The Weakest Failure Detector for Solving Consensus*

Tushar Deepak Chandra**+
Vassos Hadzilacos***
Sam Toueg+

TR 92-1293
July 1992

Department of Computer Science
Cornell University
Ithaca, NY 14853-7501

*Research supported by NSF grants CCR-8901780 and CCR-9102231, DARPA/NASA Ames grant NAG 2-593, grants from the IBM Endicott Programming Laboratory and Siemens Corp., and a grant from the Natural Sciences and Engineering Research Council of Canada.
**Also supported by an IBM graduate fellowship.
***Computer Systems Research Institute, University of Toronto, 6 King's Colege Road, Toronto, Ontario M5S 1A1.
+Dept. of Computer Science, Upson Hall, Cornell University, Ithaca, NY 14853.

The Weakest Failure Detector for Solving Consensus*

Tushar Deepak Chandra†§ Vassos Hadzilacos‡ Sam Toueg§

July 18, 1992

Abstract

We determine what information about failures is necessary and sufficient to solve Consensus in asynchronous distributed systems subject to crash failures. In [CT91], we proved that $\diamond\mathcal{W}$, a failure detector that provides surprisingly little information about which processes have crashed, is sufficient to solve Consensus in asynchronous systems with a majority of correct processes. In this paper, we prove that to solve Consensus, any failure detector has to provide at least as much information as $\diamond\mathcal{W}$. Thus, $\diamond\mathcal{W}$ is indeed the weakest failure detector for solving Consensus in asynchronous systems with a majority of correct processes.

1 Introduction

1.1 Background

The asynchronous model of distributed computing has been extensively studied. Informally, an *asynchronous distributed system* is one in which message transmission times and relative processor speeds are both unbounded. Thus an algorithm designed for an asynchronous system does not rely on such bounds for its correctness. In practice, asynchrony is introduced by unpredictable loads on the system.

Although the asynchronous model of computation is attractive for the reasons outlined above, it is well-known that many fundamental problems of fault-tolerant distributed computing that are solvable in synchronous systems, are unsolvable in asynchronous systems. In particular, it is well-known that *Consensus*, and several forms

*Research supported by NSF grants CCR-8901780 and CCR-9102231, DARPA/NASA Ames grant NAG-2-593, grants from the IBM Endicott Programming Laboratory and Siemens Corp, and a grant from the Natural Sciences and Engineering Research Council of Canada.

†Also supported by an IBM graduate fellowship.

‡Computer Systems Research Institute, University of Toronto, 6 King's College Road, Toronto, Ontario M5S 1A1

§Dept. of Computer Science, Upson Hall, Cornell University, Ithaca, NY 14853

of reliable broadcast, including *Atomic Broadcast*, cannot be solved deterministically in an asynchronous system that is subject to even a single *crash* failure [FLP85,DDS87]. Essentially, these impossibility results stem from the inherent difficulty of determining whether a process has actually crashed or is only "very slow".

To circumvent these impossibility results, previous research focused on the use of randomization techniques [CD89], the definition of some weaker problems and their solutions [DLP$^+$86,ABD$^+$87,BW87,BMZ88], or the study of several models of *partial synchrony* [DDS87,DLS88]. However, the impossibility of deterministic solutions to many *agreement* problems (such as Consensus and Atomic Broadcast) remains a major obstacle to the use of the asynchronous model of computation for fault-tolerant distributed computing.

An alternative approach to circumvent such impossibility results is to augment the asynchronous model of computation with a *failure detector*. Informally, a failure detector is a distributed oracle that gives (possibly incorrect) hints about which processes may have crashed so far: Each process has access to a local *failure detector module* that monitors other processes in the system, and maintains a list of those that it currently suspects to have crashed. Each process periodically consults its failure detector module, and uses the list of suspects returned in solving Consensus.

A failure detector module can make *mistakes* by erroneously adding processes to its list of suspects: i.e., it can suspect that a process p has crashed even though p is still running. If it later believes that suspecting p was a mistake, it can remove p from its list. Thus, each module may repeatedly *add* and *remove* processes from its list of suspects. Furthermore, at any given time the failure detector modules at two different processes may have different lists of suspects.

It is important to note that the mistakes made by a failure detector should not prevent any correct process from behaving according to specification. For example, consider an algorithm that uses a failure detector to solve Atomic Broadcast in an asynchronous system. Suppose all the failure detector modules wrongly (and permanently) suspect that a correct process p has crashed. The Atomic Broadcast algorithm must still ensure that p delivers the same set of messages, in the same order, as all the other correct processes. Furthermore, if p broadcasts a message m, all correct processes must deliver m.[1]

In [CT91], we showed that a surprisingly weak failure detector is sufficient to solve Consensus and Atomic Broadcast in asynchronous systems with a majority of correct processes. This failure detector, called the *eventually weak failure detector* and denoted $\mathcal{W}$ here, satisfies only the following two properties:[2]

1. There is a time after which every process that crashes is always suspected by some correct process.

[1] A different approach was taken in [RB91]: a correct process that is wrongly suspected to have crashed, voluntarily leaves the system. It may later rejoin the system by assuming a new identity.

[2] In [CT91], this was denoted $\Diamond\mathcal{W}$.

2. There is a time after which some correct process is never suspected by any correct process.

Note that, at any given time t, processes cannot use $\mathcal{W}$ to determine the identity of a correct process. Furthermore, they cannot determine whether there is a correct process that will not be suspected after time t.

The failure detector $\mathcal{W}$ can make an *infinite* number of mistakes. In fact, it can forever add and then remove some *correct* processes from the lists of suspects (this reflects the inherent difficulty of determining whether a process is just slow or has crashed). Moreover, some correct processes may be erroneously suspected to have crashed by all the other processes throughout the entire execution.

The two properties of $\mathcal{W}$ state that eventually something must hold forever; this may appear too strong a requirement to implement in practice. However, when solving a problem that "terminates", such as Consensus, it is not really required that the properties hold *forever*, but merely that they hold for a *sufficiently long time*, i.e., long enough for the algorithm that uses the failure detector to achieve its goal. For instance, in practice the algorithm of [CT91] that solves Consensus using $\mathcal{W}$ only needs the two properties of $\mathcal{W}$ to hold for a relatively short period of time.[3] However, in an asynchronous system it is not possible to quantify "sufficiently long", since even a single process step or a single message transmission is allowed to take an arbitrarily long amount of time. Thus it is convenient to state the properties of $\mathcal{W}$ in the stronger form given above.

1.2 The problem

The failure detection properties of $\mathcal{W}$ are *sufficient* to solve Consensus in asynchronous systems. But are they *necessary*? For example, consider failure detector $\mathcal{A}$ that satisfies Property 1 of $\mathcal{W}$ and the following weakening of Property 2:

> There is a time after which some correct process is never suspected by at least 99% of the correct processes.

$\mathcal{A}$ is clearly weaker than $\mathcal{W}$. Is it possible to solve Consensus using $\mathcal{A}$? Indeed what is the *weakest* failure detector *sufficient* to solve Consensus in asynchronous systems? In trying to answer this fundamental question we run into a problem. Consider failure detector $\mathcal{B}$ that satisfies the following two properties:

1. There is a time after which every process that crashes is always suspected by *all* correct processes.

2. There is a time after which some correct process is never suspected by a majority of the processes.

[3] In that algorithm processes are cyclically elected as "coordinators". Consensus is achieved as soon as a correct coordinator is reached, and no process suspects it to have crashed while this coordinator is trying to enforce consensus.

It seems that $\mathcal{B}$ and $\mathcal{W}$ are *incomparable*: $\mathcal{B}$'s first property is stronger than $\mathcal{W}$'s, and $\mathcal{B}$'s second property is weaker than $\mathcal{W}$'s. Is it possible to solve Consensus in an asynchronous system using $\mathcal{B}$? The answer turns out to be "yes" (provided that this asynchronous system has a majority of correct processes, as $\mathcal{W}$ also requires). Since $\mathcal{W}$ and $\mathcal{B}$ appear to be incomparable, one may be tempted to conclude that $\mathcal{W}$ cannot be the "weakest" failure detector with which Consensus is solvable. Even worse, it raises the possibility that no such "weakest" failure detector exists.

However, a closer examination reveals that $\mathcal{B}$ and $\mathcal{W}$ are indeed comparable in a natural way: There is a distributed algorithm $T_{\mathcal{B}\to\mathcal{W}}$ that can transform $\mathcal{B}$ into a failure detector with the Properties 1 and 2 of $\mathcal{W}$. $T_{\mathcal{B}\to\mathcal{W}}$ works for any asynchronous system that has a majority of correct processes. We say that $\mathcal{W}$ is *reducible to* $\mathcal{B}$ in such a system. Since $T_{\mathcal{B}\to\mathcal{W}}$ is able to transform $\mathcal{B}$ into $\mathcal{W}$ in an asynchronous system, $\mathcal{B}$ must provide at least as much information about process failures as $\mathcal{W}$ does. Intuitively, $\mathcal{B}$ is at least as strong as $\mathcal{W}$.

1.3 The result

In [CT91], we showed that $\mathcal{W}$ is sufficient to solve Consensus in asynchronous systems if and only if $n > 2f$ (where n is the total number of processes, and f is the maximum number of processes that may crash). In this paper, we prove that $\mathcal{W}$ is reducible to *any* failure detector $\mathcal{D}$ that can be used to solve Consensus (this result holds for any asynchronous system). We show this reduction by giving a distributed algorithm $T_{\mathcal{D}\to\mathcal{W}}$ that transforms any such $\mathcal{D}$ into $\mathcal{W}$. Therefore, $\mathcal{W}$ is indeed the weakest failure detector that can be used to solve Consensus in asynchronous systems with $n > 2f$. Furthermore, if $n \leq 2f$, any failure detector that can be used to solve Consensus must be strictly stronger than $\mathcal{W}$.

The task of transforming any given failure detector $\mathcal{D}$ (that can be used to solve Consensus) into $\mathcal{W}$ runs into a serious technical difficulty for the following reasons:

- To strengthen our result, we do not restrict the output of $\mathcal{D}$ to lists of suspects. Instead, this output can be *any value* that encodes some information about failures. For example, a failure detector $\mathcal{D}$ should be allowed to output any boolean formula, such as "(not p) and (q or r)" (i.e., p is up and either q or r has crashed)—or any *encoding* of such a formula. Indeed, the output of $\mathcal{D}$ could be an arbitrarily complex (and unknown) encoding of failure information. Our transformation from $\mathcal{D}$ into $\mathcal{W}$ must be able to decode this information.

- Even if the failure information provided by $\mathcal{D}$ is not encoded, it is not clear how to extract from it the failure detection properties of $\mathcal{W}$. Consequently, if $\mathcal{D}$ is given in isolation, the task of transforming it into $\mathcal{W}$ may not be possible.

Fortunately, since $\mathcal{D}$ can be used to solve Consensus, there is a corresponding algorithm, $Consensus_{\mathcal{D}}$, that is somehow able to "decode" the information about failures provided by $\mathcal{D}$, and knows how to use it to solve Consensus. Our reduction algorithm, $T_{\mathcal{D}\to\mathcal{W}}$ uses *Consensus*$_{\mathcal{D}}$ to extract this information from $\mathcal{D}$ and transforms it into the properties of $\mathcal{W}$.

2 The model

We describe a model of asynchronous computation with failure detection patterned after the one in [FLP85].

2.1 Failure Detectors

We assume the existence of a discrete global clock to simplify the presentation. This is merely a fictional device: the processes do not have access to it. We take the range $\mathcal{T}$ of the clock's ticks to be the set of natural numbers.

The system consists of a set of n *processes*, $\Pi = \{p_1, p_2, \ldots, p_n\}$, that may fail by crashing. A *failure pattern* F is a function from $\mathcal{T}$ to 2^{Π}, where $F(t)$ denotes the set of processes that have *crashed* through time t. Once a process crashes, it does not "recover", i.e., $\forall t : F(t) \subseteq F(t+1)$. We define $\mathit{crashed}(F) = \bigcup_{t\in\mathcal{T}} F(t)$ and $\mathit{correct}(F) = \Pi - \mathit{crashed}(F)$. If $p \in \mathit{crashed}(F)$ we say *p crashes in F* and if $p \in \mathit{correct}(F)$ we say *p is correct in F*.

Associated with each failure detector is a range $\mathcal{R}$ of values output by that failure detector. A *failure detector history H with range $\mathcal{R}$* is a function from $\Pi\times\mathcal{T}$ to $\mathcal{R}$. $H(p,t)$ is the value of the failure detector module of process p at time t. A *failure detector $\mathcal{D}$* is a function that maps each failure pattern F to a *set* of failure detector histories with range $\mathcal{R}_{\mathcal{D}}$ (where $\mathcal{R}_{\mathcal{D}}$ denotes the range of failure detector outputs of $\mathcal{D}$). $\mathcal{D}(F)$ denotes the set of possible failure detector histories permitted by $\mathcal{D}$ for the failure pattern F.

For example, consider the failure detector $\mathcal{W}$ mentioned in the introduction. Each failure detector module of $\mathcal{W}$ outputs a *set of processes* that are suspected to have crashed: in this case $\mathcal{R}_{\mathcal{W}} = 2^{\Pi}$. For each failure pattern F, $\mathcal{W}(F)$ is the set of all failure detector histories $H_{\mathcal{W}}$ with range $\mathcal{R}_{\mathcal{W}}$ that satisfy the following properties:

1. There is a time after which every process that crashes in F is always suspected by some process that is correct in F:

$$\exists t \in \mathcal{T}, \forall p \in \mathit{crashed}(F), \exists q \in \mathit{correct}(F), \forall t' \geq t : p \in H_{\mathcal{W}}(q,t')$$

2. There is a time after which some process that is correct in F is never suspected by any process that is correct in F:

$$\exists t \in \mathcal{T}, \exists p \in \mathit{correct}(F), \forall q \in \mathit{correct}(F), \forall t' \geq t : p \notin H_{\mathcal{W}}(q,t')$$

Note that we *specify* a failure detector $\mathcal{D}$ as a function of the failure pattern F of an execution. However, this does *not* preclude an *implementation* of $\mathcal{D}$ from using other aspects of the execution such as when messages are received. Thus, executions with the same failure pattern F may still have different failure detector histories. It is for this reason that we allow $\mathcal{D}(F)$ to be a *set* of failure detector histories from which the actual failure detector history for a particular execution is selected non-deterministically.

2.2 Algorithms

We model the asynchronous communication channels as a *message buffer* which contains messages of the form $(p, data, q)$ indicating that process p has sent *data* addressed to process q and q has not yet received that message. An *algorithm* A is a collection of n (possibly infinite state) deterministic automata, one for each of the processes. $A(p)$ denotes the automaton running on process p. Computation proceeds in *steps of the given algorithm* A. In each step of A, process p performs atomically the following three phases:

Receive phase: p receives a single message of the form $(q, data, p)$ from the message buffer, or a "null" message, denoted λ, meaning that no message is received by p during this step.

Failure detector query phase: p queries and receives a value from its failure detector module. We say that *p sees a value d* when the value returned by p's failure detector module is d.

Send phase: p changes its state and sends a message to all the processes according to the automaton $A(p)$, based on its state at the beginning of the step, the message received in the receive phase, and the value that p sees in the failure detector query phase.[4]

The message actually received by the process p in the receive phase is chosen *non-deterministically* from amongst the messages in the message buffer destined to p, and the null message λ. The null message may be received *even if* there are messages in the message buffer that are destined to p: the fact that m is in the message buffer merely indicates that m was sent to p. Since ours will be a model of asynchronous systems, where messages may experience arbitrary (but finite) delays, the amount of time m may remain in the message buffer before it is received is unbounded. Indeed, our model will allow a message sent later than another to be received earlier than the other. Though message delays are arbitrary, we also want them to be finite. We model this by introducing a liveness assumption: every message sent will eventually be received,

[4]In the send phase, p sends a message to all the processes atomically. As was shown in [FLP85], the ability to do so is *not* sufficient for solving Consensus. An alternative formulation of a step could restrict a process to sending a message to a single process in the send phase. We can show that both formulations are equivalent for our purposes.

provided its recipient makes "sufficiently many" attempts to receive messages. All this will be made more precise later.

We also remark that the non-determinism arising from the choice of the message to be received reflects the asynchrony of the message buffer — it is not due to non-deterministic choices made by the process. The automaton $A(p)$ is deterministic in the sense that the message that p sends in a step and p's new state are uniquely determined from the present state of p, the message p received during the step and the failure detector value seen by p during the step.

To keep things simple we assume that a process p sends a message m to q at most once. This allows us to speak of the contents of the message buffer as a set, rather than a multiset. We can easily enforce this by adding a counter to each message sent by p to q — so this assumption does not damage generality.

2.3 Configurations, Runs and Environments

A *configuration* is a pair (s, M), where s is a function mapping each process p to its local state, and M is a set of triples of the form $(q, data, p)$ representing the messages presently in the message buffer. An *initial configuration of an algorithm* A is a configuration (s, M), where $s(p)$ is an initial state of $A(p)$ and $M = \emptyset$. A *step* of a given algorithm A transforms one configuration to another. A step of A is uniquely determined by the identity of the process p that takes the step, the message m received by p during that step, and the failure detector value d seen by p during the step. Thus, we identify a step of A with a tuple (p, m, d, A). If the message received in that step is the null message, then $m = \lambda$, otherwise m is of the type $(-, -, p)$. We say that a step $e = (p, m, d, A)$ *is applicable to a configuration* $C = (s, M)$ if and only if $m \in M \cup \{\lambda\}$. We write $e(C)$ to denote the unique configuration that results when e is applied to C.

A *schedule* S *of algorithm* A is a finite or infinite sequence of steps of A. $S_\perp$ denotes the empty schedule. We say that a schedule S of an algorithm A *is applicable to a configuration* C if and only if (a) $S = S_\perp$, or (b) $S[1]$ is applicable to C, $S[2]$ is applicable to $S[1](C)$, etc.[5] If S is a finite schedule applicable to C, $S(C)$ denotes the unique configuration that results from applying S to C. Note $S_\perp(C) = C$ for all configurations C. We say that C' *is a configuration of* (S, C) if there is a prefix S' of S such that $C' = S'(C)$.

A *partial run of algorithm* A *using a failure detector* $\mathcal{D}$ is a tuple $R = \langle F, H_\mathcal{D}, I, S, T \rangle$ where F is a failure pattern, $H_\mathcal{D} \in \mathcal{D}(F)$ is a failure detector history, I is an initial configuration of A, S is a *finite* schedule of A, and T is a *finite* list of increasing time values (indicating when each step in S occurred) such that $|S| = |T|$, S is applicable to I, and for all $i \leq |S|$, if $S[i]$ is of the form (p, m, d, A) then:

- p has not crashed by time $T[i]$, i.e., $p \notin F(T[i])$
- d is the value of the failure detector module of p at time $T[i]$, i.e., $d = H_\mathcal{D}(p, T[i])$

[5] We denote by $v[i]$ the ith element of a sequence v.

Informally, a partial run of A using $\mathcal{D}$ represents a finite point of some execution of A using $\mathcal{D}$.

A *run of an algorithm A using a failure detector $\mathcal{D}$* is a tuple $R = \langle F, H_{\mathcal{D}}, I, S, T \rangle$ where F is a failure pattern, $H_{\mathcal{D}} \in D(F)$ is a failure detector history, I is an initial configuration of A, S is an *infinite* schedule of A, and T is an *infinite* list of increasing time values indicating when each step in S occurred. In addition to satisfying the above properties of a partial run, a run must also satisfy the following properties:

- Every correct process takes an infinite number of steps in S. Formally:

$$\forall p \in \mathit{correct}(F), \forall i, \exists j > i : S[j] \text{ is of the type}(p, -, -, A)$$

- Every message sent to a correct process is eventually received. Formally:

$$\forall p \in \mathit{correct}(F), \forall C = (s, M) \text{ of } (S, I) : m = (q, \mathit{data}, p) \in M \Rightarrow \\ (\exists i : S[i] \text{ is of the type } (p, m, -, A))$$

In [CT91], we proved that any algorithm that uses $\mathcal{W}$ to solve Consensus requires $n > 2f$. With other failure detectors the requirements may be different. For example, there is a failure detector that can be used to solve Consensus only if p_1 and p_2 do not both crash. In general whether a given failure detector can be used to solve Consensus depends upon assumptions about the underlying "environment". Formally, an *environment* $\mathcal{E}$ (of an asynchronous system) is set of possible failure patterns.[6]

3 The Consensus problem

In the Consensus problem, each process p has an initial value, 0 or 1, and must reach an irrevocable decision on one of these values. Thus, the algorithm of process p, $A(p)$, has two distinct *initial states* σ_0^p and σ_1^p signifying that p's initial value is 0 or 1. $A(p)$ also has two disjoint sets of *decision states* Σ_0^p and Σ_1^p. If p enters a state in Σ_k^p, we require that it remain in states in Σ_k^p, and we say that *p has decided k*.

We say that algorithm A *uses failure detector $\mathcal{D}$ to solve Consensus in environment $\mathcal{E}$* if every run $R = \langle F, H_{\mathcal{D}}, I, S, T \rangle$ of A using $\mathcal{D}$ where $F \in \mathcal{E}$ satisfies:

Termination: Each correct process eventually decides. Formally:

$$\forall p \in \mathit{correct}(F), \exists C = (s, M) \text{ of } (S, I) : s(p) \in \Sigma_0^p \cup \Sigma_1^p$$

[6] In a synchronous system, assumptions about the underlying environment may also include other characteristics such as the relative process speeds, the maximum message delay, the degree of clock synchronization, etc. In such a system, a more elaborate definition of an environment would be required.

Validity: Each correct process decides on the initial value of some process. Formally, let $I = (s_0, M_0)$:

$$\forall p \in \mathit{correct}(F), \forall k \in \{0,1\} : (\exists C = (s, M) \text{ of } (S, I) : \\ s(p) \in \Sigma_k^p) \Rightarrow (\exists q \in \Pi : s_0(q) = \sigma_k^q)$$

Agreement: No two correct processes decide differently. Formally:

$$\forall p, p' \in \mathit{correct}(F), \forall C = (s, M) \text{ of } (S, I), \forall k, k' \in \{0,1\} : \\ (s(p) \in \Sigma_k^p \wedge s(p') \in \Sigma_{k'}^{p'}) \Rightarrow k = k'$$

4 Reducibility

We now define what it means for an algorithm $T_{\mathcal{D}\to\mathcal{D}'}$ to transform a failure detector $\mathcal{D}$ into another failure detector $\mathcal{D}'$ in an environment $\mathcal{E}$. Algorithm $T_{\mathcal{D}\to\mathcal{D}'}$ uses $\mathcal{D}$ to maintain a variable output_p at every process p. This variable, reflected in the local state of p, emulates the output of $\mathcal{D}'$ at p. Let O_R be the history of all the *output* variables in run R, i.e., $O_R(p,t)$ is the value of output_p at time t in run R. Algorithm $T_{\mathcal{D}\to\mathcal{D}'}$ *transforms* $\mathcal{D}$ *into* $\mathcal{D}'$ *in* $\mathcal{E}$ if and only if for every run $R = \langle F, H_{\mathcal{D}}, I, S, T\rangle$ of $T_{\mathcal{D}\to\mathcal{D}'}$ using $\mathcal{D}$, where $F \in \mathcal{E}$, $O_R \in \mathcal{D}'(F)$.

Given $T_{\mathcal{D}\to\mathcal{D}'}$, anything that can be done using $\mathcal{D}'$ in $\mathcal{E}$, can be done using $\mathcal{D}$ instead. To see this, suppose a given algorithm B requires failure detector $\mathcal{D}'$ (when it executes in $\mathcal{E}$), but only $\mathcal{D}$ is available. We can still execute B as follows. Concurrently with B, we run $T_{\mathcal{D}\to\mathcal{D}'}$ to transform $\mathcal{D}$ into $\mathcal{D}'$. We now modify the failure detector query phase of each step of B at process p: p reads the current value of output_p (which is concurrently maintained by $T_{\mathcal{D}\to\mathcal{D}'}$) instead of querying its failure detector module. This is illustrated in Fig. 1.

Intuitively, since $T_{\mathcal{D}\to\mathcal{D}'}$ is able to use $\mathcal{D}$ to emulate $\mathcal{D}'$, $\mathcal{D}$ provides at least as much information about process failures in $\mathcal{E}$ as $\mathcal{D}'$ does. Thus, if there is an algorithm $T_{\mathcal{D}\to\mathcal{D}'}$ that transforms $\mathcal{D}$ into $\mathcal{D}'$ in $\mathcal{E}$, we write $\mathcal{D} \succeq_{\mathcal{E}} \mathcal{D}'$ and say that $\mathcal{D}'$ *is reducible to* $\mathcal{D}$ *in* $\mathcal{E}$; we also say that $\mathcal{D}'$ *is weaker than* $\mathcal{D}$ *in* $\mathcal{E}$.

5 An outline of the result

In [CT91] we showed that $\mathcal{W}$ can be used to solve Consensus in any environment in which $n > 2f$. We now show that $\mathcal{W}$ is weaker than any failure detector that can be used to solve Consensus. This result holds for any environment $\mathcal{E}$. Together with [CT91], this implies that $\mathcal{W}$ is indeed the weakest failure detector that can be used to solve Consensus in any environment in which $n > 2f$.

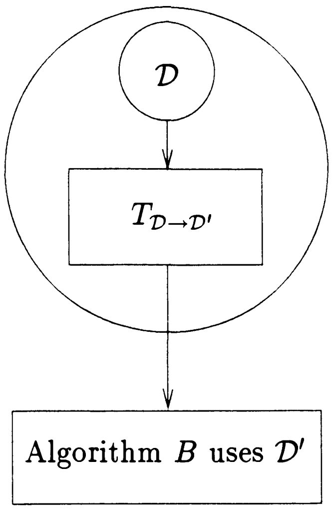

Figure 1: Transforming $\mathcal{D}$ into $\mathcal{D}'$

To prove our result, we first define a new failure detector, denoted Ω, that is at least as strong as $\mathcal{W}$. We then show that any failure detector $\mathcal{D}$ that can be used to solve Consensus is at least as strong as Ω. Thus, $\mathcal{D}$ is at least as strong as $\mathcal{W}$.

The output of the failure detector module of Ω at a process p is a *single* process, q, that p currently considers to be *correct*; we say that p *trusts* q. In this case, $\mathcal{R}_\Omega = \Pi$. For each failure pattern F, $\Omega(F)$ is the set of all failure detector histories H_Ω with range $\mathcal{R}_\Omega$ that satisfy the following property:

- There is a time after which all the correct processes always trust the same correct process:

$$\exists t \in \mathcal{T}, \exists q \in \mathit{correct}(F), \forall p \in \mathit{correct}(F), \forall t' \geq t : H_\Omega(p, t') = q$$

As with $\mathcal{W}$, the output of the failure detector module of Ω at a process p may change with time, i.e., p may trust different processes at different times. Furthermore, at any given time t, processes p and q may trust different processes.

Theorem 1: For all environments $\mathcal{E}$, $\Omega \succeq_{\mathcal{E}} \mathcal{W}$.

PROOF: [Sketch] The reduction algorithm $T_{\Omega\to\mathcal{W}}$ that transforms Ω into $\mathcal{W}$ is as follows. Each process p periodically sets $\mathit{output}_p \leftarrow \Pi - \{q\}$, where q is the process that p currently trusts according to Ω. It is easy to see that (in any environment $\mathcal{E}$) this output satisfies the two properties of $\mathcal{W}$. $\square$

Theorem 2: For all environments $\mathcal{E}$, if a failure detector $\mathcal{D}$ can be used to solve

Consensus in $\mathcal{E}$, then $\mathcal{D} \succeq_{\mathcal{E}} \Omega$.

PROOF: The reduction algorithm $T_{\mathcal{D}\rightarrow\Omega}$ is shown in Section 6. It is the core of our result. □

Corollary 3: For all environments $\mathcal{E}$, if a failure detector $\mathcal{D}$ can be used to solve Consensus in $\mathcal{E}$, then $\mathcal{D} \succeq_{\mathcal{E}} \mathcal{W}$.

PROOF: If $\mathcal{D}$ can be used to solve Consensus in $\mathcal{E}$, then, by Theorem 2, $\mathcal{D} \succeq_{\mathcal{E}} \Omega$. From Theorem 1, $\Omega \succeq_{\mathcal{E}} \mathcal{W}$. By transitivity, $\mathcal{D} \succeq_{\mathcal{E}} \mathcal{W}$. □

In [CT91] we proved that, for all environments $\mathcal{E}$ in which $n > 2f$, $\mathcal{W}$ can be used to solve Consensus. Together with Corollary 3, this shows that:

Theorem 4: For all environments $\mathcal{E}$ in which $n > 2f$, $\mathcal{W}$ is the weakest failure detector that can be used to solve Consensus in $\mathcal{E}$.

6 The reduction algorithm

Let $\mathcal{E}$ be an environment, $\mathcal{D}$ be a failure detector that can be used to solve Consensus in $\mathcal{E}$, and *Consensus*$_{\mathcal{D}}$ be the Consensus algorithm that uses $\mathcal{D}$. We describe an algorithm $T_{\mathcal{D}\rightarrow\Omega}$ that transforms $\mathcal{D}$ into Ω in $\mathcal{E}$. Intuitively, this algorithm works as follows. Fix an arbitrary run of $T_{\mathcal{D}\rightarrow\Omega}$ using $\mathcal{D}$ in $\mathcal{E}$, with failure pattern $F \in \mathcal{E}$, and failure detector history $H_{\mathcal{D}} \in \mathcal{D}(F)$. We shall first construct an infinite directed acyclic graph, denoted G, whose vertices are some of the failure detector values that occur in $H_{\mathcal{D}}$, and whose edges are consistent with the time at which these values occur. We then show that G induces a simulation forest Υ that encodes an infinite set of possible runs of *Consensus*$_{\mathcal{D}}$. Finally, we show how to extract from Υ the identity of a process p^* that is correct in F.

The induced simulation forest is infinite and thus cannot be computed by any process. However, the information needed to extract p^* is present in a *finite* subgraph of the forest. It will be sufficient for each correct process p to construct ever increasing finite approximations of the simulation forest Υ that will eventually include this crucial finite subgraph. At all times, p uses its present approximation of Υ to select the identity of some process: once p's approximation of Υ includes the crucial finite subgraph, the selected process will be p^* (forever). Thus, there is a time after which all correct processes trust the same correct process, p^*—which is exactly what Ω requires.

We say that a process is *correct* (*crashes*) if it is correct (crashes) in F. For simplicity, we assume that a process p sees a value d at most once (this can be enforced by tagging a counter to each value seen). For the rest of this paper, whenever we refer to a run of *Consensus*$_{\mathcal{D}}$, we mean a run of *Consensus*$_{\mathcal{D}}$ using $\mathcal{D}$. Furthermore, we only consider schedules of *Consensus*$_{\mathcal{D}}$, and therefore we write (p, m, d) instead of $(p, m, d, \textit{Consensus}_{\mathcal{D}})$ to denote a step.

6.1 A DAG and a forest

Given the failure pattern F and the corresponding failure detector history $H_{\mathcal{D}} \in \mathcal{D}(F)$ that were fixed above, let G be any infinite directed acyclic graph with the following properties:

1. The vertices of G are of the form $[p, d]$ where $p \in \Pi$ and $d \in \mathcal{R}_{\mathcal{D}}$. If $[p, d]$ is a vertex of G, then there is a time t such that $p \notin F(t)$ and $d = H_{\mathcal{D}}(p, t)$ (i.e., at time t, p has not crashed and the value of p's failure detector module is d).

2. If $[q_1, d_1] \to [q_2, d_2]$ is an edge of G and $d_1 = H_{\mathcal{D}}(q_1, t_1)$ and $d_2 = H_{\mathcal{D}}(q_2, t_2)$ then $t_1 < t_2$.

3. G is transitively closed.

4. Let p be any correct process and V be a finite subset of vertices of G. There is a failure detector value d such that for all vertices $[p', d']$ in V, $[p', d'] \to [p, d]$ is an edge of G.

Note that such a DAG represents only a "sampling" of the failure detector values that occur in $H_{\mathcal{D}}$. In particular, we do not require that it contain all the values that occur in $H_{\mathcal{D}}$ or that it relate (with an edge) all the values according to the time at which they occur. However, Property 4 implies that the DAG contains infinitely many "samplings" of the failure detector module of each *correct* process.

Lemma 5: Let V be any finite subset of vertices in G. G has an infinite path g such that:

- There is an edge from every vertex of V to the first vertex of g.
- If $[p, -]$ is a vertex of g then p is correct; for each correct p, there are infinitely many vertices $[p, -]$ in g.

PROOF: By repeated application of Property 4. □

Let $g = [q_1, d_1], [q_2, d_2], \ldots$ be any (finite or infinite) path of G. A schedule S is *compatible with* g if it has the same length as g, and $S = (q_1, m_1, d_1), (q_2, m_2, d_2), \ldots$, for some (possibly null) messages $m_1, m_2, \ldots$ We say that S is *compatible with* G if it is compatible with some path of G.

Let I be any initial configuration of $Consensus_{\mathcal{D}}$. We define the *simulation tree* Υ_G^I *induced by* G *and* I as follows. The vertices of Υ_G^I are the *finite* schedules S that are compatible with G and are applicable to I. The root of Υ_G^I is the empty schedule $S_\perp$. There is an edge from vertex S to vertex S' if and only if $S' = S \cdot e$ for a step e;[7] this edge is labeled e. With each (finite or infinite) path in Υ_G^I, we associate the unique schedule

[7] If u, w are sequences and v is finite then $v \cdot w$ denotes the concatenation of the two sequences.

$S = e_1, e_2, \ldots, e_k, \ldots$ consisting of the sequence of labels of the edges on that path. Note that if a path starts from the root of Υ_G^I and it is finite, the schedule S associated with it is also the last vertex of that path.

Lemma 6: S is a schedule associated with a path of Υ_G^I that starts from the root if and only if S is a schedule compatible with G and applicable to I.

PROOF: The lemma obviously holds if S is a *finite* schedule (this is immediate from the definitions). Now let $S = e_1, e_2, \ldots, e_i, \ldots$ be an *infinite* schedule, where $e_i = [q_i, m_i, d_i]$. We define $S_0 = S_\perp$, $S_1 = e_1$, $S_2 = S_1 \cdot e_2$, and in general $S_i = S_{i-i} \cdot e_i$ for all $i = 1, 2, \ldots$

Assume that S is compatible with G and applicable to I. We must show that S is a schedule associated with a path of Υ_G^I that starts from the root. To see this, note that for all $i \geq 0$, S_i is a finite schedule that is also compatible with G and applicable to I. Thus, all the schedules $S_0, S_1, S_2, \ldots, S_{i-1}, S_i, \ldots$ are vertices of Υ_G^I. Since $S_i = S_{i-1} \cdot e_i$, the edge from S_{i-1} to S_i is labeled e_i, for all $i \geq 1$. Thus, $S = e_1, e_2, \ldots, e_i, \ldots$ is the schedule associated with the infinite path $S_0 \to S_1 \to S_2 \to \ldots \to S_{i-1} \to S_i \to \ldots$ of Υ_G^I; this path starts from the root $S_0 = S_\perp$.

Assume that S is a schedule associated with an infinite path of Υ_G^I that starts from the root. We must show that S is compatible with G and is applicable to I. First note that for all i, S_i is a vertex in Υ_G^I, thus S_i is compatible with G and is applicable to I. Since $S_i = [q_1, m_1, d_1], [q_2, m_2, d_2], \ldots, [q_i, m_i, d_i]$ is compatible with G, G must contain the path $\pi_i = [q_1, d_1], [q_2, d_2], \ldots, [q_i, d_i]$ (for all i). Note that, for all i, $\pi_{i+1} = \pi_i \cdot [q_{i+1}, d_{i+1}]$ is an extension of the path π_i in G. Therefore, G contains the *infinite* path $[q_1, d_1], [q_2, d_2], \ldots, [q_i, d_i], \ldots$ So S is compatible with G. Furthermore, since all S_i's are applicable to I, by definition of applicability, the infinite schedule S is also applicable to I. Thus, S is compatible with G and applicable to I. $\square$

The following two lemmata show that the finite and infinite paths of Υ_G^I correspond to partial runs and runs of $Consensus_\mathcal{D}$ with initial configuration I.

Lemma 7: Let S be a schedule associated with a *finite* path of Υ_G^I that starts from the root. There is a sequence of times T such that $\langle F, H_\mathcal{D}, I, S, T\rangle$ is a partial run of $Consensus_\mathcal{D}$.

PROOF: By Lemma 6, S is applicable to I and compatible with G. Thus S is compatible with some finite path $g = [q_1, d_1], [q_2, d_2], \ldots, [q_i, d_i], \ldots, [q_k, d_k]$ of G. From Property 1 of G (applied to every vertex of the path g), there is a sequence $T = t_1, t_2, \ldots, t_i, \ldots, t_k$ of times such that for all i, $1 \leq i \leq k$, $d_i = H_\mathcal{D}(q_i, t_i)$ and $q_i \notin F(t_i)$. From Property 2 of G (applied to every edge of the path g), for all i, $1 \leq i < k$, $t_i < t_{i+1}$. Thus T is a sequence of increasing times, and, by definition, $\langle F, H_\mathcal{D}, I, S, T\rangle$ is a partial run of $Consensus_\mathcal{D}$. $\square$

Lemma 8: Let S be a schedule associated with an *infinite* path of Υ_G^I that starts from the root. If in S every correct process takes an infinite number of steps and every message sent to a correct process is eventually received, there is a sequence of times T such that $\langle F, H_{\mathcal{D}}, I, S, T\rangle$ is a run of $Consensus_{\mathcal{D}}$.

PROOF: Similar to Lemma 7. □

The following lemmata show some "richness" properties of the simulation trees induced by G.

Lemma 9: For any two initial configurations I and I', if S is a vertex of Υ_G^I and is applicable to I' then S is also a vertex of $\Upsilon_G^{I'}$.

PROOF: Follows directly from the definitions. □

Lemma 10: Let S be any vertex of Υ_G^I and p be any correct process. Let m be a message in the message buffer of $S(I)$ addressed to p or the null message. For some d, S has a child $S \cdot (p, m, d)$ in Υ_G^I.

PROOF: From the definition of Υ_G^I, S is compatible with some finite path g of G and applicable to I. Let v denote the last vertex of g. By Property 4, there is a d such that $v \rightarrow [p, d]$ is an edge of G. Therefore, $g \cdot [p, d]$ is a path of G, and $S \cdot (p, m, d)$ is compatible with G.

It remains to show that $S \cdot (p, m, d)$ is applicable to I. Since S is applicable to I, it suffices to show that (p, m, d) is applicable to $S(I)$. But this is true since, by hypothesis, m is in the message buffer of $S(I)$ and addressed to p, or the null message. □

Lemma 11: Let S be any vertex of Υ_G^I and p be any process. Let m be a message in the message buffer of $S(I)$ addressed to p or the null message. Let S' be a descendent of S such that, for some d, $S' \cdot (p, m, d)$ is in Υ_G^I. For each vertex S'' on the path from S to S' (inclusive), $S'' \cdot (p, m, d)$ is also in Υ_G^I.

PROOF: Since they are vertices of Υ_G^I, S, S'' and $S' \cdot (p, m, d)$ are compatible with some finite paths g, $g \cdot g''$ and $g \cdot g'' \cdot g' \cdot [p, d]$ of G, respectively. From Property 3 (transitive closure) of G, $g \cdot g'' \cdot [p, d]$ is also a path of G. So $S'' \cdot (p, m, d)$ is compatible with this path of G. We now show that $S'' \cdot (p, m, d)$ is also applicable to I, and therefore it is a vertex of Υ_G^I.

Since S'' is a vertex of Υ_G^I, S'' is applicable to I. If $m = \lambda$, then (p, m, d) is obviously applicable to $S''(I)$. Now suppose $m \neq \lambda$. Since $S' \cdot (p, m, d)$ is a vertex of Υ_G^I, (p, m, d) is applicable to $S'(I)$, and thus m is in the message buffer of $S'(I)$. Since each message is sent at most once and m is in the message buffers of $S(I)$ and $S'(I)$, there is no edge of the type $(p, m, -)$ on the path from S to S'. So m is also in the message buffer of $S''(I)$, and (p, m, d) is applicable to $S''(I)$. □

Lemma 12: Let S, S_0, and S_1 be any vertices of Υ_G^I. There is a finite schedule E containing only steps of correct processes such that:

1. $S \cdot E$ is a vertex of Υ_G^I and all correct processes have decided in $S \cdot E(I)$.

2. For $i = 0, 1$, if E is applicable to $S_i(I)$ then $S_i \cdot E$ is a vertex of Υ_G^I.

$j \leftarrow 0$
$S^0 \leftarrow S$ $\{S^0$ *is compatible with* g *and applicable to* $I\}$
repeat forever
  $j \leftarrow j + 1$
  Let $[q_j, d_j]$ be the j-th vertex of path g_∞
  Let m_j be the oldest message addressed to q_j in the message buffer of $S^{j-1}(I)$
    (if no such message exists, $m_j = \lambda$)
  $e_j \leftarrow (q_j, m_j, d_j)$
  $S^j \leftarrow S^{j-1} \cdot e_j$ $\{S^j$ *is compatible with* $g \cdot [q_1, d_1] \cdot \ldots \cdot, [q_j, d_j]$ *and applicable to* $I\}$

Figure 2: Generating schedule $S \cdot E^\infty$, compatible with path $g \cdot g_\infty$, in Υ_G^I

Proof: Since S is a vertex of Υ_G^I, S is compatible with some finite path g of G and is applicable to I. Similarly, S_0 and S_1 are compatible with some finite path g_0 and g_1, respectively, of G. From Lemma 5 (applied to the last vertices of g, g_0 and g_1), G has an infinite path $g_\infty = [q_1, d_1], [q_2, d_2], \ldots, [q_j, d_j], \ldots$ with the following two properties:

1. There is an edge from the last vertex of g, g_0 and g_1 to the first vertex of g_∞. (Thus, $g \cdot g_\infty$, $g_0 \cdot g_\infty$, and $g_1 \cdot g_\infty$ are infinite paths in G.)

2. If $[p, -]$ is a vertex of g_∞ then p is correct; for each correct p, there are infinitely many vertices $[p, -]$ in g_∞.

We now show how to construct the required schedule E. Consider the infinite sequence of schedules $S^0, S^1, S^2, \ldots, S^j, \ldots$ constructed by the algorithm in Figure 2. An easy induction shows that for all $j > 0$, S^j is applicable to I and is compatible with $g \cdot [q_1, d_1] \cdot \ldots \cdot [q_j, d_j]$, a prefix of the path $g \cdot g_\infty$ in G. So, for all $j > 0$, S^j is a vertex of Υ_G^I. Consider the infinite path of Υ_G^I that starts from the root of Υ_G^I then goes to $S^0 = S$, and then to $S^1, S^2, \ldots, S^j, \ldots$ The infinite schedule associated with that path is $S^\infty = S \cdot e_1 \cdot e_2 \cdot \ldots \cdot e_j \ldots$ Note that schedule $E^\infty = e_1 \cdot e_2 \cdot \ldots \cdot e_j \ldots$ is compatible with path g_∞ of G. By Property (2) of path g_∞, every correct process p takes an infinite number of steps in E^∞ (and thus also in $S^\infty = S \cdot E^\infty$). Since in each one of these steps p receives the oldest message that is addressed to it, every message sent to p (in S^∞) is eventually received. By Lemma 8, there is a T such that $R = \langle F, H_\mathcal{D}, I, S^\infty, T\rangle$ is a run of $Consensus_\mathcal{D}$.

From the termination requirement of Consensus, S^∞ has a finite prefix S^d such that all correct processes have decided in $S^d(I)$. There are two cases:

- S^d is a prefix of S. Since decisions are irrevocable, all correct processes remain decided in $S(I)$. Thus $S_\perp$, the empty schedule, is the required E.

- S is a prefix of S^d. Thus, $S^d = S \cdot E$ where E is a finite prefix of E^∞. Since E^∞ is compatible with g_∞, E is compatible with a prefix of g_∞. Now consider S_0 (the following argument also applies to S_1). Since S_0 is compatible with g_0, $S_0 \cdot E$ is compatible with a prefix of $g_0 \cdot g_\infty$, a path in G. So, $S_0 \cdot E$ is compatible with G. If $S_0 \cdot E$ is also applicable to I, then, by the definition of Υ_G^I, it is a vertex of Υ_G^I. The same argument holds for S_1. It remains to show that E contains only steps of correct processes. This is immediate from Property (2) of g_∞ and from the fact that E is compatible with a prefix of g_∞. □

Let I^i, $0 \leq i \leq n$, denote the initial configuration of $Consensus_{\mathcal{D}}$ in which the initial values of $p_1 \dots p_i$ are 1, and the initial values of $p_{i+1} \dots p_n$ are 0. The *simulation forest induced by G* is the set $\{\Upsilon_G^{I^0}, \Upsilon_G^{I^1}, \dots, \Upsilon_G^{I^n}\}$ of simulation trees induced by G and initial configurations $I^0, I^1, \dots, I^n$.

6.2 Tagging the simulation forest

We assign a set of *tags* to each vertex of every tree in the simulation forest induced by G. Vertex S of tree Υ_G^I gets tag k if and only if it has a descendent S' such that some *correct* process has decided k in $S'(I)$. Hereafter, Υ^i denotes the tagged tree $\Upsilon_G^{I^i}$, and Υ denotes the tagged simulation forest $\{\Upsilon^0, \Upsilon^1, \dots, \Upsilon^n\}$.

Lemma 13: Every vertex of Υ^i has at least one tag.

PROOF: From Lemma 12, every vertex S of Υ^i has a descendent $S' = S \cdot E$ (for some E) such that all correct processes have decided in $S'(I^i)$. □

A vertex of Υ^i is *monovalent* if it has only one tag, and *bivalent* if it has both tags, 0 and 1. A vertex is *0-valent* if it is monovalent and is tagged 0; *1-valent* is similarly defined.

Lemma 14: Every vertex of Υ^i is either 0-valent, 1-valent, or bivalent.

PROOF: Immediate from Lemma 13. □

Lemma 15: The ancestors of a bivalent vertex are bivalent. The descendents of a k-valent vertex are k-valent.

PROOF: Immediate from the definitions. □

Lemma 16: If vertex S of Υ^i has tag k, then no correct process has decided $1-k$ in $S(I^i)$.

PROOF: Since S has tag k, it has a descendent S' such that a correct process p has decided k in $S'(I^i)$. From Lemma 7, there is a T such that $R = \langle F, H_{\mathcal{D}}, I^i, S', T\rangle$ is a partial run of $Consensus_{\mathcal{D}}$. Since p has decided k in $S'(I^i)$, from the agreement requirement of Consensus, no correct process has decided $1-k$ in $S'(I^i)$. Since S' is a descendent of S, no correct process could have decided $1-k$ in $S(I^i)$. □

Lemma 17: If vertex S of Υ^i is bivalent, then no correct process has decided in $S(I^i)$.

PROOF: Immediate from Lemma 16. □

Recall that in I^0 all processes have initial value 0, while in I^n they all have initial value 1.

Lemma 18: The root of Υ^0 is 0-valent; the root of Υ^n is 1-valent.

PROOF: We first show that the root of Υ^0 is 0-valent. Suppose, for contradiction, that the root of Υ^0 has tag 1. There must be a vertex S of Υ^0 such that some correct process has decided 1 in $S(I^0)$. From Lemma 7, there is a T such that $R = \langle F, H_{\mathcal{D}}, I^0, S, T\rangle$ is a partial run of *Consensus*$_{\mathcal{D}}$. R violates the validity requirement of Consensus—a contradiction. Thus the root of Υ^0 cannot have a tag of 1. From Lemma 13, the root of Υ^0 has at least one tag: thus it is 0-valent.

By a symmetric argument, the root of Υ^n is 1-valent. □

Index i is *critical* if the root of Υ^i is bivalent, or if the root of Υ^{i-1} is 0-valent while the root of Υ^i is 1-valent. In the first case, we say that index i is *bivalent critical*; in the second case, we say that i is *monovalent critical.*

Lemma 19: There is a critical index i, $0 < i \leq n$.

PROOF: Apply Lemmata 14 and 18 to the roots of $\Upsilon^0, \Upsilon^1, \ldots, \Upsilon^n$. □

The critical index i is the key to extracting the identity of a correct process. In fact, if i is monovalent critical, we shall prove that p_i must be correct (Lemma 21). If i is bivalent critical, the correct process will be found by focusing on the tree Υ^i, as explained in the following section.

6.3 Of hooks and forks

We describe two types of finite subtrees of Υ^i referred to as *decision gadgets of* Υ^i. Each type of decision gadget is rooted at the root $S_\perp$ of Υ^i and has exactly two leaves: one 0-valent and one 1-valent. The least common ancestor of these leaves is called the *pivot.* The pivot is clearly bivalent.

The first type of decision gadget is called a *fork*, and is shown in Figure 3. The two leaves are children of the pivot, obtained by applying different steps of the *same* process p. Process p is the *deciding process of the fork*, because its step after the pivot determines the decision of correct processes.

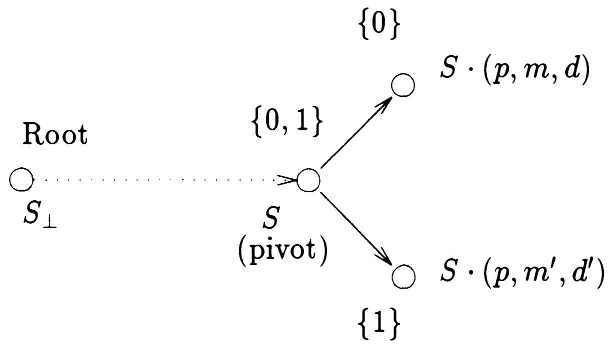

Figure 3: A fork—p is the deciding process

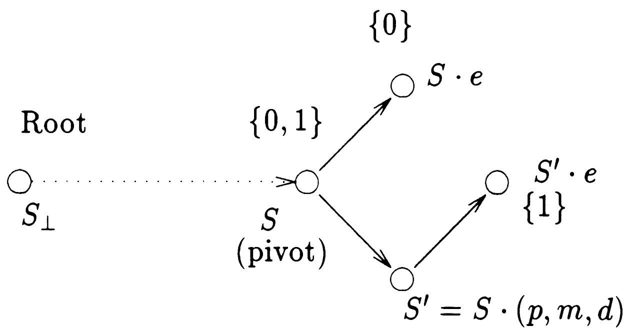

Figure 4: A hook—p is the deciding process

$S \leftarrow S_\perp$ $\{S_\perp$ *is the bivalent root of* $\Upsilon^i\}$
repeat forever
  Let p be the next correct process in round-robin order
  Let m be the oldest message addressed to p in the message buffer of $S(I^i)$
    (if no such message exists, $m = \lambda$)
  if S has a descendent S' such that, for some d, $S' \cdot (p, m, d)$ is a <u>bivalent</u> vertex in Υ^i
    then $S \leftarrow S' \cdot (p, m, d)$ $\{S$ *is bivalent*$\}$
  else exit

Figure 5: Generating path π in Υ^i

The second type of decision gadget is called a *hook*, and is shown in Figure 4. Let S be the pivot of the hook. There is a step e such that $S \cdot e$ is one leaf, and the other leaf is $S \cdot (p, m, d) \cdot e$ for some p, m, d. Process p is the *deciding process of the hook*, because the decision of correct processes is determined by whether p takes the step (p, m, d) before e.

We shall prove that the deciding process p of a decision gadget must be correct (Lemma 23). Intuitively, this is because if p crashes no process can figure out whether p has taken the step that determines the decision value. The existence of such a critical "hidden" step is also at the core of many impossibility proofs starting with [FLP85]. In our case, the "hiding" is more difficult because now processes have recourse to the failure detector $\mathcal{D}$. Despite this, the hiding of the step of the deciding process of a decision gadget is still possible.

Lemma 20: If index i is bivalent critical then Υ^i has at least one decision gadget (and hence a deciding process).

PROOF: Starting from the bivalent root of Υ^i, we generate a path π in Υ^i, all the vertices of which are bivalent, as follows. We consider all correct processes in round-robin fashion. Suppose we have generated path S so far, and it is the turn of process p. Let m be the the oldest message destined to p that is in the message buffer of $S(I^i)$.[8] (If no such message exists, we take m to be the null message.) We try to extend the path S so that the last edge in the extension corresponds to p receiving m and the target of that edge is a bivalent vertex. The path construction ends if and when such an extension is no longer possible. This construction is shown in Figure 5. Each iteration of the loop extends the path by at least one edge. Let π be the path generated by these iterations; π is finite or infinite depending on whether the loop terminates.
Claim 1: π is finite.

PROOF: Suppose, for contradiction, that π is infinite. Let S be the schedule associated

[8] By a slight abuse of notation we identify a finite path from the root and its associated schedule.

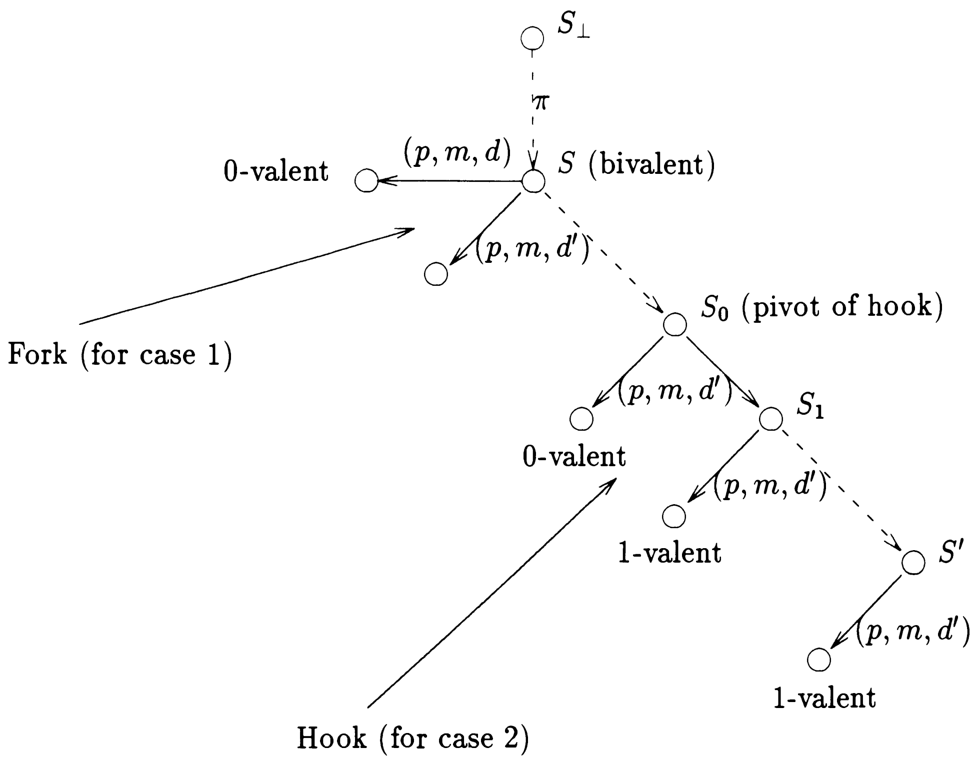

Figure 6: The decision gadgets in Υ^i if i is bivalent critical

with π. By construction, in S every correct process takes an infinite number of steps and every message sent to a correct process is eventually received. By Lemma 8, there is a T such that $R = \langle F, H_\mathcal{D}, I^i, S, T\rangle$ is a run of *Consensus*$_\mathcal{D}$. By construction, all vertices in π are bivalent. By Lemma 17, no correct process decides in R, thus violating the termination requirement of Consensus—a contradiction. $\square_{\textbf{claim 1}}$

Let S be the last vertex of π (clearly, S is bivalent). Let p be the next correct process in round-robin order when the loop in Figure 5 terminates. Let m be the oldest message addressed to p in the message buffer of $S(I^i)$ (if no such message exists, m is the null message). The loop exit condition and Lemma 14 imply that

$$\text{All descendents } S' \cdot (p, m, -) \text{ of } S \text{ are monovalent.} \qquad (*)$$

From Lemma 10, for some d, S has a child $S \cdot (p, m, d)$ in Υ^i. By $(*)$, $S \cdot (p, m, d)$ is monovalent. Without loss of generality, assume it is 0-valent.
Claim 2: For some d' there is a descendent S' of S such that $S' \cdot (p, m, d')$ is a 1-valent vertex of Υ^i, and the path from S to S' contains no edge labeled $(p, m, -)$.

Proof: Since S is bivalent, it has a descendent S^* such that some correct process has decided 1 in $S^*(I^i)$. From Lemmata 13 and 16, S^* is 1-valent. There are two cases:

1. The path from S to S^* does not have an edge labeled $(p, m, -)$. Suppose $m \neq \lambda$. Since m is in the message buffer of $S(I^i)$ and p does not receive m in the path from S to S^*, m is still in the message buffer of $S^*(I^i)$. From Lemma 10, for some d', $S^* \cdot (p, m, d')$ is in Υ^i. Since S^* is 1-valent, by Lemma 15, $S^* \cdot (p, m, d')$ is also 1-valent. In this case, the required S' is S^*.

2. The path from S to S^* has an edge labeled $(p, m, -)$. Let (p, m, d') be the first such edge on that path. Let S' be the source of this edge. By $(*)$, $S' \cdot (p, m, d')$ is monovalent. Since $S' \cdot (p, m, d')$ has a 1-valent descendent S^*, by Lemma 15, $S' \cdot (p, m, d')$ is 1-valent. □**claim 2**

Consider the vertex S' and edge (p, m, d') of Claim 2. By Lemma 11, for each vertex S'' on the path from S to S' (inclusive), $S'' \cdot (p, m, d')$ is also in Υ^i. By $(*)$, all such vertices $S'' \cdot (p, m, d')$ are monovalent. In particular, $S \cdot (p, m, d')$ is monovalent. There are two cases (see Figure 6):

1. $S \cdot (p, m, d')$ is 1-valent. Since $S \cdot (p, m, d)$ is 0-valent, Υ^i has a fork with pivot S.

2. $S \cdot (p, m, d')$ is 0-valent. Recall that $S' \cdot (p, m, d')$ is 1-valent and for each vertex S'' between S and S', $S'' \cdot (p, m, d')$ is monovalent. Thus, the path from S to S' must have two vertices S_0 and S_1 such that S_0 is the parent of S_1, $S_0 \cdot (p, m, d')$ is 0-valent and $S_1 \cdot (p, m, d')$ is 1-valent. Hence, Υ^i has a hook with pivot S_0. □

6.4 Extracting the correct process

By Lemma 19, there is a critical index i. If i is monovalent critical, Lemma 21 below shows how to extract a correct process. If i is bivalent critical, a correct process can be found by applying Lemmata 20 and 23.

Lemma 21: If index i is monovalent critical then p_i is correct.

Proof: Suppose, for contradiction, that p_i crashes. By Lemma 12(1) (applied to the root $S = S_\perp$ of Υ^i), there is a finite schedule E that contains only steps of correct processes (and hence no step of p_i) such that all correct processes have decided in $E(I^i)$. Since index i is monovalent critical, the root $S_\perp$ of Υ^i is 1-valent. Hence all correct processes must have decided 1 in $E(I^i)$.

I^i and I^{i-1} only differ in the state of p_i. Since S is applicable to I^i and does not contain any steps of p_i, an easy induction on the number of steps in S shows that: (a) S is also applicable to I^{i-1}, and (b) the state of all processes other than p_i are the same in $S(I^i)$ and $S(I^{i-1})$. Using Lemma 9, (a) implies that S is also a vertex of Υ^{i-1}. By (b),

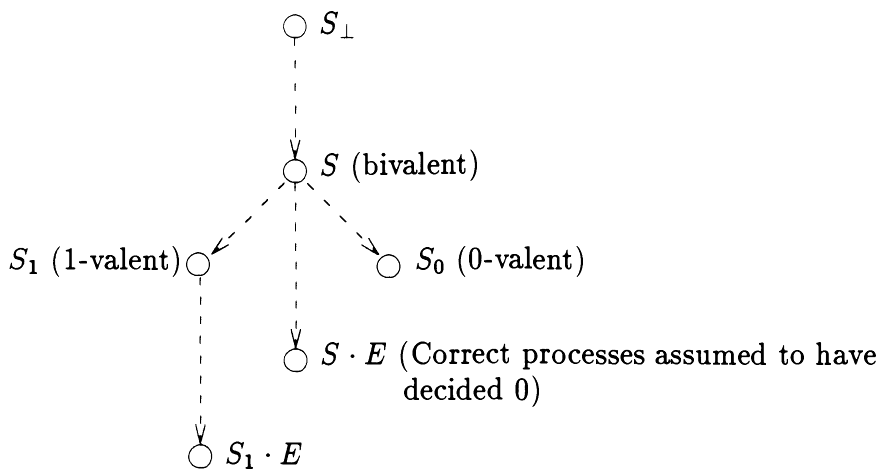

Figure 7: Lemma 22

all correct processes have decided 1 in $S(I^{i-1})$. Thus the root of Υ^{i-1} has tag 1. Since i is monovalent critical, the root of Υ^{i-1} is 0-valent—a contradiction. $\square$

Lemma 22: Let S be any bivalent vertex of Υ^i, and S_0, S_1 be any 0-valent and 1-valent descendents of S. If the paths from S to S_0 and from S to S_1 contain only steps of the form $(p, -, -)$, then p is correct.

PROOF: Suppose, for contradiction, that p crashes. From Lemma 12, there is a schedule E containing only steps of correct processes (and hence no step of p) such that:

1. $S \cdot E$ is a vertex of Υ^i and all correct processes have decided in $S \cdot E(I^i)$.
2. For $k = 0, 1$, if $S_k \cdot E$ is applicable to I^i then $S_k \cdot E$ is a vertex of Υ^i.

Without loss of generality assume that all correct processes decided 0 in $S \cdot E(I^i)$. Refer to Figure 7. Since all steps in the path from S to S_1 are steps of p, the state of every process other than p is the same in $S(I^i)$ and in $S_1(I^i)$. Furthermore, any message addressed to a process other than p that is in the message buffer in $S(I^i)$ is still in the message buffer in $S_1(I^i)$. Since E is applicable to $S(I^i)$ and does not contain any steps of p, an easy induction on the number of steps in E shows that: (a) E is also applicable to $S_1(I^i)$, and (b) the state of every process other than p is the same in $S \cdot E(I^i)$ and $S_1 \cdot E(I^i)$. By (ii), (a) implies that $S_1 \cdot E(I^i)$ is a vertex in Υ^i. By (b), all correct processes decide 0 in $S_1 \cdot E(I^i)$. Thus S_1, has tag 0. But S_1 is 1-valent—a contradiction. $\square$

Lemma 23: The deciding process of a decision gadget is correct.

PROOF: Let γ be any decision gadget of Υ^i. There are two cases to consider:

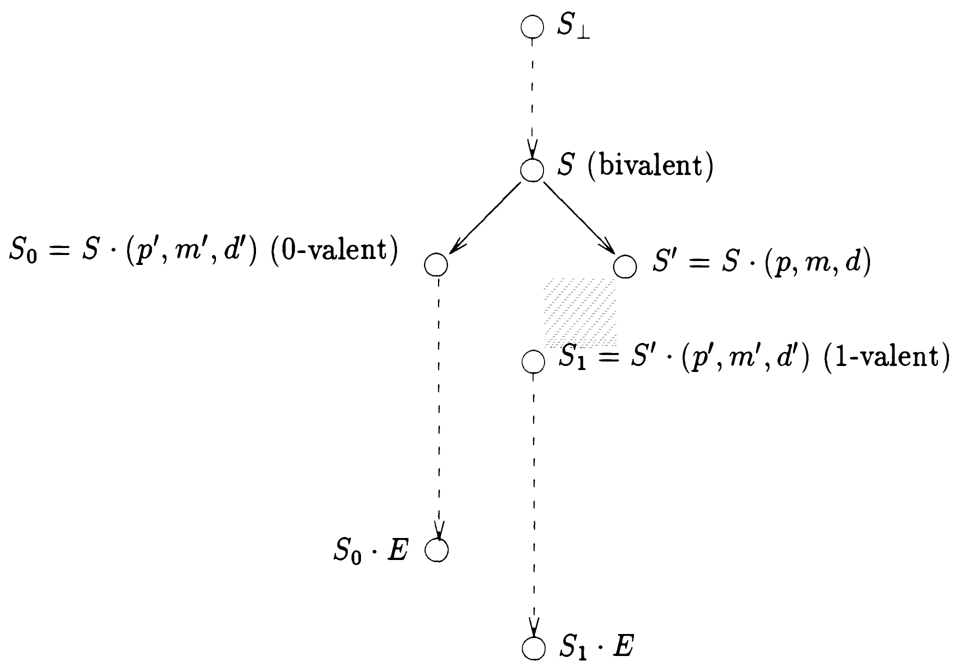

Figure 8: Lemma 23

1. γ is a fork. By Lemma 22, the deciding process of γ is correct.

2. γ is a hook. Assume (without loss of generality) that S is the pivot of γ, $S_0 = S \cdot (p', m', d')$ is the 0-valent leaf of γ and $S_1 = S \cdot (p, m, d) \cdot (p', m', d')$ is the 1-valent leaf of γ (see Figure 8). There are two cases:

   (a) $p = p'$. By Lemma 22, p is correct.

   (b) $p \neq p'$. Suppose, for contradiction, that p crashes. By Lemma 12, there is a schedule E containing only steps of correct processes (and hence no step of p) such that:

      i. $S_0 \cdot E$ is a vertex of Υ^i and all correct processes have decided in $S_0 \cdot E(I^i)$. Since S_0 is 0-valent, all correct processes must have decided 0 in $S_0 \cdot E(I^i)$.

      ii. If E is applicable to $S_1(I^i)$ then $S_1 \cdot E$ is a vertex of Υ^i.

   Let $S' = S \cdot (p, m, d)$ be the parent of S_1. The state of every process other than p is the same in $S(I^i)$ and $S'(I^i)$. Furthermore, any message addressed to a process other than p that is in the message buffer in $S(I^i)$ is still in the message buffer in $S'(I^i)$. Therefore, since $S_0 = S \cdot (p', m', d')$ and $S_1 = S' \cdot (p', m', d')$, the state of every process other than p is the same in $S_0(I^i)$ and $S_1(I^i)$. In addition, any message addressed to a process other than p that is in the

{*Build and tag simulation forest* Υ *induced by* G}
for $i \leftarrow 0, 1, \ldots, n$:
    $\Upsilon^i \leftarrow$ simulation tree induced by G and I^i
    for every vertex S of Υ^i
        if S has a descendent S' such that a correct process has decided k in $S'(I^i)$
        then add tag k to S

{*Select a process from tagged simulation forest* Υ}
$i \leftarrow$ smallest critical index (1)
if i is monovalent critical **then return** p_i (2)
else return deciding process of the smallest decision gadget in Υ^i (3)

Figure 9: Selecting a correct process

message buffer in $S_0(I^i)$ is also in the message buffer in $S_1(I^i)$. Since E is applicable to $S_0(I^i)$ and does not contain any steps of p, an easy induction on the number of steps in E shows that: (a) E is also applicable to $S_1(I^i)$, and (b) the state of every process other than p is the same in $S_0 \cdot E(I^i)$ and $S_1 \cdot E(I^i)$. By (ii), (a) implies that $S_1 \cdot E$ is a vertex of Υ^i. By (b), all correct processes decide 0 in $S_1 \cdot E(I^i)$. Thus S_1, receives a tag of 0. But S_1 is 1-valent—a contradiction. □

There may be several critical indices and several decision gadgets in the simulation forest. Thus, Lemmata 21 and 23 may identify many correct processes. Our selection rule will choose *one* of these, as the failure detector Ω requires, as follows. It first determines the smallest critical index i. If i is monovalent critical, it selects p_i. If, on the other hand, i is bivalent critical, it chooses the "smallest" decision gadget in Υ^i according to some encoding of gadgets, and selects the corresponding deciding process. It is easy to encode finite graphs as natural numbers. Since a decision gadget is just a finite graph, the selection rule can use any such encoding. The whole method of selecting a correct process is shown in Figure 9.

Theorem 24: The algorithm in Figure 9 selects a correct process.

PROOF: By Lemma 19, there is a critical index $i, 0 < i \leq n$. If i is monovalent critical, Line 2 returns p_i which, by Lemma 21, is correct. If i is bivalent critical, by Lemma 20, Υ^i contains at least one decision gadget. Let γ be the decision gadget in Υ^i with the smallest encoding. By Lemma 23, the deciding process of γ is correct in F. Thus, Line 3 returns the identity of a process that is correct. □

6.5 The reduction algorithm $T_{\mathcal{D}\to\Omega}$

The selection of a correct process described in Figure 9 is not yet the distributed algorithm $T_{\mathcal{D}\to\Omega}$ that we are seeking: it involves an infinite simulation forest and is "centralized". To turn it into a distributed algorithm, we will modify it as follows. Each process will cooperate with other processes to construct ever increasing finite approximations of the simulation forest. Such approximations will eventually contain the decision gadget and the other tagging information necessary to extract the identity of the *same* correct process chosen by the selection method in Figure 9.

Note that the selection method in Figure 9 involves three stages: The construction of G, a graph representing samples of failure detector values and their temporal relationship; the construction and tagging of the simulation forest induced by G; and, finally, the selection of a correct process using this forest.

Algorithm $T_{\mathcal{D}\to\Omega}$ consists of two components. In the first component, each process repeatedly queries its failure detector module and sends the failure detector values it sees to the other processes. This component enables correct processes to construct ever increasing finite approximations of the *same* G. Since all inter-process communication occurs in this component, we call it the *communication component* of $T_{\mathcal{D}\to\Omega}$.

In the second component, each process repeatedly (a) constructs and tags the simulation forest induced by its current approximation of G, and (b) selects the identity of a process using its current simulation forest. Since this component does not require any communication, we call it the *computation component* of $T_{\mathcal{D}\to\Omega}$.

6.5.1 The communication component

In this component processes cooperate to construct ever increasing approximations of the same G. Let G_p denote p's current approximation of G. Roughly speaking, each process p periodically performs the following two tasks: (i) If p receives G_q for some q, it incorporates this information by replacing G_p with the union of G_p and G_q. (ii) Process p queries its own failure detector module. Let d be the value that it sees and $[p', d']$ be any vertex currently in G_p. Clearly, p saw d after p' saw d'. Thus p adds $[p, d]$ to G_p, with edges from all other vertices of G_p to $[p, d]$. Process p then sends its updated G_p to all other processes. The communication component of $T_{\mathcal{D}\to\Omega}$ for p is shown in Figure 10.

Let $G_p(t)$ denote the value of G_p at time t. If p takes a step at time t, $G_p(t)$ denotes the value of G_p *at the end* of that step. The next two lemmata establish certain useful properties of the graphs constructed by the communication component. In reading the proofs of these results it will be useful to keep in mind that in our model the three phases of a step — receive, failure detection query, and send — occur atomically at a single time t.[9]

[9] As mentioned in footnote 4, our results would be valid in a model where process steps have a finer granularity. In such a model the proofs of Lemmata 25 and 26 below would be the same in essence, although some of the details would be different to account for the fact that the three phases of what is now considered an atomic step would not necessarily take place at the same time.

{*Build the directed acyclic graph* G_p}
$G_p \leftarrow$ empty graph
repeat forever
  RECEIVE PHASE:
    p receives m
  FAILURE DETECTOR QUERY PHASE:
    $d_p \leftarrow$ query failure detector $\mathcal{D}$
  SEND PHASE:
    if m is of the form (q, G_q, p) **then**
      $G_p \leftarrow G_p \cup G_q$ (1)
    add $[p, d_p]$ to G_p and edges from all other vertices of G_p to $[p, d_p]$ (2)
    output$_p \leftarrow$ *computation component* {*Figure 11*} (3)
    p sends (p, G_p, q) to all $q \in \Pi$ (4)

Figure 10: Process p's communication component

Lemma 25: Let v be a vertex contained in some local graph during the execution of the communication component. Let $G_p(t)$ be the first graph that contains v. (That is, v is in $G_p(t)$, but not in $G_q(t')$, for any process q and time $t' < t$.) Then

1. $v = [p, d]$, and p saw d at time t.

2. If $u \rightarrow v$ is an edge contained in some local graph during the execution of the communication component then $u \rightarrow v$ is contained in $G_p(t)$.

3. $G_p(t)$ is a subgraph of any graph that contains v.

PROOF: 1. Process p adds v into $G_p(t)$ in Line (1) or (2). In the latter case, the result follows immediately. In the former case, p must have received a message at time t with a graph that contains v. The process that sent that message must have therefore had v in its graph before time t, contradicting the choice of $G_p(t)$ as the first graph to contain v.

2. Consider the earliest time t' when the edge $u \rightarrow v$ was added to some graph, say of process q. By definition of t, $t' \geq t$. If $t' > t$, at time t' process p' receives a message that contains a graph with the edge $u \rightarrow v$. The sender of that message had a graph that contained the edge $u \rightarrow v$ at some time before t', contrary to the choice of t'. Therefore it must be that $t' = t$. Then, by Part (1), $q = p$ and so $u \rightarrow v$ is in $G_p(t)$, as wanted.

3. Suppose, for contradiction, that some graph contains v but is not a supergraph of $G_p(t)$. Choose the first such graph, say, $G_q(t')$. By definition of t, $t' \geq t$. Clearly, $q \neq p$ because p never removes any vertices or edges from its own graph. Therefore, at time t' process q receives a message with a graph that contains v but is not a supergraph of

$G_p(t)$. The sender of that message must have had a graph that contains v but is not a supergraph of $G_p(t)$ before time t', contrary to the choice of $G_q(t')$. □

Recall that we are considering a fixed run of $T_{\mathcal{D}\to\Omega}$, with failure pattern F, and failure detector history $H_{\mathcal{D}} \in \mathcal{D}(F)$. We now prove that the graphs constructed by the communication component of $T_{\mathcal{D}\to\Omega}$ satisfy certain properties. The reader should note the similarity between the first four and the four properties of the graphs defined in Section 6.1.

Lemma 26: For any correct process p and time t:

1. The vertices of $G_p(t)$ are of the form $[p', d']$ where $p' \in \Pi$ and $d' \in \mathcal{R}_{\mathcal{D}}$. If $[p', d']$ is a vertex of $G_p(t)$, then there is a time t' such that $p' \notin F(t')$ and $d' = H_{\mathcal{D}}(p', t')$.
2. If $[q_1, d_1] \to [q_2, d_2]$ is an edge of $G_p(t)$ and $d_1 = H_{\mathcal{D}}(q_1, t_1)$ and $d_2 = H_{\mathcal{D}}(q_2, t_2)$ then $t_1 < t_2$.
3. $G_p(t)$ is transitively closed.
4. There is a time $t' \geq t$ and a failure detector value d such that for all vertices $[p', d']$ of $G_p(t)$, $[p', d'] \to [p, d]$ is an edge of $G_p(t')$.
5. $G_p(t)$ is a subgraph of $G_p(t')$, for all $t' \geq t$.
6. For all correct q, there is a time $t' \geq t$ such that $G_p(t)$ is a subgraph of $G_q(t')$.

PROOF:

Property 1 : Consider the first graph that contains the vertex $[p', d']$. By Lemma 25(1), this graph is $G_{p'}(t')$ for some time t', and p' saw d' at time t'. This means that $p' \notin F(t')$ (otherwise p' would not have taken a step at time t' and would not have seen d'), and $d' = H_{\mathcal{D}}(p', t')$, as wanted.

Property 2 : By Lemma 25(2), $[q_1, d_1] \to [q_2, d_2]$ is an edge of $G_{q_2}(t_2)$. Let t' be the time when q_2 inserted vertex $[q_1, d_1]$ into G_{q_2}. Of course, $t' \leq t_2$. There are two cases:

1. $t' < t_2$. By Lemma 25(1), $[q_1, d_1]$ was not in any graph before time t_1. Thus, $t_1 \leq t'$ and from the hypothesis of this case, $t_1 < t_2$.
2. $t' = t$. Then q_2 received a graph containing $[q_1, d_1]$ at t_2. Let t'' be the time when this graph was sent. Of course, $t'' < t_2$. By Lemma 25(1), $[q_1, d_1]$ was not in any graph before t_1, and therefore $t_1 \leq t''$. Thus, $t_1 < t_2$.

Property 3 : Let $[q_1, d_1] \rightarrow \ldots \rightarrow [q_k, d_k]$ be a path in $G_p(t)$. We must show that there is an edge $[q_1, d_1] \rightarrow [q_k, d_k]$ in $G_p(t)$.

Let t_i be the time when q_i inserted $[q_i, d_i]$ in G_{q_i}, for $1 \leq i \leq k$. By induction on i we show that $[q_1, d_1] \rightarrow \ldots \rightarrow [q_i, d_i]$ is a path in $G_{q_i}(t_i)$. The basis, $i = 1$, is trivial. For the induction step, suppose that $[q_1, d_1] \rightarrow \ldots \rightarrow [q_{i-1}, d_{i-1}]$ is a path in $G_{q_{i-1}}(t_{i-1})$. Since $[q_{i-1}, d_{i-1}] \rightarrow [q_i, d_i]$ is an edge in $G_p(t)$, by Lemma 25(2), it is also an edge in $G_{q_i}(t_i)$. Since $[q_{i-1}, d_{i-1}]$ is a vertex in $G_{q_i}(t_i)$, by Lemma 25(3), $G_{q_{i-1}}(t_{i-1})$ is a subgraph of $G_{q_i}(t_i)$. In particular, $G_{q_i}(t_i)$ contains the path $[q_1, d_1] \rightarrow \ldots \rightarrow [q_{i-1}, d_{i-1}]$. Thus, $[q_1, d_1] \rightarrow \ldots \rightarrow [q_i, d_i]$ is a path in $G_{q_i}(t_i)$, as wanted.

Therefore, the vertices $[q_1, d_1], \ldots, [q_k, d_k]$ are all in $G_{q_k}(t_k)$. At time t_k, q_k adds an edge from every other vertex to $[q_k, d_k]$. Thus, the edge $[q_1, d_1] \rightarrow [q_k, d_k]$ is in $G_{q_k}(t_k)$. By Lemma 25(3), $G_{q_k}(t_k)$ is a subgraph of $G_p(t)$ (since the latter contains $[q_k, d_k]$). Therefore, $[q_1, d_1] \rightarrow [q_k, d_k]$ is in $G_p(t)$, as wanted.

Property 5 : Once a vertex or edge is added to G_p it is not removed.

Property 4 : Since p is correct, it takes a step at some time t' after t. In the failure detector query phase of this step, p queries its failure detector module and obtains a value, say d. In Line 2 of this step, p adds the vertex $[p, d]$ to G_p and an edge from all other vertices of $G_p(t')$ to $[p, d]$. From Property 5, $G_p(t)$ is a subgraph of $G_p(t')$, hence the result follows.

Property 6 : Since p is correct, it eventually sends $G_p(t)$ to all processes, including q (this occurs in p's first execution of Line 4 after time t). Since q is correct, it eventually receives $G_p(t)$, and then replaces G_q with $G_q \cup G_p(t)$, say at time t'. So, $G_p(t)$ is a subgraph of $G_q(t')$. □

Property 5 of the above lemma allows us to define $G_p^\infty = \bigcup_{t \in \mathcal{T}} G_p(t)$. From Property 6, we get:

Lemma 27: For any correct processes p and q, $G_p^\infty = G_q^\infty$.

PROOF: Let o be any vertex or edge of G_p^∞, i.e., there is a time t at which o is in $G_p(t)$. From Lemma 26 (6), there is a time t' such that $G_p(t)$ is a subgraph of $G_q(t')$. Thus o is in G_q^∞. Thus G_p^∞ is a subgraph of G_q^∞. By a symmetric argument, G_q^∞ is a subgraph of G_p^∞, hence $G_p^\infty = G_q^\infty$. □

Lemma 27 allows us to define the *limit graph* G to be G_p^∞ for any correct process p. The first four properties of Lemma 26 immediately imply:

Lemma 28: The limit graph G satisfies the four properties of the DAG defined in Section 6.1.

As before, Υ^i denotes the tagged simulation tree induced by G and initial configuration I^i, and Υ denotes the tagged simulation forest $\{\Upsilon^0, \Upsilon^1, \ldots, \Upsilon^n\}$.

6.5.2 The computation component

Since the limit graph G has the four properties of the DAG, we can apply the "centralized" selection method of Figure 9 to identify a correct process. This method involved:

- Constructing and tagging the infinite simulation forest Υ induced by G.
- Applying a rule to Υ to select a particular correct process p^*.

In the computation component of $T_{\mathcal{D}\to\Omega}$, each p approximates the above method by repeatedly:

- Constructing and tagging the *finite* simulation forest Υ_p induced by G_p, its present finite approximation of G.
- Applying the same rule to Υ_p to select a particular process.

Since the limit of Υ_p over time is Υ, and the information necessary to select p^* is in a finite subgraph of Υ, we can show that *eventually* p will keep selecting the correct process p^*, forever.

Actually, p cannot quite use the tagging method of Figure 9: that method requires knowing which processes are correct! Instead, p assigns tag k to a vertex S in Υ_p^i if and only if S has a descendent S' such that p itself has decided k in $S'(I^i)$. If p is correct, this is eventually equivalent to the tagging method of Figure 9. If p crashes, we do not care how it tags its forest. Also, p cannot use exactly the same selection method as that of Figure 9: its current simulation forest Υ_p may not *yet* have a critical index or contain any decision gadget (although it eventually will!). In that case, p temporizes by just selecting itself. The computation component of $T_{\mathcal{D}\to\Omega}$ is shown in Figure 11 (compare it with the selection method of Figure 9).

We first show that Υ_p, the simulation forest that p constructs, is indeed an increasingly accurate approximation of Υ (Lemma 29). We then show that the tags that p gives to any vertex S in Υ_p are eventually the same ones that the tagging rule of Figure 9 gives to S in Υ (Lemma 30). Let $\Upsilon_p(t)$ denote Υ_p at time t, i.e., $\Upsilon_p(t)$ is the finite simulation forest induced by $G_p(t)$.

Lemma 29: For any correct p and any time t:

1. $\Upsilon_p(t)$ is a subgraph[10] of Υ.
2. $\Upsilon_p(t)$ is a subgraph of $\Upsilon_p(t')$, for all $t' \geq t$.
3. $\bigcup_{t\in T} \Upsilon_p(t) = \Upsilon$.

[10]The subgraph and graph equality relations ignore the tags.

{Build and tag simulation forest Υ_p induced by G_p}
for $i \leftarrow 0, 1, \ldots, n$:
    $\Upsilon_p^i \leftarrow$ simulation tree induced by G_p and I^i
    for every vertex S of Υ_p^i
        if S has a descendent S' such that p has decided k in $S'(I^i)$
        then add tag k to S

{Select a process from tagged simulation forest Υ_p}
if there is no critical index **then return** p
else
    $i \leftarrow$ smallest critical index (1)
    if i is monovalent critical **then return** p_i (2)
    else if Υ_p^i has no decision gadgets **then return** p
        else return deciding process of the smallest decision gadget in Υ_p^i (3)

Figure 11: Process p's computation component

PROOF:

Property 1 : Let S be any vertex of tree $\Upsilon_p^i(t)$ (for some i, $0 \leq i \leq n$). From the definition of $\Upsilon_p^i(t)$, S is compatible with some path g of $G_p(t)$ and applicable to I^i. Since $G_p(t)$ is a subgraph of G, g is also a path of G. Thus, S is compatible with G; since it is also applicable to I^i, it is a vertex of Υ^i.

Similarly, let $S \rightarrow S'$ be an edge e of $\Upsilon_p^i(t)$. Since S and S' are also vertices of Υ^i, and $S' = S \cdot e$, $S \rightarrow S'$ is also an edge of Υ^i.

Property 2 : Follows from Lemma 26 (5).

Property 3 : We first show that Υ is a subgraph of $\bigcup_{t \in \mathcal{T}} \Upsilon_p(t)$. Let S be any vertex of any tree Υ^i of Υ. From the definition of Υ^i, S is compatible with some finite path g of G and is applicable to I^i. Since $G = \bigcup_{t \in \mathcal{T}} G_p(t)$ and g is a finite path of G, there is a time t such that g is also a path of $G_p(t)$. Since S is compatible with g of $G_p(t)$ and is applicable to I^i, S is a vertex of $\Upsilon_p^i(t)$.

Let $S \rightarrow S'$ be any edge e of Υ^i. By the argument above, there is a time t after which both S and S' are vertices of Υ_p^i. Since $S' = S \cdot e$, after time t the edge e is also in Υ_p^i. Thus, every vertex and every edge of Υ is also in $\bigcup_{t \in \mathcal{T}} \Upsilon_p(t)$, i.e., Υ is a subgraph of $\bigcup_{t \in \mathcal{T}} \Upsilon_p(t)$.

By Property 1, $\bigcup_{t \in \mathcal{T}} \Upsilon_p(t) = \Upsilon$. □

Lemma 30: Let p be any correct process, and S be any vertex of Υ_p. There is a time after which the tags of S in Υ_p are the same as the tags of S in Υ.

PROOF: Suppose that at some time t, p assigns tag k to vertex S of tree Υ^i_p. This means that S has a descendent S' in $\Upsilon^i_p(t)$ such that p has decided k in $S'(I^i)$. By Lemma 29(1), S' is also a descendent of S in Υ^i, and since p is correct, S has tag k in Υ^i as well.

Conversely, suppose a vertex S of a tree Υ^i of Υ has tag k. We show that, eventually, p also assigns tag k to S in Υ^i_p. Since S has tag k in Υ^i, S has a descendent S' in Υ^i such that some correct process has decided k in $S'(I^i)$ (cf. tagging rule in Figure 9). By Lemma 12(1), there is a descendent S'' of S' in Υ^i, such that *all* correct processes, including p, have decided in $S''(I^i)$. By Lemma 7, $S''(I^i)$ is a configuration of a partial run of $Consensus_{\mathcal{D}}$. By the Agreement property of Consensus, p must have decided k in $S''(I^i)$. Consider the path that starts from the root of Υ^i and goes to vertex S and then to S''. By Lemma 29(3), there is a time t after which this path is also in Υ^i_p. Therefore, when p executes the tagging rule of Figure 11 after time t, p assigns tag k to S in Υ^i_p (because p has decided k in $S''(I^i)$, and S'' is a descendent of S in Υ^i_p). □

Recall that p^* is the correct process obtained by applying the selection rule of Figure 9 to the infinite simulation forest Υ. We now show that there is a time after which any correct p always selects p^* when it applies the corresponding selection rule of Figure 11 to its own finite approximation of the simulation forest Υ_p. Roughly speaking, the reason is as follows. By Lemma 30, there is a time t after which the tags of all the roots in p's forest Υ_p are the same as in the infinite forest Υ. Since these tags determine the sets of monovalent and bivalent critical indices, after time t these sets according to p are the same as in Υ. Let i be the minimum critical index in these sets, and consider the situation after time t. If i is monovalent critical, the selection rule of Figure 11 selects p_i, which is what p^* is in this case. If i is bivalent critical, then p selects the deciding process of its current minimum decision gadget of Υ^i_p (if it has one). This case is examined below.

Let γ^* be the minimum decision gadget of Υ^i (so, p^* is the deciding process of γ^*). For a while, γ^* may not be the minimum decision gadget of Υ^i_p. This may be because γ^* (and its tags) is not yet in Υ^i_p. However, by Lemmata 29(3) and 30, γ^* (including its tags) will eventually be in Υ^i_p. Alternatively, it may be because Υ^i_p contains a subgraph γ whose encoding is smaller than γ^*'s, and for a while γ looks like a decision gadget according to its *present* tags. However, by Lemma 30, p will eventually determine *all* the tags of γ, and discover that γ is not really a decision gadget. Since there are only *finitely* many graphs whose encoding is smaller than γ^*'s, p will eventually discard all the "fake" decision gadgets (like γ) that are smaller than γ^*, and then select γ^* as its minimum decision gadget. After that time, p always selects the deciding process of γ^* — which is precisely p^*, in this case.

Theorem 31: For all correct processes p, there is a time after which $output_p = p^*$, forever.

PROOF: Let i^* denote the critical index selected by Line 1 of Figure 9 applied to Υ. By

Lemma 30, there is a time t_{init} after which every root of Υ_p has the same tags as the corresponding root of Υ. Thus after time t_{init}, p always sets $i = i^*$ in Line 1 of Figure 11. We now show that there is a time after which the computation component of p (Figure 11) always return p^*. There are two cases:

1. i^* is monovalent critical. In this case, p^* is process p_{i^*} (by Line 2 of the selection rule Figure 9). Similarly, after time t_{init}: (a) p always sets i to i^* (Line 1 of Figure 11); (b) p always returns p_{i^*} (Line 2 of Figure 11).

2. i^* is bivalent critical. Let γ^* denote the smallest decision gadget of Υ^{i^*}. In this case, p^* is the deciding process of γ^*. Since γ^* is a finite subgraph of Υ^{i^*}, by Lemma 29(3), there is a time after which γ^* is also a subgraph of Υ^i_p. By Lemma 30, there is a time t_{γ^*} after which all the (finitely many) vertices of γ^* receive the same tags in Υ^{i^*} and $\Upsilon^{i^*}_p$. Thus after time t_{γ^*}, γ^* is a also decision gadget of $\Upsilon^{i^*}_p$.

   Since each graph is encoded as a unique natural number, there are finitely many graphs with a smaller encoding than γ^*. Let $\mathcal{G}$ denote the set of graphs with a smaller encoding than γ^*, and γ be any graph in $\mathcal{G}$. We show that there is a time after which γ is not a decision gadget of $\Upsilon^{i^*}_p$. There are two cases:

   (a) γ is not a subgraph of Υ^{i^*}. In this case, by Lemma 29(1), γ is never a subgraph of $\Upsilon^{i^*}_p$.

   (b) γ is a subgraph of Υ^{i^*}. Since γ^* is the smallest decision gadget of Υ^{i^*} and γ is smaller than γ^*, γ is not a decision gadget of Υ^{i^*}. By Lemma 30, there is a time t_γ after which all the (finitely many) vertices of γ have the same tags in Υ^{i^*} and $\Upsilon^{i^*}_p$. Thus after time t_γ, γ is not a decision gadget of Υ^i_p.

   Since $\mathcal{G}$ is finite, there is a time $t_{\mathcal{G}}$ after which no graph in $\mathcal{G}$ is a decision gadget of Υ^i_p.

   Consider the process that is returned by the computation component of p (Figure 11) at any time $t > \max(t_{init}, t_{\gamma^*}, t_{\mathcal{G}})$. Since $t > t_{init}$, p always sets i to i^* in Line 1. Since $t > t_{\gamma^*}$, γ^* is a decision gadget of $\Upsilon^i_p(t)$. Finally, since $t > t_{\mathcal{G}}$, γ^* is the smallest decision gadget of $\Upsilon^i_p(t)$. Thus, since i^* is bivalent, at any time after $\max(t_{init}, t_{\gamma^*}, t_{\mathcal{G}})$, Line 3 of Figure 11 returns the deciding process of γ^*. Therefore, after time $\max(t_{init}, t_{\gamma^*}, t_{\mathcal{G}})$, the computation component of p always returns p^*.

From the above, there is a time after which p sets $output_p \leftarrow p^*$, forever, in Line 3 of Figure 10. □

We now have all the pieces needed to prove our main result, Theorem 2 in Section 5:

Theorem 2: For all environments $\mathcal{E}$, if a failure detector $\mathcal{D}$ can be used to solve Consensus in $\mathcal{E}$, then $\mathcal{D} \succeq_{\mathcal{E}} \Omega$.

PROOF: Consider the execution of algorithm $T_{\mathcal{D}\rightarrow\Omega}$ in any environment $\mathcal{E}$. By Theorem 31, there is a time after which all correct processes set $output_p = p^*$, forever. By

Theorem 24, p^* is a correct process. Thus, $T_{\mathcal{D}\rightarrow\Omega}$ is a reduction algorithm that transforms $\mathcal{D}$ into Ω. In other words, Ω is reducible to $\mathcal{D}$. □

7 Discussion

7.1 Granularity of atomic actions

Our model incorporates very strong assumptions about the atomicity of steps. First, the three phases of each step are assumed to occur indivisibly, and at a single time. In particular, the failure of a process cannot happen in the "middle of a step". This allows us to associate a single time t with a step and think of the step as occuring at that time. Second, in the send phase of a step a message is sent to *all* processes. Given that the entire step is indivisible, this means that either all or none of the correct processes eventually receive the message sent in a step. Finally, no two steps can occur at the same time.[11] These assumptions are convenient because they make the formal model simpler to describe. Also, they are consistent with those made in the model of [FLP85] that provided the impetus for this work.

On the other hand, in [CT91] a model with weaker properties is used. There, the three phases of a step need not occur indivisibly, and may occur at different times. Even within the send phase, the messages sent to the different processes may be sent at different times. Thus, a failure may occur in the middle of the send phase, resulting in some correct processes eventually receiving the messages sent to them in that step while others never do. Also, actions of *different* processes may take place simultaneously, subject to the restriction that a message can only be received strictly *after* it was sent. Since [CT91] is mainly concerned with showing how to use various types of failure detectors to achieve Consensus, the use of a weaker model strengthens the results. (In fact, the negative results of [CT91] hold even in the model of this paper, with the stronger atomicity assumptions.)

The question naturally arises whether our result also applies to this weaker model. In other words, if a failure detector $\mathcal{D}$ can be used to solve Consensus in the weak model, is it true that we can transform $\mathcal{D}$ to $\mathcal{W}$ *in that model*? It turns out that the answer is affirmative. To see this, first note that if $\mathcal{D}$ solves Consensus in the weak model then it surely solves Consensus in the strong model. By our result, $\mathcal{D}$ can be transformed to $\mathcal{W}$ in the strong model. It remains to show that $\mathcal{D}$ can be transformed to $\mathcal{W}$ in the weak model. This is not obvious, since it is conceivable that the extra properties of the strong model are crucial in the transformation of $\mathcal{D}$ to $\mathcal{W}$. Fortunately, the transformation presented in this paper actually works even in the weak model!

To see this, it is sufficient to make sure that the communication component of the transformation (cf. Figure 10 in Section 6.5.1) constructs graphs that satisfy the properties listed in Lemma 26, *even* if we run it in the weak model. It is not difficult to verify

[11] This is reflected in our formal model by the fact that the list of times in a run (which indicate when the events in the run's schedule occur) is *increasing*.

that this is indeed so. The proof is virtually the same, except for the fact that we must distinguish the time t in which a process p queries its failure detector and the time t' in which p adds the value it saw into G_p. In our proof we assume that $t = t'$; in the weak model we would have $t \leq t'$. Similar comments apply to all actions within a step that are no longer assumed to occur at the same instant of time. These changes make the proofs slightly more cumbersome, since we must introduce notation for all the different times in which relevant actions within a step take place, but the reasoning remains essentially the same.[12]

Thus, our result is not merely a fortuitous consequence of some whimsical choice of model. We view the robustness of the result across different models of asynchrony as further testimony to the significance of the failure detector $\mathcal{W}$.

7.2 Failure detection and partial synchrony

The fundamental reason why Consensus cannot be solved in completely asynchronous systems is the fact that, in such systems, it is impossible to reliably distinguish a process that has crashed from one that is merely very slow. In other words, Consensus is unsolvable because accurate failure detection is impossible. On the other hand, it is well-known that Consensus is solvable (deterministically) in completely synchronous systems — that is, systems where all processes take steps at the same rate and each message arrives at its destination a fixed and known amount of time after it is sent. In such a system we can use timeouts to implement a "perfect" failure detector — i.e., one in which no process is ever wrongly suspected, and every faulty process is eventually suspected. Thus the ability to solve Consensus in a given system is intimately related to the failure detection capabilities of that system. This realization led to the extension of the asynchronous model of computation with failure detectors in [CT91]. In that paper Consensus is shown to be solvable even with very weak failure detectors that could make an infinite number of "mistakes".

A different tack on circumventing the unsolvability of Consensus is pursued in [DDS87] and [DLS88]. The approach of those papers is based on the observation that between the completely synchronous and completely asynchronous models of distributed systems there lie a variety of intermediate "partially synchronous" models. For instance, in one model of partial synchrony, processes take steps at the same rate, but message delays are unbounded (albeit finite). Alternatively, it may be known that message delays are bounded, but the actual bound may be unknown. In yet another variation, the *eventual* maximum message delay is known, but during some initial period of finite but unknown duration some messages may experience longer delays. These and many other models of

[12] Another problem that must be confronted is that in the proofs of Lemmata 25 and 26 we often refer to the "first graph" in which a vertex or edge is present. In the strong model there is no difficulty with this, since processes cannot execute steps simultaneously. In the weak model, we have to justify that it makes sense to speak of the "first" graph to contain a vertex or edge, in spite of the fact that certain actions can be executed at the same time. The fact that a message can be received only *after* it was sent is needed here.

partial synchrony are studied in [DDS87] and [DLS88], and the question of solvability of Consensus in each of them is answered either positively or negatively.

In particular, [DDS87] defines a space of 32 models by considering five key parameters, each of which admits a "favourable" and an "unfavourable" setting. For instance, one of the parameters is whether the maximum message delay is known (favourable setting) or not (unfavourable setting). Each of the 32 models corresponds to a particular setting of the 5 parameters. [DDS87] identifies four "minimal" models in which Consensus is solvable. These are minimal in the sense that the weakening of any parameter from favourable to unfavourable would yield a model of partial synchrony where Consensus is unsolvable. Thus, within the space of the models considered, [DDS87] and [DLS88] delineate precisely the boundary between solvability and unsolvability of Consensus, and provide an answer to the question "What is the least amount of synchrony sufficient to solve Consensus?".

Failure detectors can be viewed as a more abstract and modular way of incorporating partial synchrony assumptions into the model of computation. Instead of focusing on the *operational features* of partial synchrony (such as the five parameters considered in [DDS87]), we can consider the *axiomatic properties* that failure detectors must have in order to solve Consensus. The problem of implementing a given failure detector in a specific model of partial synchrony becomes a separate issue; this separation affords greater modularity.

To see the connection between partial synchrony and failure detectors, it is useful to examine how one might go about implementing a failure detector. By the impossibility result of [FLP85], a failure detector that can be used to solve Consensus cannot be implemented in a completely asynchronous system. Now consider partially synchronous systems in which correct processes have accurate timers (i.e., they can measure elapsed time). If in such a system message delays are bounded and the maximum delay is known, we can use timeouts to implement the "perfect" failure detector described above. In a weaker system where message delays are bounded but the maximum delay is *not* known, we can implement a failure detector satisfying a weaker property: *eventually* no correct process is suspected. This can be done by using timeouts of increasing length; once the timeout period has been increased sufficiently to exceed the unknown maximum delay, no correct process will be suspected. A failure detector with the same property can also be implemented in a distributed system where the *eventual* maximum message delay is known, but messages may be delayed for longer during some initial period of finite but unknown duration. With these remarks we wish to illustrate two points: First, that stronger failure detectors correspond to stronger models of partial synchrony; and second, that the same failure detector can be implemented in different models of partial synchrony.

Studying failure detectors rather than various models of partial synchrony has several advantages. By determining whether Consensus is solvable using some specific failure detector we thereby determine whether Consensus is solvable in *all* systems in which that failure detector can be implemented. An algorithm that relies on the axiomatic properties of a given failure detector is more general, more modular, and simpler to understand than one that relies directly on some specific operational features of partial synchrony (that can be used to implement the given failure detector).

From this more abstract point of view, the question "What is the least amount of synchrony sufficient to solve Consensus?" translates to "What is the weakest failure detector sufficient to solve Consensus?". In contrast to [DDS87], which identified a *set* of minim*al* models of partial synchrony in which Consensus is solvable, we are able to exhibit a *single* minim*um* failure detector that can be used to solve Consensus. The technical device that made this possible is the notion of *reduction* between failure detectors. We suspect that a corresponding notion of reduction between models of partial synchrony, although possible, would be more complex. This is because there are models which are not comparable in general (in the sense that there are tasks that are possible in one but not in the other and vice versa), although they are comparable *as far as failure detection is concerned* — which is all that matters for solving Consensus! In this connection, it is useful to recall our earlier observation, that the same failure detector can be implemented in different (indeed, incomparable) models of partial synchrony.

7.3 Failure detectors with infinite range of output values

The failure detectors in [RB91, CT91] only output lists of processes suspected to have crashed. Since the set of processes is finite, the range of possible output values of these failure detectors is also finite. In this paper our model allows for failure detectors with arbitrary ranges of output values, including the possibility of infinite ranges! We illustrate the significance of this generality by describing a natural class of failure detectors whose range of output values is infinite (though each value output is finite).

Example: One apparent weakness with our formulation of failure detection is that a brief change in the value output by a failure detector module may go unnoticed. For example, process p's module of the given failure detector, $\mathcal{D}$, may output d_1 at time t_1, d_2 at a later time t_2 and d_1 again at time t_3 after t_2. If due to the asynchrony of the system p does not take a step between time t_1 and t_3, p may never notice that its failure detector module briefly output d_2. A natural way of overcoming this problem is to replace $\mathcal{D}$ with failure detector $\mathcal{D}'$ that has the following property: $\mathcal{D}'$ maintains the same list of suspects as $\mathcal{D}$ but when queried, $\mathcal{D}'$ returns the entire history of its list of suspects up to the present time. In this manner, correct processes are guaranteed to notice every change in $\mathcal{D}'$'s list of suspects. As the system continues executing, the values output by $\mathcal{D}'$ grow in size. This means that $\mathcal{D}'$ has an infinite range of output values.

However, since $\mathcal{D}$ is a function of F, the failure pattern encountered, $\mathcal{D}'$ is also a function of F, and can be described by our model. Thus, the result in this paper applies to $\mathcal{D}'$, a natural failure detector with infinite range of output values.

Acknowledgements

We would like to thank Prasad Jayanti for his valuable comments on various versions of this paper. We would also like to thank Cynthia Dwork, Dexter Kozen and the distributed systems group at Cornell for putting up with half-baked ideas.

References

[ABD+87] Hagit Attiya, Amotz Bar-Noy, Danny Dolev, Daphne Koller, David Peleg, and Rüdiger Reischuk. Achievable cases in an asynchronous environment. In *Proceedings of the Twenty-Eighth Symposium on Foundations of Computer Science*, pages 337–346. IEEE Computer Society Press, October 1987.

[BMZ88] Ofer Biran, Shlomo Moran, and Shmuel Zaks. A combinatorial characterization of the distributed tasks that are solvable in the presence of one faulty processor. In *Proceedings of the Seventh ACM Symposium on Principles of Distributed Computing*, pages 263–275, August 1988.

[BW87] M. Bridgland and R. Watro. Fault-tolerant decision making in totally asynchronous distributed systems. In *Proceedings of the Sixth ACM Symposium on Principles of Distributed Computing*, pages 52–63, August 1987.

[CD89] Benny Chor and Cynthia Dwork. Randomization in byzantine agreement. *Advances in Computer Research*, 5:443–497, 1989.

[CT91] Tushar Chandra and Sam Toueg. Unreliable failure detectors for asynchronous systems (preliminary version). In *Proceedings of the Tenth ACM Symposium on Principles of Distributed Computing*, pages 325–340. ACM Press, August 1991.

DDS87] Danny Dolev, Cynthia Dwork, and Larry Stockmeyer. On the minimal synchronism needed for distributed consensus. *Journal of the ACM*, 34(1):77–97, January 1987.

[DLP+86] Danny Dolev, Nancy A. Lynch, Shlomit S. Pinter, Eugene W. Stark, and William E. Weihl. Reaching approximate agreement in the presence of faults. *Journal of the ACM*, 33(3):499–516, July 1986.

[DLS88] Cynthia Dwork, Nancy A. Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):288–323, April 1988.

[FLP85] 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.

[RB91] Aleta Ricciardi and Ken Birman. Using process groups to implement failure detection in asynchronous environments. In *Proceedings of the Tenth ACM Symposium on Principles of Distributed Computing*, pages 341–351. ACM Press, August 1991.