say the process *decides* the value). An algorithm $\mathcal{A}$ solves consensus in an environment $\mathcal{E}$ if in every *fair* run of $\mathcal{A}$ in $\mathcal{E}$, (i) every correct process eventually decides, (ii) every decided value was previously proposed, and (iii) no two processes decide different values.

Given a an algorithm that solves consensus, it is straightforward to implement an abstraction cons that can be accessed with an operation *propose*(v) ($v \in \{0, 1\}$) returning a value in $\{0, 1\}$, and guarantees that every *propose* operation invoked by a correct process eventually returns, every returned value was previously proposed, and no two different values are ever returned.

2.6 Weakest failure detector

We say that an algorithm $\mathcal{A}$ using $\mathcal{D}'$ *extracts* the output of $\mathcal{D}$ in $\mathcal{E}$, if $\mathcal{A}$ implements a distributed variable $\mathcal{D}$*-output* such that for every run $R = \langle F, H', S, T \rangle$ of $\mathcal{A}$ in which $F \in \mathcal{E}$, there exists $H \in \mathcal{D}(F)$ such that for all $p_i \in \Pi$ and $t \in \mathbb{T}$, $\mathcal{D}$*-output*$_i(t) = H(p_i, t)$ (i.e., the value of $\mathcal{D}$*-output* at p_i at time t is $H(p_i, t)$). We say that $\mathcal{A}$ is a *reduction* algorithm. (A more precise definition of a reduction algorithm is given in [11].)

If, for failure detectors $\mathcal{D}$ and $\mathcal{D}'$ and an environment $\mathcal{E}$, there is a reduction algorithm using $\mathcal{D}'$ that extracts the output $\mathcal{D}$ in $\mathcal{E}$, then we say that $\mathcal{D}$ *is weaker than* $\mathcal{D}'$ in $\mathcal{E}$.

$\mathcal{D}$ is the weakest failure detector to solve a problem $\mathcal{M}$ (e.g., consensus) in $\mathcal{E}$ if there is an algorithm that solves $\mathcal{M}$ using $\mathcal{D}$ in $\mathcal{E}$ and $\mathcal{D}$ is weaker than any failure detector that can be used to solve $\mathcal{M}$ in $\mathcal{E}$.

3 Extracting Ω

Let $\mathcal{A}$ be an algorithm that solves consensus using a failure detector $\mathcal{D}$. The goal is to construct an algorithm that emulates Ω using $\mathcal{A}$ and $\mathcal{D}$. Recall that to emulate Ω means to output, at each time and at each process, a process identifiers such that, eventually, the same correct process is always output.

```
Shared variables:
  for all $p_i \in \Pi$: $G_i$, initially empty graph

$k_i := 0$
while true do
  for all $p_j \in \Pi$ do $G_i \leftarrow G_i \cup G_j$
  $d_i :=$ query failure detector $\mathcal{D}$
  $k_i := k_i + 1$
  add $[p_i, d_i, k_i]$ and edges from all other vertices
      of $G_i$ to $[p_i, d_i, k_i]$, to $G_i$
```

Figure 1. Building a DAG: the code for each process p_i

3.1 Overview

As in [4], the reduction algorithm of this paper uses failure detector $\mathcal{D}$ to construct an ever-growing *directed acyclic graph* (DAG) that contains a sample of the values output by $\mathcal{D}$ in the current run and captures some temporal relations between them. Following [15], this DAG can be used by an *asynchronous* algorithm $\mathcal{A}'$ to simulate a (possibly finite and unfair) run of $\mathcal{A}$. Recall that, using BG-simulation [1, 3], 2 processes can simulate a 1-resilient run of $\mathcal{A}'$. The fact that 1-resilient 2-process consensus is impossible implies that the simulation must produce at least one ”non-deciding” 1-resilient run of $\mathcal{A}'$.

Now every correct process locally simulates all executions of BG-simulation on two processes q_1 and q_2 that simulate a 1-resilient run of $\mathcal{A}'$ of the whole system Π. Eventually, every correct process locates a never-deciding run of $\mathcal{A}'$ and uses the run to extract the output of Ω: it is sufficient to output the process that takes the least number of steps in the “smallest” non-deciding simulated run of $\mathcal{A}'$. Indeed, exactly one correct process takes finitely many steps in the non-deciding 1-resilient run of $\mathcal{A}'$: otherwise, the run would simulate a fair and thus deciding run of $\mathcal{A}$.

The reduction algorithm extracting Ω from $\mathcal{A}$ and $\mathcal{D}$ consists of two components that are running in parallel: the *communication component* and the *computation component*. In the communication component, every process p_i maintains the ever-growing directed acyclic graph (DAG) G_i by periodically querying its failure detector module and exchanging the results with the others through the shared memory. In the computation component, every process simulates a set of runs of $\mathcal{A}$ using the DAGs and maintains the extracted output of Ω.

3.2 DAGs

The communication component is presented in Figure 1. This task maintains an ever-growing DAG that contains a finite sample of the current failure detector history. The DAG is stored in a register G_i which can be updated by p_i and read by all processes.

DAG G_i has some special properties which follow from its construction [4]. Let F be the current failure pattern, and $H \in \mathcal{D}(F)$ be the current failure detector history. Then a fair run of the algorithm in Figure 1 guarantees that there exists a map $\tau : \Pi \times \mathcal{R}_{\mathcal{D}} \times \mathbb{N} \mapsto \mathbb{T}$, such that, for every correct process p_i and every time t ($x(t)$ denotes here the value of variable x at time t):

(1) The vertices of $G_i(t)$ are of the form $[p_j, d, \ell]$ where $p_j \in \Pi$, $d \in \mathcal{R}_{\mathcal{D}}$ and $\ell \in \mathbb{N}$.

(a) For each vertex $v = [p_j, d, \ell]$, $p_j \notin F(\tau(v))$ and $d = H(p_j, \tau(v))$. That is, d is the value output by p_j's failure detector module at time $\tau(v)$.

(b) For each edge (v, v'), $\tau(v) < \tau(v')$. That is, any edge in G_i reflects the temporal order in which the failure detector values are output.

(2) If $v = [p_j, d, \ell]$ and $v' = [p_j, d', \ell']$ are vertices of $G_i(t)$ and $\ell < \ell'$ then (v, v') is an edge of $G_i(t)$.

(3) $G_i(t)$ is transitively closed: if (v, v') and (v', v'') are edges of $G_i(t)$, then (v, v'') is also an edge of $G_i(t)$.

(4) For all correct processes p_j, there is a time $t' \geq t$, a $d \in \mathcal{R}_{\mathcal{D}}$ and a $\ell \in \mathbb{N}$ such that, for every vertex v of $G_i(t)$, $(v, [p_j, d, \ell])$ is an edge of $G_i(t')$.

(5) For all correct processes p_j, there is a time $t' \geq t$ such that $G_i(t)$ is a subgraph of $G_j(t')$.

The properties imply that ever-growing DAGs at correct processes tend to the same infinite DAG G: $\lim_{t\to\infty} G_i(t) = G$. In a fair run of the algorithm in Figure 1, the set of processes that obtain infinitely many vertices in G is the set of correct processes [4].

3.3 Asynchronous simulation

It is shown below that *any* infinite DAG G constructed as shown in Figure 1 can be used to simulate partial runs of $\mathcal{A}$ in the *asynchronous* manner: instead of querying $\mathcal{D}$, the simulation algorithm $\mathcal{A}'$ uses the samples of the failure detector output captured in the DAG. The pseudo-code of this simulation is presented in Figure 2. The algorithm is hypothetical in the sense that it uses an infinite input, but this requirement is relaxed later.

In the algorithm, each process p_i is initially associated with an initial state of $\mathcal{A}$ and performs a sequence of simulated steps of $\mathcal{A}$. Every process p_i maintains a shared register V_i that stores the vertex of G used for the most recent step of $\mathcal{A}$ simulated by p_i. Each time p_i is about to perform a step of $\mathcal{A}$ it first reads registers $V_1, \ldots, V_n$ to obtain the vertexes of G used by processes $p_1, \ldots, p_n$ for simulating the most recent causally preceding steps of $\mathcal{A}$ (line 10 in Figure 2). Then p_i selects the next vertex of G that succeeds all vertices (lines 11-14). If no such vertex is found, p_i blocks forever (line 13).

Note that a correct process p_i may block forever if G contains only finitely many vertices of p_i. As a result an infinite run of $\mathcal{A}'$ may simulate an *unfair* run of $\mathcal{A}$: a run in which some correct process takes only finitely many steps. But every finite run simulated by $\mathcal{A}'$ is a partial run of $\mathcal{A}$.

Theorem 1 *Let G be the DAG produced in a fair run $R = \langle F, H, I, S, T\rangle$ of the communication component in Figure 1. Let $R' = \langle F', H', I', S', T'\rangle$ be any fair run of $\mathcal{A}'$ using G. Then the sequence of steps simulated by $\mathcal{A}'$ in R' belongs to a (possibly unfair) run of $\mathcal{A}$, $R_{\mathcal{A}}$, with input vector of I' and failure pattern F. Moreover, the set of processes that take infinitely many steps in $\mathcal{R}_{\mathcal{A}}$ is correct$(F) \cap$ correct(F'), and if correct$(F) \subseteq$ correct(F'), then $R_{\mathcal{A}}$ is fair.*

Proof. Recall that a step of a process p_i can be either a *memory* step in which p_i accesses shared memory or a *query* step in which p_i queries the failure detector. Since memory steps simulated in $\mathcal{A}'$ are performed as in $\mathcal{A}$, to show that algorithm $\mathcal{A}'$ indeed simulates a run of $\mathcal{A}$ with failure pattern F, it is enough to make sure that the sequence of simulated *query* steps in the simulated run (using vertices of G) *could have been observed* in a run $R_{\mathcal{A}}$ of $\mathcal{A}$ with failure pattern F and the input vector based on I'.

```
   Shared variables:
      $V_1, \ldots, V_n := \perp, \ldots, \perp$,
            {for each $p_j$, $V_j$ is the vertex of $G$
            corresponding to the latest simulated step of $p_j$}
      Shared variables of $\mathcal{A}$

7  initialize the simulated state of $p_i$ in $\mathcal{A}$, based on $I'$
8  $\ell := 0$
9  while $true$ do
      {Simulating the next $p_i$'s step of $\mathcal{A}$}
10    $U := [V_1, \ldots, V_n]$
11    repeat
12       $\ell := \ell + 1$
13       wait until $G$ includes $[p_i, d, \ell]$ for some $d$
14    until $\forall j, U[j] \neq \perp$: $(U[j], [p_i, d, \ell]) \in G$
15    $V_i := [p_i, d, \ell]$
16    take the next $p_i$'s step of $\mathcal{A}$ using $d$ as the output of $\mathcal{D}$
```

Figure 2. DAG-based asynchronous algorithm $\mathcal{A}'$: code for each p_i

Let τ be a map associated with G that carries each vertex of G to an element in $\mathbb{T}$ such that (a) for any vertex $v = [p, d, \ell]$ of G, $p \notin F(\tau(v))$ and $d = H(p_j, \tau(v))$, and (b) for every edge (v, v') of G, $\tau(v) < \tau(v')$ (the existence of τ is established by property (5) of DAGs in Section 3.2). For each step s simulated by $\mathcal{A}'$ in $\mathcal{R}'$, let $\tau'(s)$ denote time when step *s occurred* in $\mathcal{R}'$, i.e., when the corresponding line 16 in Figure 2 was executed, and $v(s)$ be the vertex of G used for simulating s, i.e., the value of V_i when p_i simulates s in line 16 of Figure 2.

Consider query steps s_i and s_j simulated by processes p_i and p_j, respectively. Let $v(s_i) = [p_i, d_i, \ell]$ and $v(s_j) = [p_j, d_j, m]$. WLOG, suppose that $\tau([p_i, d_i, \ell]) < \tau([p_j, d_j, m])$, i.e., $\mathcal{D}$ outputs d_i at p_i before outputting d_j at p_j.

If $\tau'(s_i) < \tau'(s_j)$, i.e., s_i is simulated by p_i before s_j is simulated by p_j, then the order in which s_i and s_j see value d_i and d_j is the run produced by $\mathcal{A}'$ is consistent with the output of $\mathcal{D}$, i.e., the values d_i and d_j indeed could have been observed in that order.

Suppose now that $\tau'(s_i) > \tau'(s_j)$. If s_i and s_j are not causally related in the simulated run, then R' is indistinguishable from a run in which s_i is simulated by p_i *before* s_j is simulated by p_j. Thus, s_i and s_j can still be observed in a run of A.

Now suppose, by contradiction that $\tau'(s_i) > \tau'(s_j)$ and s_j causally precedes s_i in the simulated run, i.e., p_j simulated at least one write step s'_j after s_j, and p_i simulated at least one read step s'_i before s_i, such that s'_j took place before s'_i in R'. Since before performing the memory access of s'_j, p_j updated V_j with a vertex $v(s'_j)$ that succeeds $v(s_j)$ in G (line 15), and s'_i occurs in R' after s'_j, p_i must have found $v(s'_j)$ or a later vertex of p_j in V_j before simulating step s_i (line 10) and, thus, the vertex of G used for simulating s_i must be a descendant of $[p_j, d_j, m]$, and, by properties (1) and (3) of DAGs (Section 3.2), $\tau([p_i, d_i, \ell]) > \tau([p_j, d_j, m])$ — a contradiction. Hence, the sequence of steps of $\mathcal{A}$ simulated in R' could have been observed in a run $R_{\mathcal{A}}$ of $\mathcal{A}$ with failure pattern F.

Since in $\mathcal{A}'$, a process simulates only its own steps of $\mathcal{A}$, every process that appears infinitely often in $R_{\mathcal{A}}$ is in *correct*(F'). Also, since each faulty in F process contains only finitely many vertices in G, eventually, each process in *correct*(F') − *correct*(F) is blocked in line 13 in Figure 2, and, thus, every process that appears infinitely often in $R_{\mathcal{A}}$ is also in $\mathit{correct}(F)$. Now consider a process $p_i \in \mathit{correct}(F') \cap \mathit{correct}(F)$. Property (4) of DAGs implies that for every set V of vertices of G, there exists a vertex of p_i in G such that for all $v' \in V$, (v', v) is an edge in G. Thus, the wait statement in line 13 cannot block p_i forever, and p_i takes infinitely many steps in $R_{\mathcal{A}}$.

Hence, the set of processes that appear infinitely often in $R_{\mathcal{A}}$ is exactly $\mathit{correct}(F') \cap \mathit{correct}(F)$. Specifically, if $\mathit{correct}(F) \subseteq \mathit{correct}(F')$, then the set of processes that appear infinitely often in $R_{\mathcal{A}}$ is $\mathit{correct}(F)$, and the run is fair. $\square$

Note that in a fair run, the properties of the algorithm in Figure 2 remain the same if the infinite DAG G is replaced with a finite ever-growing DAG $\bar{G}$ constructed in parallel (Figure 1) such that $\lim_{t \to \infty} \bar{G} = G$. This is because such a replacement only affects the wait statement in line 13 which blocks p_i until the first vertex of p_i that causally succeeds every simulated step recently "witnessed" by p_i is found in G, but this cannot take forever if p_i is correct (properties (4) and (5) of DAGs in Section 3.2). The wait blocks forever if the vertex is absent in G, which may happen only if p_i is faulty.

3.4 BG-simulation

Borowsky and Gafni proposed in [1, 3], a simulation technique by which $k+1$ *simulators* $q_1, \ldots, q_{k+1}$ can wait-free simulate a k-resilient execution of any asynchronous n-process protocol. Informally, the simulation works as follows. Every process q_i tries to simulate steps of all n processes $p_1, \ldots, p_n$ in a round-robin fashion. Simulators run an *agreement protocol* to make sure that every step is simulated at most once. Simulating a step of a given process may block forever if and only if some simulator has crashed in the middle of the corresponding agreement protocol. Thus, even if k out of $k+1$ simulators crash, at least $n-k$ simulated processes can still make progress. The simulation thus guarantees at least $n-k$ processes in $\{p_1, \ldots, p_n\}$ accept infinitely many simulated steps.

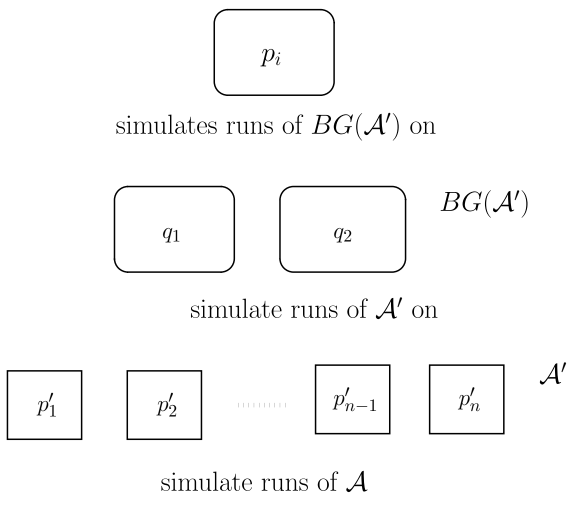
In the computational component of the reduction algorithm, the BG-simulation technique is used as follows. Let $BG(\mathcal{A}')$ denote the simulation protocol for 2 processes q_1 and q_2 which allows them to simulate, in a wait-free manner, a 1-resilient execution of algorithm $\mathcal{A}'$ for n processes $p_1, \ldots, p_n$. The complete reduction algorithm thus employs a *triple* simulation (Figure 3): every process p_i simulates multiple runs of two processes q_1 and q_2 that use BG-simulation to produce a 1-resilient run of $\mathcal{A}'$ on processes $p'_1, \ldots, p'_n$ in which steps of the original algorithm $\mathcal{A}$ are periodically simulated using (ever-growing) DAGs $G_1, ..., G_n$. (To avoid confusion, we use p'_j to denote the process that models p_j in a run of $\mathcal{A}'$ simulated by a "real" process p_i.)

We are going to use the following property which is trivially satified by BG-simulation:

(BG0) A run of BG-simulation in which every simulator take infinitely many steps simulates a run in which every simulated process takes infinitely many steps.

3.5 Using consensus

The triple simulation we are going to employ faces one complication though. The simulated runs of the asynchronous algorithm $\mathcal{A}'$ may vary depending on which process the simulation is running. This is because $G_1, ..., G_n$ are maintained by a parallel computation component (Figure 1), and a process simulating a step of $\mathcal{A}'$ may perform a different number of cycles reading the current version of its DAG before a vertex with desired properties is located (line 13 in Figure 2). Thus, the same sequence of steps of q_1 and q_2 simulated at different processes may result in different 1-resilient runs of $\mathcal{A}'$: waiting until a vertex $[p_i, d, \ell]$ appears



p_i
simulates runs of $BG(\mathcal{A}')$ on
q_1
q_2
$BG(\mathcal{A}')$
simulate runs of $\mathcal{A}'$ on
p'_1
p'_2
p'_{n-1}
p'_n
$\mathcal{A}'$
simulate runs of $\mathcal{A}$

Figure 3. Three levels of simulation: real processes p_i simulate a system of two BG-simulators q_1 and q_2 that run $BG(\mathcal{A}')$ to simulate an $(n-1)$-resilient run of $\mathcal{A}'$ on $p'_1, \ldots, p'_n$.

```
$r := 0$
**repeat**
    $r := r + 1$
    **if** $G$ contains $[p_i, d, \ell]$ for some $d$ **then** $u := 1$
    **else** $u := 0$
    $v := \mathsf{cons}_r^{i,\ell}.propose(u)$
**until** $v = 1$
```

Figure 4. Expanded line 13 of Figure 2: waiting until G includes a vertex $[p_i, d, \ell]$ for some d. Here G is any DAG generated by the algorithm in Figure 1.

in G_j at process p_j may take different number of local steps checking G_j, depending on the time when p_j executes the wait statement in line 13 of Figure 2.

To resolve this issue, the wait statement is implemented using a series of consensus instances $\mathsf{cons}_1^{i,\ell}$, $\mathsf{cons}_2^{i,\ell}$, … (Figure 4). If p_i is correct, then eventually each correct process will have a vertex $[p_i, d, \ell]$ in its DAG and, thus, the code in Figure 4 is non-blocking, and Theorem 1 still holds. Furthermore, the use of consensus ensures that if a process, while simulating a step of $\mathcal{A}'$ at process p_i, went through r steps before reaching line 14 in Figure 2, then every process simulating this very step does the same. Thus, a given sequence of steps of q_1 and q_2 will result in the same simulated 1-resilient run of $\mathcal{A}'$, regardless of when and where the simulation is taking place.

3.6 Extracting Ω

The computational component of the reduction algorithm is presented in Figure 5. In the component, every process p_i locally simulates multiple runs of a system of 2 processes q_1 and q_2 that run algorithm $BG(\mathcal{A}')$, to produce a 1-resilient run of $\mathcal{A}'$ (Figures 2 and 4). Recall that $\mathcal{A}'$, in its turn, simulates a run of

```
17 for all binary 2-vectors $J_0$ do
       { For all possible consensus inputs for $q_1$ and $q_2$ }
18   $\sigma_0$ := the empty string
19   explore($J_0, \sigma_0$)

20 function explore($J, \sigma$)
21   for all $q_j = q_1, q_2$ do
22     $\rho$ := empty string
23     repeat
24       $\rho := \rho \cdot q_j$
25       let $p'_\ell$ be the process that appears the least in $SCH_{\mathcal{A}'}(J, \sigma \cdot \rho)$
26       $\Omega$-output := $p_\ell$
27     until $ST_{\mathcal{A}}(J, \sigma \cdot \rho)$ is decided
28   explore($J, \sigma \cdot q_1$)
29   explore($J, \sigma \cdot q_2$)
```

Figure 5. Computational component of the reduction algorithm: code for each process p_i. Here $ST_{\mathcal{A}}(J, \sigma)$ denotes the state of $\mathcal{A}$ reached by the partial run of $\mathcal{A}'$ simulated in the partial run of $BG(\mathcal{A}')$ with schedule σ and input state J, and $SCH_{\mathcal{A}'}(J, \sigma)$ denotes the corresponding schedule of $\mathcal{A}'$.

the original algorithm $\mathcal{A}$, using, instead of $\mathcal{D}$, the values provided by an ever-growing DAG G. In simulating the part of $\mathcal{A}'$ of process p'_i presented in Figure 4, q_1 and q_2 count each access of a consensus instance $\mathsf{cons}_r^{i,\ell}$ as *one local step* of p'_i that need to be simulated. Also, in $BG(\mathcal{A}')$, when q_j is about to simulate the first step of p'_i, q_j uses its own input value as an input value of p'_i.

For each simulated state S of $BG(\mathcal{A}')$, p_i periodically checks whether the state of $\mathcal{A}$ in S is *deciding*, i.e., whether some process has decided in the state of $\mathcal{A}$ in S. As we show, eventually, the same infinite non-deciding 1-resilient run of $\mathcal{A}'$ will be simulated by all processes, which allows for extracting the output of Ω.

The algorithm in Figure 5 explores *solo* extensions of q_1 and q_2 starting from growing prefixes. Since, by property (BG0) of BG-simulation (Section 3.4), a run of $BG(\mathcal{A}')$ in which both q_1 and q_2 participate infinitely often simulates a run of $\mathcal{A}'$ in which every $p_j \in \{p'_1, \ldots, p'_n$ participates infinitely often, and, by Theorem 1, such a run will produce a fair and thus deciding run of $\mathcal{A}$. Thus, if there is an infinite non-deciding run simulated by the algorithm in Figure 2, it must be a run produced by a solo extension of q_1 or q_2 starting from some finite prefix.

Lemma 2 *The algorithm in Figure 5 eventually forever executes lines 23–27.*

Proof. Consider any run of the algorithm in Figures 1, 4 and 5. Let F be the failure pattern of that run. Let G be the infinite limit DAG approximated by the algorithm in Figure 1. By contradiction, suppose that lines 23–27 in Figure 5 never block p_i.

Suppose that for some initial J_0, the call of *explore*(J_0, σ_0) performed by p_i in line 19 never returns. Since the cycle in lines 23–27 in Figure 5 always terminates, there is an infinite sequence of recursive calls *explore*(J_0, σ_0), *explore*(J_0, σ_1), *explore*(J_0, σ_2), ..., where each σ_ℓ is a one-step extension of $\sigma_{\ell-1}$. Thus, there exists an infinite never deciding schedule $\tilde{\sigma}$ such that the run of $BG(\mathcal{A}')$ based on $\tilde{\sigma}$ and J_0 produces a never-deciding run of $\mathcal{A}'$. Suppose that both q_1 and q_2 appear in $\tilde{\sigma}$ infinitely often. By property (BG0) of BG-simulation (Section 3.4), a run of $BG(\mathcal{A}')$ in which both q_1 and q_2 participate infinitely often simulates a run of $\mathcal{A}'$ in which every $p_j \in \{p'_1, \dots, p'_n\}$ participates infinitely often, and, by Theorem 1, such a run will produce a fair and thus deciding run of $\mathcal{A}$ — a contradiction.

Thus, if there is an infinite non-deciding run simulated by the algorithm in Figure 2, it must be a run produced by a solo extension of q_1 or q_2 starting from some finite prefix. Let $\bar{\sigma}$ be the first such prefix in the order defined by the algorithm in Figure 2 and q_ℓ be the first process whose solo extension of σ is never deciding. Since the cycle in lines 23–27 always terminates, the recursive exploration of finite prefixes σ in lines 28 and 29 eventually reaches $\bar{\sigma}$, the algorithm reaches line 22 with $\sigma = \bar{\sigma}$ and $q_j = q_\ell$. Then the succeeding cycle in lines 23–27 never terminates — a contradiction.

Thus, for all inputs J_0, the call of *explore*(J_0, σ_0) performed by p_i in line 19 returns. Hence, for every finite prefix σ, any solo extension of σ produces a finite deciding run of $\mathcal{A}$. We establish a contradiction, by deriving a wait-free algorithm that solves consensus among q_1 and q_2.

Let $\tilde{G}$ be the infinite limit DAG constructed in Figure 1. Let β be a map from vertices of $\tilde{G}$ to $\mathbb{N}$ defined as follows: for each vertex $[p_i, d, \ell]$ in G, $\beta([p_i, d, \ell])$ is the value of variable r at the moment when any run of $\mathcal{A}'$ (produced by the algorithm in Figure 2) exits the cycle in Figure 4, while waiting until $[p_i, d, \ell]$ appears in G. If there is no such run, $\beta([p_i, d, \ell])$ is set to 0. Note that the use of consensus implies that if in any simulated run of $\mathcal{A}'$, $[p_i, d, \ell]$ has been found after r iterations, then $\beta([p_i, d, \ell]) = r$, i.e., β is well-defined.

Now we consider an asynchronous read-write algorithm $\mathcal{A}'_\beta$ that is defined exactly like $\mathcal{A}'$, but instead of going through the consensus invocations in Figure 4, $\mathcal{A}'_\beta$ performs $\beta([p_i, d, \ell])$ *local* steps. Now consider the algorithm $BG(\mathcal{A}'_\beta)$ that is defined exactly as $BG(\mathcal{A}')$ except that in $BG(\mathcal{A}'_\beta)$, q_1 and q_2 BG-simulate runs of $\mathcal{A}'_\beta$. For every sequence σ of steps of q_1 and q_2, the runs of $BG(\mathcal{A}')$ and $BG(\mathcal{A}'_\beta)$ agree on the sequence of steps of $p'_1, \dots, p'_n$ in the corresponding runs of $\mathcal{A}'$ and $\mathcal{A}'_\beta$, respectively. Moreover, they agree on the runs of $\mathcal{A}$ resulting from these runs of $\mathcal{A}'$ and $\mathcal{A}'_\beta$. This is because the difference between $\mathcal{A}'$ and $\mathcal{A}'_\beta$ consist only in the local steps and does not affect the simulated state of $\mathcal{A}$.

We say that a sequence σ of steps of q_1 and q_2 is *deciding with* J_0, if, when started with J_0, the run of $BG(\mathcal{A}'_\beta)$ produces a deciding run of $\mathcal{A}$. By our hypothesis, every eventually solo schedule σ is deciding for each input J_0. As we showed above, every schedule in which both q_1 and q_2 appear sufficiently often is deciding by property (BG0) of BG-simulation. Thus, every schedule of $BG(\mathcal{A}'_\beta)$ is deciding for all inputs.

Consider the trees of all deciding schedules of $BG(\mathcal{A}'_\beta)$ for all possible inputs J_0. All these trees have finite branching (each vertex has at most 2 descendants) and finite paths. By König's lemma, the trees are finite. Thus, the set of vertices of $\tilde{G}$ used by the runs of $\mathcal{A}'$ simulated by deciding schedules of $BG(\mathcal{A}'_\beta)$ is also finite. Let $\bar{G}$ be a finite subgraph of $\tilde{G}$ that includes all vertices of $\tilde{G}$ used by these runs.

Finally, we obtain a wait-free consensus algorithm for q_1 and q_2 that works as follows. Each q_j runs $BG(\mathcal{A}'_\beta)$ (using a finite graph $\bar{G}$) until a decision is obtained in the simulated run of $\mathcal{A}$. At this point, q_j returns the decided value. But $BG(\mathcal{A}'_\beta)$ produces only deciding runs of $\mathcal{A}$, and each deciding run of $\mathcal{A}$ solves consensus for inputs provided by q_1 and q_2 — a contradiction. □

Theorem 3 *In all environments $\mathcal{E}$, if a failure detector $\mathcal{D}$ can be used to solve consensus in $\mathcal{E}$, then Ω is weaker than $\mathcal{D}$ in $\mathcal{E}$.*

Proof. Consider any run of the algorithm in Figures 1, 4 and 5 with failure pattern F.

By Lemma 2, at some point, every correct process p_i gets stuck in lines 23–27 simulating longer and longer q_j-solo extension of some finite schedule σ with input J_0. Since, processes $p_1, \dots, p_n$ use a series of consensus instances to simulate runs of $\mathcal{A}'$ in exactly the same way, the correct processes eventually agree on σ and q_j.

Let e be the sequence of process identifiers in the 1-resilient execution of $\mathcal{A}'$ simulated by q_1 and q_2 in schedule $\sigma \cdot (q_j)$ with input J_0. Since a 2-process BG-simulation produces a 1-resilient run of $\mathcal{A}'$, at least $n-1$ simulated processes in $p'_1, \ldots, p'_n$ appear in e infinitely often. Let U ($|U| \geq n-1$) be the set of such processes.

Now we show that exactly one correct (in F) process appears in e only finitely often. Suppose not, i.e., $correct(F) \subseteq U$. By Theorem 1, the run of $\mathcal{A}'$ simulated a far run of $\mathcal{A}$, and, thus, the run must be deciding — a contradiction. Since $|U| \geq n-1$, exactly one process appears in the run of $\mathcal{A}'$ only finitely often. Moreover, the process is correct.

Thus, eventually, the correct processes in F stabilize at simulating longer and longer prefixes of the same infinite non-deciding 1-resilient run of $\mathcal{A}'$. Eventually, the same correct process will be observed to take the least number of steps in the run and output in line 26 — the output of Ω is extracted. $\square$

4 Related Work

Chandra et al. derived the first "weakest failure detector" result by showing that Ω is necessary to solve consensus in the message-passing model in their fundamental paper [4]. The result was later generalized to the read-write shared memory model [12, 10]. [1]

The technique presented in this paper builds atop two fundamental results. The first is the celebrated BG-simulation [1, 3] that allows $k+1$ processes simulate, in a wait-free manner, a k-resilient run of any n-process asynchronous algorithm. The second is a brilliant observation made by Zieliński [15] that any run of an algorithm $\mathcal{A}$ using a failure detector $\mathcal{D}$ induces an *asynchronous* algorithm that simulates (possibly unfair) runs of $\mathcal{A}$. The recursive structure of the algorithm in Figure 5 is also borrowed from [15]. Unlike [15], however, the reduction algorithm of this paper assumes the conventional read-write memory model without using immediate snapshots [2]. Also, instead of growing "precedence" and "detector" maps of [15], this paper uses directed acyclic graphs á la [4].

5 Concluding Remarks

This paper presents a new proof that Ω is the weakest failure detector to solve consensus in read-write shared memory models. The proof applies a novel reduction technique, and is based on the very fact that wait-free 2-process consensus is impossible, unlike the original technique of [4] that partially rehashes elements of the consensus impossibility proof.

Several recent papers focused on determining the weakest failure detector for a generalization of consensus, (n, k)-set agreement, in which n processes have to decide on at most k distinct proposed values [14, 6, 9]. For $k = 1$, the problem is consensus, and the weakest failure detector for $(n, 1)$-set agreement is Ω. Zieliński [15] determined the weakest failure detector for $(n, n-1)$-set agreement (sometimes called simply set agreement in the literature). However, the case $1 < k < n$ remained unresolved until recently.

As we show in the forthcoming paper [8], the technique proposed in this paper can be applied to determine the weakest failure detector for (n, k)-set agreement for any $1 \leq k \leq n$. Intuitively, BG simulation allows $k+1$ processes to simulate a k-resilient run of any asynchronous algorithm, and, generalizing the technique described in this paper, we can derive an infinite non-deciding k-resilient run R of $\mathcal{A}'$. At least one correct process appears only finitely often in R (otherwise, the run would be deciding). Thus, a failure detector that

[1]The result for the shared memory as stated in [12], but the only published proof of it appears in [10].

periodically outputs the latest $n - k$ processes in growing prefixes of R guarantees that eventually some correct process is never output. It can be easily shown that this information about failures is sufficient to solve (n, k)-set agreement. For $k > 1$, we cannot use consensus to make sure that correct processes simulate runs of $\mathcal{A}'$ in exactly the same way, regardless of how their local DAGs evolve. Therefore, our generalized reduction algorithm employs a slightly more sophisticated "eventual agreement" mechanism to make sure that the simulation converges.

Acknowledgements

I am grateful to Eli Gafni for the crucial observation that a reduction algorithm can directly use consensus objects to help processes agree on the simulated executions of $\mathcal{A}'$.

References

[1] Elizabeth Borowsky and Eli Gafni. Generalized FLP impossibility result for t-resilient asynchronous computations. In *STOC*, pages 91–100. ACM Press, May 1993.

[2] Elizabeth Borowsky and Eli Gafni. Immediate atomic snapshots and fast renaming. In *PODC*, pages 41–51, New York, NY, USA, 1993. ACM Press.

[3] Elizabeth Borowsky, Eli Gafni, Nancy A. Lynch, and Sergio Rajsbaum. The BG distributed simulation algorithm. *Distributed Computing*, 14(3):127–146, 2001.

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

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

[6] Wei Chen, Jialin Zhang, Yu Chen, and Xuezheng Liu. Weakening failure detectors for -set agreement via the partition approach. In *DISC*, pages 123–138, 2007.

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

[8] Eli Gafni and Petr Kuznetsov. The weakest failure detector for solving k-set agreement. In submission, February 2009.

[9] Rachid Guerraoui, Maurice Herlihy, Petr Kouznetsov, Nancy A. Lynch, and Calvin C. Newport. On the weakest failure detector ever. In *PODC*, pages 235–243, August 2007.

[10] Rachid Guerraoui and Petr Kouznetsov. Failure detectors as type boosters. *Distributed Computing*, 20(5):343–358, 2008.

[11] Prasad Jayanti and Sam Toueg. Every problem has a weakest failure detector. In *PODC*, pages 75–84, 2008.

[12] Wai-Kau Lo and Vassos Hadzilacos. Using failure detectors to solve consensus in asynchronous shared memory systems. In *WDAG*, LNCS 857, pages 280–295, September 1994.

[13] M.C. Loui and H.H. Abu-Amara. Memory requirements for agreement among unreliable asynchronous processes. *Advances in Computing Research*, 4:163–183, 1987.

[14] Michel Raynal and Corentin Travers. In search of the holy grail: Looking for the weakest failure detector for wait-free set agreement. In *OPODIS*, pages 3–19, 2006.

[15] Piotr Zieliński. Anti-omega: the weakest failure detector for set agreement. In *PODC*, August 2008.