The Weakest Failure Detector for Solving Consensus*

Tushar Deepak Chandra[†§] Vassos Hadzilacos[‡] Sam Toueg[§]

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 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, ABND+87, BW87], 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

*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

PoDC '92-8/92/B.C.

© 1992 ACM 0-89791-496-1/92/0008/0147...$1.50

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.

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.

[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 $\diamond\mathcal{W}$.

[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.

$\mathcal{A}$ 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.

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 $T_{\mathcal{B}\rightarrow\mathcal{W}}$ that can transform $\mathcal{B}$ into a failure detector with the Properties 1 and 2 of $\mathcal{W}$. $T_{\mathcal{B}\rightarrow\mathcal{W}}$ 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 $T_{\mathcal{B}\rightarrow\mathcal{W}}$ 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 D that can be used to solve Consensus (this result holds for any asynchronous system). We show this reduction by giving a distributed algorithm $T_{D\rightarrow\mathcal{W}}$ that transforms any such 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 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 D to lists of suspects. Instead, this output can be *any value* that encodes some information about failures. For example, a failure detector 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 D could be an arbitrarily complex (and unknown) encoding of failure information. Our transformation from D into $\mathcal{W}$ must be able to decode this information.

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

Fortunately, since D can be used to solve Consensus, there is a corresponding algorithm, $Consensus_D$, that is somehow able to "decode" the information about failures provided by D, and knows how to use it to solve Consensus. Our reduction algorithm, $T_{D\rightarrow\mathcal{W}}$ uses $Consensus_D$ to extract this information from 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* D is a function that maps each failure pattern F to a *set* of failure detector histories with range $\mathcal{R}_D$ (where $\mathcal{R}_D$ denotes the range of failure detector outputs of D). $D(F)$ denotes the set of possible failure detector histories permitted by 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 D as a function of the failure pattern F of an execution. However, this does *not* preclude an *implementation* of 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 $D(F)$ 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]

[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.

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

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) ($m = \lambda$ when the null message is received). 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 (possibly finite) 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.

A *partial run of algorithm A using a failure detector D* is a tuple $R = \langle F, H_D, I, S, T\rangle$ where F is a failure pattern, $H_D \in D(F)$ 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., $d = H_D(p, T[i])$

Informally, a partial run of A using D represents a finite point of some execution of A using D.

A *run of an algorithm A using a failure detector D* is a tuple $R = \langle F, H_D, I, S, T\rangle$ where F is a failure pattern, $H_D \in D(F)$ 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.
- Every message sent to a correct process is eventually received.

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

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

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.

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.

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

Termination: Each correct process eventually decides.

Validity: Each correct process decides on the initial value of some process.

Agreement: No two correct processes decide differently.

4 Reducibility

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

Given $T_{D\to D'}$, anything that can be done using D' in $\mathcal{E}$, can be done using D instead. To see this, suppose a given algorithm B requires failure detector D' (when it executes in $\mathcal{E}$), but only D is available. We can still execute B as follows. Concurrently with B, we run $T_{D\to D'}$ to transform D into 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 $T_{D\to D'}$) instead of querying its failure detector module. This is illustrated in Fig. 1.

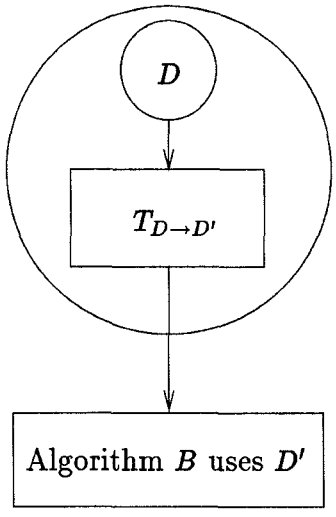

Figure 1: Transforming D into D'

Intuitively, since $T_{D\to D'}$ is able to use D to emulate D', D provides at least as much information about process failures in $\mathcal{E}$ as D' does. Thus, if there is an algorithm $T_{D\to D'}$ that transforms D into D' in $\mathcal{E}$, we write $D \succeq_{\mathcal{E}} D'$ and say that D' *is reducible to* D *in* $\mathcal{E}$; we also say that D' *is weaker than* 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 $n > 2f$.

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 D that can be used to solve Consensus is at least as strong as Ω. Thus, 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 $\mathcal{W}$. □

Theorem 2: For all environments $\mathcal{E}$, if a failure detector D can be used to solve Consensus in $\mathcal{E}$, then $D \succeq_\mathcal{E} \Omega$.
Proof: The reduction algorithm $T_{D \to \Omega}$ is shown in Section 6. It is the core of our result. □

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

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:

Corollary 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, D be a failure detector that can be used to solve Consensus in $\mathcal{E}$, and $\mathit{Consensus}_D$ be the Consensus algorithm that uses D. We describe an algorithm $T_{D \to \Omega}$ that transforms D into Ω in $\mathcal{E}$. Intuitively, this algorithm works as follows. Fix an arbitrary run of $T_{D \to \Omega}$ using D in $\mathcal{E}$, with failure pattern $F \in \mathcal{E}$, and failure detector history $H_D \in D(F)$. We shall first construct an infinite directed acyclic graph, denoted G, whose vertices are some of the failure detector values that occur in H_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 $\mathit{Consensus}_D$. 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 it 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 $\mathit{Consensus}_D$, we mean a run of $\mathit{Consensus}_D$ using D. Furthermore, we only consider schedules of $\mathit{Consensus}_D$, and therefore we write (p, m, d) instead of $(p, m, d, \mathit{Consensus}_D)$ to denote a step.

6.1 A DAG and a forest

Given the failure pattern F and the corresponding failure detector history $H_D \in D(F)$ 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 $d = H_D(p, t)$ for some time t.

2. If $[q_1, d_1] \rightarrow [q_2, d_2]$ is an edge of G and $d_1 = H_D(q_1, t_1)$ and $d_2 = H_D(q_2, t_2)$ 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, $[p', d'] \rightarrow [p, d]$ is an edge of G.

Note that such a DAG represents only a "sampling" of the failure detector values that occur in H_D. In particular, we do not require that it contain all the values that occur in H_D or that it relate (with an edge) all the values according to the time at which they occur.

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 $m_1, m_2, \ldots$; S is *compatible with* G if it is compatible with some path of G. S is *induced by g and an initial configuration I* (of $Consensus_D$) if S is compatible with g and applicable to I. S is *induced by G and I* if S is compatible with G and applicable to I. Note that each g and I induce several schedules, each corresponding to a different sequence of messages received.

Lemma 5: Let S be any finite schedule induced by G and some initial configuration I of $Consensus_D$. There is a T such that $\langle F, H_D, I, S, T \rangle$ is a partial run of $Consensus_D$.

Lemma 6: Let S be any infinite schedule induced by G and some I, such that every correct process takes an infinite number of steps and every message sent to a correct process is eventually received. There is a T such that $\langle F, H_D, I, S, T \rangle$ is a run of $Consensus_D$.

The set of schedules that are induced by G and some particular I, can be organized as a tree, the *simulation tree Υ_G^I induced by G and I*. These schedules are the vertices of the tree, with (the empty schedule) $S_\perp$ at the root. There is an edge from S to S' if and only if $S' = S \cdot e$ for a step e.

Lemma 7: 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. Υ_G^I has a vertex $S \cdot (p, m, d)$ for some d.

Lemma 8: Let $S, S_1, S_2, \ldots, S_k$ be vertices of Υ_G^I. There is a 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. $S_i \cdot E$ $(1 \leq i \leq k)$ is compatible with G.

Note that E may not be applicable to $S_i(I)$, and thus $S_i \cdot E$ is not necessarily a vertex of Υ_G^I.

Let I^i, $0 \leq i \leq n$ denote the initial configuration of $Consensus_D$ in which the initial values of $p_1 \ldots p_i$ are 1, and the initial values of $p_{i+1} \ldots p_n$ are 0. We define the *simulation forest induced by G* to be the set of $n+1$ simulation trees induced by G and these initial configurations.

6.2 Tagging the simulation forest

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

Lemma 9: Each vertex of Υ^i has at least one tag.

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

Lemma 11: If S is a bivalent vertex of Υ^i then no correct process has decided in $S(I^i)$.

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

Lemma 12: The root of Υ^0 is 0-valent and the root of Υ^n is 1-valent.

If the root of Υ^i is bivalent, then i is *bivalent critical.* If the root of Υ^{i-1} is 0-valent but the root of Υ^i is 1-valent, then i is *monovalent critical.* Index i is *critical* if it is monovalent or bivalent critical.

Lemma 13: There is a critical i, $0 < i \leq n$.

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 15). 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 $S_\perp$ 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 2. 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.

The second type of decision gadget is called a *hook*, and is shown in Figure 3. 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 gadget must be correct (Lemma 16). 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 D. Despite this, the hiding of the step of the deciding process of a gadget is still possible. The key to proving this is Lemma 8.

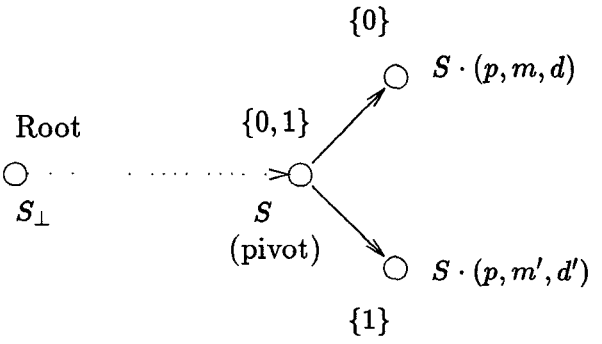

Figure 2: A fork—p is the deciding process

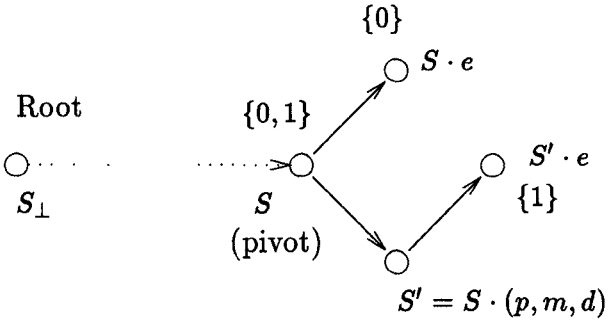

Figure 3: A hook—p is the deciding process

Lemma 14: If i is bivalent critical then Υ^i has at least one decision gadget (and hence a deciding process).

6.4 Extracting the correct process

By Lemma 13, there is a critical index i. If i is monovalent critical, Lemma 15 below shows how to extract a correct process. If i is bivalent critical, a correct process can be found by applying Lemmata 14 and 16.

Lemma 15: If i is monovalent critical then p_i is correct.

Lemma 16: The deciding process of a decision gadget is correct.

There may be several critical indices and several

{*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
if i is monovalent critical **then return** p_i
else return deciding process
    of the smallest gadget in Υ^i

Figure 4: Selecting a correct process

decision gadgets in the simulation forest. Thus, the above Lemmata 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" 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 gadget is just a finite graph, the selection rule can use any such encoding. The selection rule is shown in Figure 4.

Lemma 17: Figure 4 selects a correct process.

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

The selection of a correct process described above is not yet the distributed algorithm $T_{D \to \Omega}$ that we are seeking: it involved an infinite simulation forest and it was "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 gadget and the other tagging information necessary to extract the identity of the *same* correct process chosen by the selection method in Figure 4.

Note that the selection method in Figure 4 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 $T_{D \to \Omega}$ 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 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 $T_{D \to \Omega}$.

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 $T_{D \to \Omega}$.

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, each process p repeatedly executes: (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 $T_{D \to \Omega}$ for p is shown in Figure 5.

Recall that we are considering a fixed run of $T_{D \to \Omega}$, with failure pattern F, and failure detector history $H_D \in D(F)$. The communication component of $T_{D \to \Omega}$ constructs graphs that satisfy the following properties. Let $G_p(t)$ denote the value of G_p at time t.

Lemma 18: For any correct process p and $t \in \mathcal{T}$:

1. The vertices of $G_p(t)$ are of the form $[p', d']$

{*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 D
  SEND PHASE:
    if m is of the form (q, G_q, p) **then**
      $G_p \leftarrow G_p \cup G_q$
    add $[p, d_p]$ to G_p and edges from all other vertices of G_p to $[p, d_p]$
    $output_p \leftarrow$ *computation component* {*Fig. 6*}
    p sends (p, G_p, q) to all $q \in \Pi$

Figure 5: Process p's communication component

where $d' = H_D(p', t')$ for some time t'.

2. If $[q_1, d_1] \rightarrow [q_2, d_2]$ is an edge of $G_p(t)$ and $d_1 = H_D(q_1, t_1)$ and $d_2 = H_D(q_2, t_2)$ 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 $G_p(t)$, $[p', d'] \rightarrow [p, d]$ is an edge of $G_p(t')$.

5. $G_p(t)$ is a subgraph of $G_p(t+1)$.

6. For all correct q, there is a time $t' \geq t$ such that $G_p(t)$ 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 19: For any correct processes p and q, $G_p^\infty = G_q^\infty$.

Lemma 19 allows us to define the *limit graph* G to be G_p^∞ for any correct process p. The first four properties of Lemma 18 imply:

Lemma 20: The limit graph G satisfies the four properties of the DAG defined in Section 6.1.

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 4 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 $T_{D \rightarrow \Omega}$, 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 4: 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 4. If p is faulty, we do not care. Also, p cannot use exactly the same selection method as that of Figure 4: its current simulation forest Υ_p may not *yet* have a critical index or contain any deciding gadget (although it eventually will!). In that case, p temporizes by just selecting itself. The computation component of $T_{D \rightarrow \Omega}$ is shown in Figure 6. Let $\Upsilon_p(t)$ denote Υ_p at time t.

Lemma 21: For any correct p and any $t \in \mathcal{T}$:

1. $\Upsilon_p(t)$ is a subgraph[6] of Υ
2. $\Upsilon_p(t)$ is a subgraph of $\Upsilon_p(t+1)$
3. $\lim_{t \rightarrow \infty} \Upsilon_p(t) = \Upsilon$

[6] The subgraph relation ignores the tags.

```
{Build and tag simulation forest Υ_p induced by G_p}
for i ← 0, 1, ..., n:
    Υ_p^i ← 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 ← smallest critical index
    if i is monovalent critical then return p_i
    else if Υ_p^i has no gadgets then return p
        else return deciding process
                    of the smallest gadget in Υ_p^i
```

Figure 6: Process p's computation component

Lemma 22: For any correct p and any vertex S of Υ_p:

1. p never removes a tag from S.
2. There is a time after which the tags of S in Υ_p will always be the same as the tags of S in Υ.

Theorem 23: For any correct process p, there is a time after which $output_p = p^*$, forever.

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

References

[ABND+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.

[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.