The Failure Detector Abstraction

Felix C. Freiling, University of Mannheim
and
Rachid Guerraoui, EPFL and MIT CSAIL
and
Petr Kuznetsov, TU Berlin/Deutsche Telekom Laboratories

A *failure detector* is a fundamental abstraction in distributed computing. This paper surveys this abstraction through two dimensions. First we study failure detectors as building blocks to simplify the design of reliable distributed algorithms. In particular, we illustrate how failure detectors can factor out timing assumptions to detect failures in distributed agreement algorithms. Second, we study failure detectors as computability benchmarks. That is, we survey the weakest failure detector question and illustrate how failure detectors can be used to classify problems. We also highlight some limitations of the failure detector abstraction along each of the dimensions.

Categories and Subject Descriptors: A.1 [**General Literature**]: Introductory and Survey; C.4 [**Computer Systems Organization**]: Performance of Systems—*fault tolerance; modeling techniques; reliability, availability, and serviceability*

General Terms: Algorithms, Design, Reliability, Theory

Additional Key Words and Phrases: distributed system, agreement problem, consensus, atomic commit, fault tolerance, liveness, message passing, safety, synchrony

The first author's work was supported by the Deutsche Forschungsgemeinschaft (DFG) as part of the Emmy Noether programme.
Contact author's address: University of Mannheim, Department of Computer Science, D-68131 Mannheim, Germany, contact author email: `fcg@acm.org`

Contents

1. INTRODUCTION

Advances in computing are typically achieved through the identification of abstractions that factor out specifics of an actual processor, machine or network. In the early times, abstractions like record, set or arrays helped the emancipation from assemblies and machine languages. The art of traditional sequential and centralized computing was then orchestrated around such data structures.

Progress in computer architectures called then however for new abstractions. In the area of concurrent computing for instance, abstractions like threads, semaphores and monitors were very helpful in understanding concurrent programs and reasoning about their correctness. In the area of distributed computation, the remote procedure call abstraction helped factor out the details of the network and was a key to the popularity of standard distributed middleware infrastructures. In short, the remote procedure call abstraction hides many differences between languages and operating systems on different machines, and encapsulate serialization and deserialization mechanisms to transfer data over the wire.

The remote procedure call does not however help capture another fundamental characteristic of distributed systems: partial failures. Basically, if a process of some machine remotely invokes an operation on a process performing on a different machine, and the latter machine fails, an exception is raised. The way the failure is detected is usually achieved using a timeout mechanism. Typically, a timeout delay is associated with the operation and when it expires, the exception is raised.

Programming with timeouts is however difficult and it hampers portability. The adequate way of choosing the duration of a timeout might vary from a system to another one, and might even dynamically depend on the load of the system. Sometimes it also is more appropriate to ping processors whereas sometimes it is better to require that they initiate heartbeat messages.

Basically, *failure detectors* are abstract devices that offer information about the operational status of processes in a distributed system [Chandra and Toueg 1996]. We believe that the failure detector abstraction is a fundamental one and should sit as a first class citizen of a distributed programming library. In fact, and as we discuss in this paper, the failure abstraction can also help classify problems in distributed computing [Chandra et al. 1996].

This paper is structured into two parts: the first part (Section 2) looks at failure detectors from an engineering point of view and discusses the advantages of using failure detectors in the design, programming and analysis of distributed algorithms. It also discusses inherent limitations of the failure detector abstraction.

The second part (Section 3) takes a more theoretical perspective and discusses the role that failure detectors can play to compare and distinguish problem specifications in distributed systems. We describe how the hardness of a problem can be measured by determining the weakest failure detector needed to solve the problem, and we illustrate this approach by several examples of the "weakest failure detector" proofs.

Several surveys about distributed programming with failure detectors have ben published. [Raynal 2002; Guerraoui et al. 1999; Raynal 2005]. Our survey is however more complete as it also covers the the theoretical aspect of failure detectors that we believe is equally important but much less understood.

2. FAILURE DETECTORS AS PROGRAMMING BUILDING BLOCKS

Information about the operational status of remote processes is often necessary to implement reliable distributed services. In this section we argue that the failure detector abstraction is a sensible one from an engineering point of view.

In Section 2.1 we first review the problems of implementing failure detection based on timeouts. In Section 2.2 we informally introduce the failure detector abstraction and argue that it has several advantages over the explicit use of timeouts: (1) It separates the concerns of reasoning about failures and reasoning about time and therefore makes programs simpler to write and analyze; (2) It helps express information about failures in a way that is closer to the control logic of many applications, so it allows to write simpler and more elegant programs; (3) It allows independent implementation and service sharing and therefore has the potential of building more efficient applications. In Section 2.3 we discuss some of the limitations of the failure detector approach.

2.1 Failure Detection using Timeouts

2.1.1 *Example of a Distributed Problem: Non-Blocking Atomic Commit (NBAC).* We introduce our subject through this seminal database problem [Bernstein et al. 1987, Chapter 7] where data is distributed over multiple geographically separated processes. At the end of a transaction on that data, these processes must decide whether the actions should be committed (made permanent) or aborted (rolled back). More precisely, at the end of the transaction each participating process votes *yes* ("I am willing to commit") or *no* ("we must abort"), and eventually processes must reach a common decision, *commit* or *abort*. A non-blocking atomic commit protocol ensures that the following properties hold:

(1) All processes that manage to reach a decision on the outcome of the transaction agree on the decision.
(2) A process cannot reverse its decision.
(3) A *commit* decision can only be reached if all processes vote *yes*.
(4) If all processes vote *yes* and there are no failures, then the decision must be *commit*.
(5) Assuming that there are only expected failures, every (surviving) process must eventually reach a decision.

The terms "expected failures" and "surviving process" in the fifth clause refer to the particular failure assumption made by the system designers. In practical settings, this often translates to rarely occurring benign *crash* failures of processes with subsequent repair and *recovery*. For simplicity, unless explicitly stated otherwise, we will disregard recovery, i.e., we assume that a failed process simply stops to execute steps of its algorithm and does not send or receive messages anymore (messages sent to crashed processes are lost). We will call a process that does not crash a *correct* process.

2.1.2 *Example of a Distributed Protocol: Three-Phase Commit (3PC).* This is a well-known protocol to implement non-blocking atomic commit. 3PC refines the popular *two-phase commit* (2PC) protocol, that is widely used in distributed

databases (although 2PC is a *blocking* protocol, i.e., it does not satisfy the final termination property of non-blocking atomic commit in every execution). 3PC makes use of a particular coordinator process.

From now on, assume that there are n processes called $p_1, p_2, \ldots, p_n$ ($n > 1$) and that process p_1 plays the role of the coordinator. In general, 3PC works as follows [Bernstein et al. 1987, p. 242]:

(1) The coordinator p_1 sends a vote request to all other processes.
(2) When a process receives a vote request, it responds with either *yes* or *no*, depending on its vote. If it sends *no* it decides *abort*, and stops.
(3) The coordinator collects his own vote and the vote messages from all other processes. If any of these votes was *no* then the coordinator decides *abort*, sends an abort message to all processes that voted *yes*, and stops. Otherwise the coordinator sends a pre-commit message to all processes.
(4) A process that votes *yes* waits for a pre-commit message or an abort message. If it receives an abort message, it decides *abort* and stops. If it receives a pre-commit message, it responds with an acknowledgement to the coordinator.
(5) The coordinator collects the acknowledgements from all processes. When they have all been received, he decides *commit*, sends a commit message to all processes, and stops.
(6) Other processes wait for the commit message from the coordinator. When they receive this message, they decide *commit*, and stop.

In the absence of failures, it is rather easy to see that the protocol satisfies the five requirements of the non-blocking atomic commit problem. However, there are several points in the protocol in which crash failures can cause a process to wait indefinitely for a message and hinder correct processes from reaching a decision. In practice, this is prevented using *timeouts*.

2.1.3 *Three-Phase Commit with Timeouts.* It makes no sense to wait for a message from a crashed process, because this might violate liveness properties of the system. So how can we find out whether a remote process is still operational or not? A pragmatic way is to monitor the time it takes for a process to send a reply. The *round-trip delay* is a network parameter that denotes the time it takes to send a message to a remote process and receive an answer from that process. Usually it is safe to assume a time interval ρ as an upper bound on the round-trip delay, meaning that if a reply has not arrived after ρ time has elapsed since sending, then the remote process is not operational anymore. In this case we say that the process *times out* after ρ time units and ρ is the *timeout interval* (or simply *timeout*).

Every statement in the 3PC algorithm that could potentially block needs to be enhanced with a timeout construct. For example, in step 2, processes wait for a vote request from the coordinator. This specific part of the algorithm is depicted at the top of Fig. 1. If a process fails to receive such a message, it unilaterally decides *abort* since it could have forced this decision through its own vote anyway. In the algorithm, the process needs to monitor the time and wait until the timeout period ρ has elapsed. In case this happens, a timeout action is activated. This is shown at the bottom of Fig. 1 where the variable *clock* refers to the value of the real-time clock of that process.

wait for ⟨vote request⟩ **from** p_1

(a)

timeout := *clock* + ρ
while *timeout* > *clock* ∧ ¬⟨vote request arrived from p_1⟩ **do**
  skip
if ¬⟨vote request arrived from p_1⟩ **then**
  ⟨decide *abort*⟩

(b)

Fig. 1. Replacing a potentially blocking program statement in three-phase commit (a) with explicit timeout actions (b).

2.1.4 *Difficulties of Determining Good Timeout Values.* The timeout intervals are usually real-time instances obtained from analyzing the characteristics of the underlying network. In fixing the timeout value ρ, there is a notorious tradeoff between correctness and efficiency. In order to not time out too early (i.e., when the remote process is not crashed), we would like to set ρ very conservatively, i.e., make it very large. However, a large value of ρ means that the protocol blocks for a very long time before making progress again in case of failure. The guideline is to make ρ as large as necessary but as small as possible.

Determining good timeout values still poses problems even to experienced engineers. The main reason for this is that ρ can only be determined with certainty in networks that offer certain real-time guarantees and most networks in use today (like local area Ethernets or the global Internet) do not fall into this category. As an extreme example, measurements of round-trip delays on the Internet for many years [Long et al. 1991; Paxson and Adams 2002] consistently show that there is a large temporal and spatial variation in round-trip delays and the distribution is asymmetric with a long tail on the right hand side. On the one hand, this means that fixing a large timeout value does not necessarily always guarantee correctness of timeout-based reasoning, it merely decreases the probability of making mistakes. On the other hand, continued premature expiration of a too small timeout may prevent any transaction from committing successfully.

In practice, determining good timeouts even has a dynamic aspect. For example, it makes sense to wait longer (i.e., have larger timeout values) during the beginning of the 3PC algorithm in order to increase the probability of all processes voting *yes*; for this to happen, they should not time out prematurely on the vote request from the coordinator. Towards the end of the algorithm, e.g., when the coordinator is about to broadcast the commit message, shorter timeouts are feasible since the result of the transaction has been determined already. In this case, a wrong but quick suspicion of a slow coordinator may even speed up the transaction if that coordinator is replaced by a very fast one.

2.1.5 *Synchronous Systems.* It is possible to characterize those systems in which timeout-based reasoning is always correct. These systems are characterized by bounds on the two critical system parameters: the message delivery delay and the relative processing speed difference. We will denote these bounds by δ (processing speed bound) and Δ (message delivery delay bound).

Instead of measuring these bounds in terms of real-time, we will take a more abstract view and measure it in terms of the number of *steps* that a process has executed. On the one hand, this abstraction allows to express time bounds better in the common models of distributed algorithms where processes do not have an external source of real-time [Dwork et al. 1988; Dolev et al. 1987]. On the other hand, the abstraction does not limit the generality of the following statements since it is possible to relate the number of steps of a process to real-time intervals in practice.

The bounds δ and Δ have the following meaning:

—Processing speeds: In the time it takes for any process to take δ steps, all other processes must take at least one step.

—Message delivery delay: If a process sends a message m after step k in the execution, then m must be delivered after at most $k + \Delta$ execution steps of the sending process.

A bound ρ on the round-trip delay between two processes p and q can be computed from δ, Δ and an additional parameter s as follows:

$$\rho = \Delta + s \cdot \delta + \delta \cdot \Delta$$

First, it takes at most Δ steps of the sending process p for the message to travel from p to q. The parameter s denotes the maximum number of steps needed by process q to receive a message and send a reply. Therefore, after at most $s \cdot \delta$ steps of process p the receiving process q must have executed at least s steps (which includes receiving the message and sending a reply). Finally, after at most δ steps of process q the message must arrive back at p. Since q may operate much slower than p and δ is measured in steps of q, we need to multiply δ with Δ, yielding an upper bound on the time it takes for q to execute δ steps.

Because of the bound Δ on relative processing speeds it is possible for any process to give bounds on the number of steps any other process in the system has executed. This means that there exists a notion of global time in the system. Because of this, these systems are called *synchronous*. In synchronous systems it is possible to determine "perfect" global timeout values. If specific values of δ and Δ are known for individual pairs of processes, it is even possible to compute perfect *local* timeout values.

In practice, the aforementioned synchrony conditions are usually expressed in terms of real time. For this it is assumed that events in the system can be related to some external source of real time. Then the bound δ is the real-time interval in which processes have to take at least one step, and Δ refers to that real-time interval within which every message must be delivered. Concerning processing speeds, it is sometimes assumed that processes have access to local hardware clocks and that δ is a bound on the drift rate of these clocks. However this does not mean that the system is synchronous in the sense above [Cristian and Fetzer 1999]. That local clocks advance at a steady rate does not mean that processes advance equally within their local algorithms.

2.1.6 *Asynchronous Model and Timeouts.* As mentioned above, having a synchronous system is not realistic in many practical situations. In fact, from an

engineering perspective it makes sense to make very little assumptions about the underlying network characteristics because this achieves the highest *assumption coverage* [Powell 1992]. Assumption coverage refers to the probability that the assumptions about the underlying network hold in a particular mission environment. More and stronger assumptions (e.g., about synchrony) achieve less assumption coverage, and only a high assumption coverage ensures that the algorithms (e.g., reasoning with timeouts) work as expected in practice.

The highest assumption coverage (with respect to synchrony) is achieved by system models that have no timing assumptions whatsoever. These systems are usually referred to as *time-free* [Cristian and Fetzer 1999] or *asynchronous* [Fischer et al. 1985] (see Schneider [1993] for a discussion of these models). They can be characterized by the following basic statements:

—A system is modeled as a set of processes connected by reliable communication channels.

—Communication is by point-to-point message passing using *send* and *receive* primitives.

—Usually it is assumed that the network is fully connected, i.e., every process can directly send messages to every other process.

—There is no order on delivery of messages through the channels.

—*Receive* and *send* are distinct atomic operations.

—There is no bound on relative processing speeds of processes and on the message delivery delays.

Note that reliable communication does not contradict high assumption coverage. Using standard transport layer network protocols [Tanenbaum 1996], message delivery can be guaranteed albeit without any upper time bound. This and the other points above result in two things: Firstly, messages can take an arbitrary (but finite) time to travel from one process to the other. This means that a message sent in step k during the execution of an algorithm will be received at its destination after arbitrarily (but finitely) many execution steps following k. Secondly, processes can be arbitrarily slow, meaning that in the time it takes a process p to take a single step, another process q can take any finite number of steps. The Internet is often taken as an example for an asynchronous system.

Unfortunately, because of the absence of any synchrony assumptions, it is impossible to do timeout-based reasoning in asynchronous systems. To see this, consider a process p that monitors the operational state of a process q using timeouts. Since there are no bounds δ and Δ we cannot use the formula above to calculate a timeout bound ρ. So even if p sets its timeout to a very large (finite) value the round-trip time to q can exceed any finite value. It is merely guaranteed that (if q does not crash), the reply will eventually arrive at p so at least p will eventually learn that it might have performed incorrect timeout-based reasoning [Garg and Mitchell 1998a].

2.1.7 *Eventually Synchronous Systems.* It is a common experience that networks perform synchronously "most of the time". This means that the system alternates between short periods of instability (i.e., where no timing guarantees can be made) and long periods of stability (i.e., where the system behaves as if it

were synchronous). Measurements by Cristian and Fetzer [1999] have shown that the average length of a stable period is several magnitudes longer than that of an unstable period in standard local area networks. The bottom line of this observation is that timeout-based reasoning can be performed perfectly "most of the time".

From measurements and experience, the average length of a stable period is in the order of minutes or hours, a time that is usually sufficient for an algorithm to terminate, e.g., a transaction to commit. Therefore, an arguable assumption is that the system is synchronous "forever" after an initial finite time of asynchrony. This is captured in the model of eventual synchrony that became known as the assumption of *partial synchrony* [Dwork et al. 1988].

There are two possible variants of partially synchronous communication, that we will exemplify using the communication bound Δ:

(1) Either Δ is known but holds only eventually, or

(2) Δ exists but is not known.

Analogous definitions of partially synchronous processes can easily be derived using bound δ instead of Δ.

Both variants of partial synchrony reflect the difficulty of choosing a system's timing parameters in practice. The first form, namely that timing bounds hold eventually, directly reflects the findings from the study of Cristian and Fetzer [1999] because it is highly improbable that an algorithm starts in a stable period and ends in an unstable period. The second variant reflects the fact that it is often safe to assume that some upper bound on message delivery time exists; the difficult question is how large this bound actually is.

The model of partial synchrony has found many refinements, starting with Chandra and Toueg [1996] where both communication *and* processes are partially synchronous (regardless of the form of partial synchrony, 1 or 2 above [Dwork et al. 1988]), followed by Hermant and Widder [2005] where merely the ratio between best-case and worst-case round trip delay is bounded, or Fetzer et al. [2005] where the average response time must be bounded. All of these models eventually allow perfect timeout-based reasoning.

2.1.8 *Conclusions.* In many practical situations (like in distributed databases with atomic transactions) it is necessary to know the operational state of a remote process. The most common way to get this information is to use timeout-based reasoning. Choosing correct timeout values, however, is a difficult task and timeout-based reasoning is only valid in systems with strict timing guarantees. This is unfortunate since this weakens the usefulness of timeout-based reasoning exactly in those (asynchronous) systems that are considered most common in practice. One path out of this dilemma is to use the abstraction of a *failure detector*, that separates timeout-based reasoning from the necessity to choose concrete timeouts. We discuss this abstraction in the following sections.

2.2 Failure Detectors as Useful Distributed Services

In the previous sections we have already talked about using synchrony bounds for detecting crash failures of processes. However, this approach mixes two separate

concerns:

(1) the abstract functionality of detecting process crashes, and
(2) a way to implement this abstract functionality using synchrony bounds.

As we show later, we can write a correct 3PC algorithm in systems without any explicit synchrony assumptions as long as we have a means to still detect process crashes.

2.2.1 *Failure Detectors as Oracles.* Separating the concerns of timeout-based reasoning and the detection of failures is at the heart of the *failure detector abstraction*, that has been introduced by Chandra and Toueg [1996]. Failure detectors are oracles that produce (possibly incomplete and unreliable) information about the operational state of processes.

In the understanding of Chandra and Toueg [1996], a failure detector is composed of several failure detector modules, one at each process. Failure detectors have an interface (defining their syntax) and guarantee certain properties (defining their semantics). In its original form, a failure detector indicates the operational state (*up*/*down*) of some other process. If the failure detector responds with *down*, we say that the failure detector at *suspects* that process.

2.2.2 *Perfect Failure Detectors.* To define the semantics of failure detectors, consider a process p that is equipped with a failure detector module that indicates the operational state of a process q. Similar to other types of detectors in distributed systems (such as termination detectors [Dijkstra et al. 1983] or general predicate detectors [Chandy and Misra 1988; Arora and Kulkarni 1998]) the failure detector module at p should guarantee two things:

—It never suspects q unless q has actually crashed, and
—if q has crashed, then the failure detector module at p will eventually permanently suspect q to have crashed.

Of course, these guarantees can only be given as long as p is alive. We can then extend the failure detector to the two process case by simply adding a failure detector module to q and requiring that each module detect the crash of the other process.

Similarly, the definition of such a failure detector can be extended to the n process case. Here it is important to note that every failure detector module at every process is responsible for checking the operational states of *all* other processes in the system. This means that the output of such a failure detector module is a general predicate involving all other processes. The failure detectors introduced by Chandra and Toueg [1996] output a list of *suspected* processes, but other forms have also been proposed (as will be seen later).

For an n process system, the first requirement of the failure detector is a composition of all safety requirements of the individual failure detector modules. It can be read like this:

—for all processes p: for all processes q:
p has not crashed $\Rightarrow$ the failure detector module at p does not suspect q unless q has crashed.

The liveness requirement can be reformulated as:

—for all processes p: for all processes q:
p has not crashed $\Rightarrow$ if q crashes, then the failure detector module at p will eventually permanently suspect q.

When referring to failure detectors, Chandra and Toueg [1996] call the above safety property *strong accuracy* and the liveness property *strong completeness*. A failure detector satisfying strong accuracy and strong completeness is called a *perfect failure detector*. The class of all perfect failure detectors is usually denoted by $\mathcal{P}$. If there is no confusion, we sometimes also denote by $\mathcal{P}$ some failure detector from this class. A perfect failure detector makes no wrong suspicions and eventually detects every crash.

2.2.3 *Asynchronous Models with Failure Detectors.* It is important to stress that a failure detector is merely defined through the service it offers, not by the way it is implemented. Of course, failure detection will most probably be implemented using timeouts in practice, but the failure detector cleanly hides the details of the underlying system model and its synchrony bounds behind its service interface. The interface of the failure detector "looks time-free" and so it makes sense to combine the asynchronous model with failure detectors and design algorithms in this new model. Of course, this model is not purely asynchronous anymore (many authors therefore write that the asynchronous model is *augmented* with failure detectors), but it allows to describe and analyze algorithms as if they were running in an asynchronous model. A failure detector can therefore be regarded as a device that encapsulates synchrony assumptions in an asynchronous interface.

2.2.4 *Non-Blocking Atomic Commit with a Perfect Failure Detector.* As an example on how to write algorithms using failure detectors, Figure 2 shows the part of the 3PC algorithm from Fig. 1 using a perfect failure detector. The failure detector abstraction simplifies the text of the protocol by removing concrete timeout values. In fact, this is similar to the descriptions commonly found in books on concurrency control (like the one by Bernstein et al. [1987]). There, every receive (or **wait for**) statement is accompanied by an **on timeout** clause specifying what to do when the timer for this statement elapses. In a sense, these algorithm descriptions already use a failure detector abstraction to simplify the writing of the protocols without naming it.

Note that now the correctness of the protocol can be analyzed without referring to timeouts or synchrony bounds. For example, if there are no crashes, the strong accuracy property of the failure detector ensures that no process will be suspected and so the protocol behaves like the 3PC protocol for the fault-free case. Similarly, if the coordinator crashes before sending out vote requests, the strong completeness of the failure detector guarantees that every process will eventually stop waiting for a message from the coordinator and advance in the protocol. Overall, reasoning about the correctness of the algorithm becomes much simpler.

2.2.5 *Solving Consensus using Failure Detectors.* To illustrate another advantages of failure detectors, consider the problem of *consensus* (see Barborak et al. [1993] and Turek and Shasha [1992] for surveys on consensus). Like non-blocking

wait for ⟨vote request arrived from p_1 or $p_1 \in \mathcal{P}$⟩
if $p_1 \in \mathcal{P}$ **then**
  ⟨decide *abort*⟩

Fig. 2. Algorithm code from Fig. 1 using a perfect failure detector $\mathcal{P}$.

atomic commit, consensus belongs to the class of agreement problems where processes must take a consistent decision starting from inconsistent values. The consensus problem is defined using two primitives called *propose* and *decide*. Both take an argument from a fixed set of decision values (usually $\{0, 1\}$). If a process invokes *propose*(u) we say that it proposes u. Analogously, if it invokes *decide*(v) we say that it decides v. A process may decide at most once. In general, an algorithm that solves the consensus problem must guarantee three properties:

—(Agreement) No two processes decide different values.
—(Termination) Every correct process eventually decides.
—(Validity) The decided value must have been proposed by some process.

The Validity property is a non-triviality property, meaning that it has been added to exclude trivial solutions where processes do not communicate (e.g., algorithms where every process always decides 1). More specifically, the above consensus specification is called *uniform consensus* [Hadzilacos and Toueg 1994] because it mandates that all processes (i.e., even the faulty ones) do not disagree on the decision value.

Similar to non-blocking atomic commit, consensus can be solved rather easily using a perfect failure detector. However, consensus can be solved even if the failure detector is "imperfect", i.e., if it can make mistakes. An example of such a failure detector is the eventually perfect failure detector (denoted $\diamond\mathcal{P}$), a failure detector that is only perfect after some finite time (before this time it can behave arbitrarily). The idea of a consensus algorithm using $\diamond\mathcal{P}$ is to be conservative, i.e., maintain the safety aspect of consensus (Agreement and Validity) always, and only terminate if the failure detector stops making mistakes. Such algorithms are called *indulgent* [Guerraoui 2000] because they are indulgent towards (the mistakes of) their failure detector.

The indulgent consensus algorithms from the literature [Chandra and Toueg 1996; Dwork et al. 1988; Schiper 1997a; 1997b; Hurfin and Raynal 1999; Brasileiro et al. 2000; Freiling and Völzer 2006] operate in a sequence of rounds. Every round is like the first round of the 3PC algorithm sketched above: a coordinator tries to impose a decision value on all other processes, only that the role of the coordinator changes every round in round-robin fashion. This protects against relying on some crashed process to be the coordinator. However, due to the unreliability of the failure detector, a correct process may not get its chance to succeed in imposing a value on the rest of the system (the others might have suspected him and advanced to the next round with a different coordinator). In this situation it must be ensured that no two processes can impose different decision values onto the system (and cause disagreement). To prevent this, the algorithms require a coordinator to "lock" a value before decision. Locking a value ensures that no other value can become the decision value.

In locking a value v, all values other than v are "extinguished" from the system. Assuming that a majority of processes is correct, this can be achieved by using the following protocol (inspired by majority voting or the use of quorums for replica control in databases [Bernstein et al. 1987]): The first coordinator sends its value v to all processes. Whenever a process receives v, it *adopts* v and acknowledges it back to the coordinator. The process also remembers the time (i.e., the round number) in which it adopted v. Later coordinators must impose the *latest adopted value* from all processes. This ensures that if a majority of processes has adopted value v in the same round, then no other value other than v can be imposed by any future coordinator.

It is rather easy to show that there is no indulgent consensus algorithm if more than half of the processes can be faulty [Chandra and Toueg 1996; Guerraoui 2000]. A set of n processes can become "virtually partitioned" by information resulting from wrong suspicions by the failure detector. This means that there can be two small subsets of processes that suspect all other processes (including those of the other set) to have crashed. In such a case, each partition can decide different values, thus violating safety. This situation cannot arise if we have the algorithm guarantee that every deciding partition must include the majority of processes. In this way no two partitions can decide differently because they must have a common process in both. In a sense, requiring a correct majority is the price you have to pay for making mistakes in detecting crashes.

Interestingly, consensus can even be solved with a failure detector that (in a precise sense which is defined later in this article) is even weaker than $\diamond\mathcal{P}$ [Chandra and Toueg 1996]. Like $\diamond\mathcal{P}$, this failure detector, called "eventually strong" (denoted $\diamond\mathcal{S}$), belongs to the class of unreliable failure detectors introduced next.

2.2.6 *Unreliable Failure Detectors.* The existence of a perfect failure detector is a very strong assumption that makes the model no more realistic than one where explicit synchrony bounds are added. As shown above, a perfect failure detector is also not always necessary. This motivates looking for weaker assumptions about the failure detector modules.

We can derive several weaker failure detectors by relaxing the completeness and accuracy properties of the perfect failure detector, e.g.:

—$\exists$ a correct process $q : \forall p$: the failure detector module at p does not suspect q.
This means there is a correct process which all processes will not falsely suspect. So every process except one may be infinitely often falsely suspected to have crashed. This property is called *weak accuracy* [Chandra and Toueg 1996] and is even useful if it only holds eventually (this is termed *eventual weak accuracy*).

—$\forall q : \exists$ a correct process p : if q crashes then the failure detector module at p will eventually permanently suspect q.
In terms of liveness this means that for every crash, there is a process that will detect this crash. This is termed *weak completeness* [Chandra and Toueg 1996]. Obviously, if at least one process eventually detects the crash of some process q, then eventually all processes can be made to detect that crash by simply disseminating the information throughout the network. Thus it is possible to turn a weakly complete failure detector into a strongly complete failure detector

if there are means to reliably disseminate information in the network.

A failure detector satisfying weak completeness and eventual weak accuracy is called an *eventually weak failure detector*. All other combinations of failure detectors and their names are depicted in Table I. All failure detectors that are allowed to make mistakes fall into the category of *unreliable failure detectors*. A general, yet precise definition of the notion of unreliable failure detector was given by Guerraoui [2000].

	accuracy			
	strong	weak	eventually strong	eventually weak
strong completeness	perfect $\mathcal{P}$	strong $\mathcal{S}$	eventually perfect $\diamond\mathcal{P}$	eventually strong $\diamond\mathcal{S}$
weak completeness	quasi perfect $\mathcal{Q}$	weak $\mathcal{W}$	eventually quasi perfect $\diamond\mathcal{Q}$	eventually weak $\diamond\mathcal{W}$

Table I. The failure detector classes of Chandra and Toueg [1996].

2.2.7 *Other Failure Detectors.* Other failure detectors have been defined with different motivations. We give here a brief selection: Chandra et al. [1996] introduced the failure detector Ω which eventually outputs the identity of a correct process that is trusted by everybody. This failure detector was shown to be equivalent to $\diamond\mathcal{S}$ in the proof that it is the weakest to solve consensus [Chu 1998]. Aguilera et al. [2000b] presented a failure detector called *heartbeat* that is useful in designing protocols that are *quiescent*, i.e., that eventually stop sending messages. Garg and Mitchell [1998b] define the *infinitely often accurate* failure detector (denoted $\Box\diamond\mathcal{P}$) in the context of predicate detection in faulty systems [Garg and Mitchell 1998a]. The *failure signal* failure detector $\mathcal{FS}$ [Delporte-Gallet et al. 2004], originally called *anonymously perfect failure detector* [Guerraoui 2002] and considered also in Charron-Bost and Toueg [2001a], can be used for solving nonblocking atomic commit. $\mathcal{FS}$ is just like a perfect failure detector, only that it does not output the identities of the failed processes; it merely outputs a boolean value whether or not *some* process has crashed. We will return to this failure detector later (in Sections 2.2.10 and 3.3.2).

2.2.8 *Justifying Unreliable Failure Detectors.* Assuming unreliable failure detectors is much more realistic than assuming a perfect failure detector, because the properties of unreliable failure detectors can be more easily guaranteed in practice than those of perfect failure detectors. This is because systems with eventual synchrony (like partially synchronous systems) allow to implement such failure detectors.

For example, $\diamond\mathcal{P}$ can be implemented in partial synchrony in the following way: The failure detection algorithm starts with an approximation ρ' of the real (but unknown) timeout value ρ; whenever the algorithm notices that it made a mistake (it receives a message from a suspected process), ρ' is increased. Once the system has become synchronous and once $\rho' > \rho$, the failure detector stops making mistakes and becomes perfect [Chandra and Toueg 1996].

Similar ideas work when implementing $\diamond\mathcal{P}$ in other forms of partial synchrony [Hermant and Widder 2005; Fetzer et al. 2005]. For the important class $\diamond\mathcal{S}$ there even exist implementations that are optimized with respect to efficiently solving consensus [Larrea et al. 2000; Larrea et al. 2000; Chen et al. 2000; Sergent et al.

1999]. For the special case of Ω other variants of partial synchrony have been proposed [Aguilera et al. 2003; 2001] in which not all communication channels have to be eventually synchronous.

2.2.9 *Solving Problems Other than Consensus using Failure Detectors.* Other problems than consensus have been studied adapting the failure detection approach. Sabel and Marzullo [1995] consider the *election* problem (see also Larrea et al. [2000]) while Matsui et al. [2000] investigate *eventual leader election of k processes* (which they call *k-consensus*). Issues of group communication have also been considered (e.g., *atomic multicast* [Guerraoui and Schiper 1997] and *generic broadcast* [Pedone and Schiper 1999; Aguilera et al. 2000]). Predicate detection in faulty environments is investigated by Garg and Mitchell [1998a], Gärtner and Kloppenburg [2000], and Gärtner and Pleisch [2001]. Termination detection was investigated by Mittal et al. [2005]. Mostéfaoui et al. [2006] explored the ways failure detectors can be used for solving set agreement [Chaudhuri 1990], a generalized form of consensus, and renaming [Attiya et al. 1990].

2.2.10 *Using and Combining Different Failure Detector Abstractions.* The use of failure detectors can relegate much of the intrinsic knowledge of the network into lower layers and leave the application only with those issues that it needs to care about: reasoning about failures. System engineers only need to agree on the interface of the particular failure detector in question, and two groups can independently go about designing solutions: one group can start building an application given a particular failure detector semantics, the other group can choose a network architecture and a failure detection algorithm such that the failure detector semantics is satisfied.

Implementing failure detection as a service has another advantage since one implementation of, say, an eventually perfect failure detector can be used by multiple applications simultaneously. Dissemination of failure detection messages and keeping track of timeouts can be done centrally at a "middleware" layer that is usually much more efficient than having every application do this on its own. Moreover, if timeouts are tweaked or adapted, this may be done centrally in the service layer instead of adapting all different algorithms independently.

Failure detectors can also be used as sources of activation in event-driven applications. For example, Aguilera et al. [1999] investigate *quiescent algorithms*, i.e., algorithms that eventually stop sending messages. They show that failure detection has no quiescent solutions, but special failure detectors can be used as a service to build quiescent algorithms (like *quiescent reliable broadcast* [Aguilera et al. 2000b]) and terminating ones (like consensus) at higher layers.

Finally, we argue here that failure detectors also remedy the problems of asymmetric or differing timeouts within an application. While failure detectors do not offer timing information *per se*, different instantiations can separate the concerns of differing timeouts within an application. In the 3PC algorithm discussed earlier, it was noted that during the first phase of the algorithm it made sense to have a more conservative (i.e., longer) timeout to increase the chances that all processes vote *yes*. During the remainder of the algorithm, a more aggressive timeout can be used because false suspicions merely delay the outcome of the algorithm. These two

```
NON-BLOCKING_ATOMIC_COMMIT(v): { v is yes or no }
1  send v to all
2  wait until [(for each process q in Π, received q's vote) or FS_p = red]
3  if the votes of all processes are received and are yes then
4     mydecision ← CONSENSUSPROPOSE(commit)
5  else { some vote was no or there was a failure }
6     mydecision ← CONSENSUSPROPOSE(abort)
7  return mydecision
```

Fig. 3. Implementing non-blocking atomic commit using $\mathcal{FS}$ and a consensus abstraction [Guerraoui 2002].

different concerns can be captured using two different types of failure detectors. In the first phase, it is not important *which* process failed, so the failure signal failure detector $\mathcal{FS}$ is sufficient ($\mathcal{FS}$ initially outputs **green** which eventually turns into **red** if and only if *some* process has failed). In the second phase (including the election within the termination protocol), it is important to be able to suspect particular processes (especially the coordinator process), so a failure detector $\diamond\mathcal{P}$ or $\diamond\mathcal{S}$ can be used given a majority of correct processes. In fact, the latter failure detector can be encapsulated within a solution for consensus and non-blocking atomic commit can be formulated in a surprisingly simple algorithm with only half a dozen lines (see Fig. 3). Hence, failure detectors even offer fine-grained abstractions where necessary.

2.3 Limitations of Failure Detectors

Apart from its many virtues in the design and analysis of algorithms, the failure detector abstraction also has some inherent limitations. Some of these limitations have been frequent sources of misunderstandings and misconceptions in fault-tolerant algorithms. We have grouped the discussion about the limitations around four basic questions that we discuss and put into context.

2.3.1 *What is not a Failure Detector?.* The original work on failure detectors [Chandra and Toueg 1996] defines a failure detector to be a mapping from a *failure pattern* F to some output range H. The failure pattern F specifies which processes fail at what time. So anything that can be defined as a function of failures can be formally called a failure detector. Not everything that looks like a failure detector can however be defined as a function of failures.

Considering the crash failure model, Charron-Bost et al. [2000] observed that there exist (even time-free) problems that cannot be solved in asynchronous systems assuming even a perfect failure detector. For example, determining how many events a process executed before it crashed cannot be determined using a perfect failure detector. This is counterintuitive since in a fully synchronous system this can easily be detected if the failed process broadcasts a message with every event and an observer waits until a correctly calculated timeout has passed. In this sense, a failure detector does not really encapsulate all the synchrony of a system.

Gärtner and Pleisch [2002] explored an extension of the original failure detector model and managed to specify a device similar to a failure detector that allows

to fully emulate a synchronous system. This device works like a perfect failure detector, only that — upon suspecting a process — the device returns a dump of the state in which that process crashed. The precise formulation of this device is not a function only of failures anymore, but rather a function of the process state and the failures in the system. Gärtner and Pleisch [2002] proved that such a device allows to embed crash events perfectly into the causal history of a computation. So any problem can be computed that depends on the causal structure of a computation. They also showed that the same can be achieved with a perfect failure detector if the communication channels are synchronous (i.e. having a bound δ but not having a bound Δ, see Section 2.1.5). In this sense, a perfect failure detector can be regarded as an abstraction of process synchrony, not of channel synchrony.

The ability of systems to detect failures can be formalized in a number of ways. In this survey, we focus primarily on Chandra-Toueg failure detectors [Chandra and Toueg 1996; Chandra et al. 1996], but we would like to mention Gafni's round-by round failure detectors [Gafni 1998], and the "heard-of" model [Charron-Bost and Schiper 2006] as interesting examples of alternative definitions. Finally, failure detectors were recently considered in systems without fixed process identifiers [Afek and Nir 2008], that allowed for posing the question of the weakest failure detector for renaming [Attiya et al. 1990].

2.3.2 *Do Failure Detectors make sense outside of the crash model?.* The "classic" failure detectors have been *crash* failure detectors, i.e., they were tailored to the crash failure model. There are however many other failure models in the literature. Except few exceptions we discuss below, there has been very little work on extending the failure detector abstraction to these models.

Among models that refer to the incorrect behavior of processes are *fail-stop* [Schlichting and Schneider 1983], *crash-recovery* [Oliveira et al. 1997; Aguilera et al. 1998; Hurfin et al. 1998], *send/receive omission* [Hadzilacos 1984; Hadzilacos and Toueg 1994] and *Byzantine* [Lamport et al. 1982]. The fail-stop failure model is just like the crash failure model, except that the crash of a process is easily detectable by other processes. In the *crash-recovery* model, processes can crash and later resume their execution from a predefined point in their program. In the send/receive omission failure model a process sends or receives only a subset of messages it was supposed to send or receive. Finally, the Byzantine failure model allows arbitrary behavior of a faulty process.

In general, the type of failure detector that is necessary to solve a problem depends on the problem itself (e.g., consensus) and the failure model assumed in the network (e.g., crash). Usually, the failure model indicates *what* information the failure detector offers and the problem dictates *how* the failure detector should present this information (i.e., the failure detector properties). For example, the crash-recovery failure model offers a new type of behavior so adequate failure detectors should be able to convey information about this behavior. Consequently, the failure detectors used in the context of solving consensus in the crash-recovery model [Aguilera et al. 2000a] output a vector of unbounded counters hinting on how often a process has been suspected.

Considering the consensus problem, failure detector specifications have been extended to environments where the network may partition [Guerraoui and Schiper

1996; Aguilera et al. 1999] and processes may experience send and receive omissions [Dolev et al. 1997; Delporte-Gallet et al. 2005]. Lossy links are a usual assumption in work on consensus in the *crash-recovery* failure model [Oliveira et al. 1997; Aguilera et al. 1998; Hurfin et al. 1998]. In this model, questions of to what extent stable storage is necessary are also important. Lo and Hadzilacos [Lo and Hadzilacos 1994] study failure detection and consensus in a shared memory setting. The model of finite transient failures that is characteristic to the area of self-stabilization [Dijkstra 1974; Dolev 2000] has also been studied in the context of failure detectors [Beauquier and Kekkonen-Moneta 1997; Hutle and Widder 2005].

Adapting the failure detector abstraction to the general case of Byzantine failures is not straightforward. First of all it is not possible to derive a clean failure detector interface that is orthogonal to the specification of the algorithm using it. This is because the notion of a failure is not only related to timing/synchrony but also to application level messages.

"Muteness" failure detectors [Malkhi and Reiter 1997; Doudou et al. 1999; Doudou et al. 2002; 2005] extend crash failure detectors to the case when processes might stop sending messages associated with a particular algorithm (they might still keep sending other messages).

Kihlstrom et al. [2003] extended this approach to more general classes of failure models. This work distinguishes between detectable and non-detectable Byzantine failures. Non-detectable failures are either unobservable (e.g., a Byzantine process spontaneously changes his input value) or undiagnosable (they cannot be tagged to a specific process, e.g., a process claims that some other process sent him something). Byzantine failure detectors can only report detectable faults, that can be further classified into commission and omission faults, the former being in the value domain and the latter in the time domain. The omission fault detectors of Kihlstrom et al. [2003] correspond to the muteness failure detectors of Doudou et al. [1999].

A generic framework for detecting observable Byzantine behavior was proposed and validated in practice by Haeberlen et al. [2007]. An abstract Byzantine failure detector is parameterized with a system specification and ensures that every occurrence of a detectable failure, i.e., one that directly or causally affects correct processes, is eventually exposed. In an authenticated system, where no faulty process can impersonate or forge messages from a correct process, the detection mechanism is able to produce irrefutable evidences of observable deviant behavior.

2.3.3 *Can Randomization be used to implement Failure Detectors?.* In 1983, Ben-Or [1983] presented an algorithm that solves consensus in the completely asynchronous model (i.e., without failure detectors) by using *randomization*. In the algorithm, processes repeatedly flip coins to reach a majority of proposed values, upon which termination is reached. In doing this and because of the properties of the random coin, it can be shown that the probability of non-terminating runs diminishes to zero and, hence, termination can be achieved with probability one. So since both randomization and failure detection can be separately used to solve consensus it is legitimate to ask: Can failure detectors be implemented using randomization?

Aguilera and Toueg [1998] presented an algorithm that uses randomization and

unreliable failure detection to solve consensus. But their approach does not use randomization to implement failure detectors, rather randomization is used to guarantee termination in case the failure detectors never become reliable.

Völzer [2004] investigated the relationship between fairness and randomization. Fairness is a particular form of liveness assumption in distributed systems and is sometimes known under the name of *strong fairness* in the context of transition systems. Briefly spoken, fairness in the context of message passing systems means that every message sent to some process x is eventually received even though some other process sends infinitely many messages to x. Völzer showed that randomization and fairness are incomparable with respect to their expressive power by exhibiting a problem that can be solved only by fairness and not with randomization. In contrast, consensus can be solved only by randomization and without fairness [Ben-Or 1983]. Intuitively, failure detectors (as well as partial synchrony) fall more into the category of fairness assumptions than randomization (see also the work on *hyperfairness* [Völzer 2005]). So while this does not answer the question of the relationship between failure detectors and randomization, the results are a strong indication that the two concepts are in fact incomparable.

2.3.4 *Can Failure Detectors be used to Reason about Real-Time?.* Failure detectors offer an asynchronous interface for timing information. It is nevertheless sometimes helpful to figure out what kind of synchrony information can be provided by failure detectors.

Strong failure detectors ensure strong completeness and weak accuracy. This means that every crash is eventually detected but processes can make mistakes about other processes *except a single same one*. Making mistakes means to "time out too soon". Thus, implementing a strong failure detector makes it necessary to have communication and processing speed bounds regarding one "central" process. Note that this feature is asymmetric: The bounds must hold for communication coming from the central process, not for communication running towards it.

Eventually Strong failure detectors have strong completeness and eventually weak accuracy. The situation here is that processes can now make mistakes about *all* processes, but must eventually stop making mistakes regarding a single same process. Systems that offer an eventually strong failure detector must ensure that *eventually* communication and processing speed bounds hold regarding one "central" process but only in direction from this process to the other processes. This was formalized into the concept of an (eventual) *source* [Aguilera et al. 2001; 2003], a process whose outgoing channels are (eventually) timely.

As discussed above, perfect failure detectors allow building a system that is very close (but not equivalent) to a fully synchronous one [Charron-Bost et al. 2000]. It has still been argued that the use of failure detectors also offers the potential of building real-time applications by using the approach of *late binding* [Hermant and Le Lann 2002]. In this approach, a real-time problem is turned into a "time-free" problem, e.g., by basing timeliness requirements on certain activation conditions using time-free extensions to the asynchronous model like failure detectors. In this context, an asynchronous solution can be devised. Then the solution is bound to an as weakly synchronous system model as possible (e.g., one of partial synchrony) and the real-time instants of the activation conditions are computed from the guar-

antees of the underlying model. Since the application satisfies its safety and liveness properties even if the underlying network transiently violates its timeliness guarantees, this approach allows to build real-time applications with higher assumption coverage than if real-time were considered from the beginning of the design process.

Along this line of research, the issue of *fast failure detectors* has been investigated, i.e., failure detectors that detect failures in a time that is orders of magnitude less than a round trip delay [Aguilera et al. 2002].

2.4 Summary

In this section we have argued that failure detectors are useful abstractions from an engineering point of view. Firstly, they can be used to hide timeout details behind a clean operational interface that makes it easier to design, build and analyze fault-tolerant distributed systems. Secondly, implementation of the failure detection functionality can be done in a centralized, re-useable fashion that enables solutions that are more efficient compared to situations in which every application performs failure detection independently. Finally, failure detectors offer the possibility to express timing assumptions in a fine-grained manner that is more suitable to be used directly by application logic than explicit timing information.

3. FAILURE DETECTORS AS A COMPUTABILITY BENCHMARK

As we discussed in the previous section, certain failure detectors are weaker than others. For instance, the guarantees provided by an eventually perfect failure detector are weaker than those provided by a perfect failure detector. This notion can be captured precisely, and this induces a hierarchization of distributed computing problems, based on the *weakest* failure detectors needed to solve them. In a precise sense, the weakest failure detector for a given problem M captures the exact amount of information about failures needed to solve M.

In this section, we discuss the question of determining the weakest failure detector on the example of the celebrated "CHT proof" determining the weakest failure detector for consensus [Chandra et al. 1996]. The technique proposed in the CHT proof is interesting in its own right and variants of it were used in multiple succeeding weakest failure detector results. Therefore we provide here a high level overview of the proof intended to capture the very essence of it.

Then we briefly overview the techniques of deriving the weakest failure detectors for implementing read-write shared memory in a message-passing system, and solving non-blocking atomic commitment (NBAC). We show that failure detectors allows for capturing that consensus and NBAC are, in a strict sense, incomparable: the weakest failure detectors for the two problems cannot be reduced to each other.

We then provide a survey of more recent results addressing the question of the weakest failure detector for *set agreement*, a generalization of consensus. We conclude the section by discussing a conjectured hierarchy of distributed tasks that can be build using the weakest failure detector abstraction.

3.1 The CHT Play

This section gives a high level and informal account of the necessary part of the proof that Ω is the weakest failure detector to implement consensus in a message-passing system with a majority of correct processes. The proof originally appeared

in a widely cited but rarely understood paper by Chandra et al. [1996]. We describe it here as a play in five acts, preceded by a prologue and followed by an epilogue.

The Prologue. Consensus and leader election are two fundamental abstractions in distributed computing. Consensus provides processes with the ability to agree on a common value. We consider here a variant of leader election, the *eventual leader* abstraction, denoted by Ω, through which the processes eventually agree on a common correct leader. Proving that Ω is the weakest failure detector to implement consensus with a majority of correct processes goes through exhibiting two algorithms:

(1) an algorithm that implements consensus with a majority of correct processes using Ω (the sufficiency part);
(2) an algorithm T (called a *transformation* or a *reduction* algorithm) that, using any algorithm A that implements consensus with some failure detector $\mathcal{D}$, implements Ω (the necessity part).

The sufficiency part, i.e., an eventual leader-based consensus algorithm, first appeared in the original Paxos protocol [Lamport 1998], and then evolved to multiple different algorithms (see, e.g., Oki and Liskov [1988], Dwork et al. [1988], and Chandra and Toueg [1996]).

The necessity part, often called the "CHT proof" (or simply "CHT "), for Chandra, Hadzilacos and Toueg, pioneers a new reduction style in the theory of distributed computing [Chandra and Toueg 1996]. It is however quite long and rather involved. That is maybe why it is rarely understood.

Below, we have tried to give a high level description of CHT. Clearly, we often sacrificed rigour for intuition: sometimes on purpose and sometimes not. We give a description of the main components of the proof and we consider a simple case of the proof while abstracting away a trickier one in the appendix. Even our very informal description requires however five acts. Moreover, the use of the first of these acts is getting visible only at the very end of the proof. We thus encourage the reader to continue reading even when it seems as though the reasoning does not make progress anymore. Catching a glimpse of the elegance and cleverness of the CHT proof will be your reward at the end.

Before delving into CHT, we first recall here some of its underlying elements. As we recalled above, CHT is about constructing algorithm T using consensus algorithm A (itself using failure detector $\mathcal{D}$). We say few words here about what is, a priori, needed to be noticed about algorithms A and T.

—About algorithm A (which is given):

(1) *A uses a failure detector $\mathcal{D}$ in a message passing system.* A failure detector is a distributed abstraction that provides processes with information about the status (crashed or not) of other processes in the system. Every process p_i has a module of that failure detector and p_i can query that module at any time. The information obtained from this query is used as an input by p_i to its algorithm automaton, as we discuss below. Algorithm A is a set of deterministic distributed automata, one per process. At every step of any run of A, exactly one process p_i triggers its algorithm automaton: we say that p_i executes a step. A correct process is one that executes an

infinite number of steps. (A process that crashes simply stops triggering its automaton.) The input of the automaton is (1) p_i's state, (2) a message that p_i has received from some other process, as well as (3) the value obtained by p_i from its local failure detector module. The output of the automaton is (4) a new state as well as (5) a message that p_i broadcasts. Communication is assumed to be reliable in the sense that if p_i sends a message, then this message is eventually received by every correct process.

(2) *A implements consensus.* The problem consists for the processes to propose values and to decide on the same final value. More precisely, every correct process will decide on a value (*termination* property of consensus); no two correct processes will ever decide differently (*agreement*); and any value decided is a value proposed (*validity*).

—About algorithm T (which is to be constructed):

(1) *T needs to implement Ω.* In other words, T is a leader election algorithm that should ensure that, eventually, all correct processes permanently elect the same correct process. The existence of both (1) algorithm T and (2) an algorithm that implements consensus using Ω is what derives the fact that Ω is the weakest failure detector to implement consensus.

(2) *T is not restricted to using A as a black-box.* In other words, we are not trying here to implement Ω out of a consensus abstraction. At first glance, using consensus as a black-box, we could indeed easily implement a leader election scheme that ensures that the processes do all elect the same leader: the processes would each propose its identity and, using consensus, we would get the same identity as a decision. This is a strictly weaker variant of Ω because there is no guarantee that the leader is correct. On the other hand, if the goal was simply for every correct process to elect a correct process, each would simply output itself. The challenge that T needs to face is to have the processes eventually and permanently *agree* on the very same *correct* leader. Algorithm T achieves this by using A's automata at *every* process, as we overview in the following.

We give here an overview of algorithm T. The basic idea underlying T is to have each process locally *simulate* the overall distributed system in which the process executes several runs of A. The processes then use the outputs provided by these runs to extract the leader process.

Every process simulates, locally, runs of algorithm A by launching threads that mimic the behavior of every other process in the system running algorithm A (Fig. 4). But how can one process p_i simulate the overall system executing several runs of A? Basically, every process p_i feeds algorithm A with a set of proposed values, one for each process of the system, i.e., p_i pretends to be every other process and proposes values to A's runs. In fact, process p_i proposes all possible combinations of input values as we discuss in Act 2. Then all automata composing A are triggered locally by p_i, that locally emulates, for every simulated run of A, the states of all processes as well as the emulated buffer of exchanged messages.

Crucial elements that are needed for the simulation are (1) the values from failure detectors that would be output by $\mathcal{D}$ as well as (2) the order according to which the processes are taking steps. For these elements, that we call the *stimuli* of algorithm

A, process p_i *exchanges* information with the other processes, as we will discuss in Act 1. It is important to notice that the output of $\mathcal{D}$ is not restricted in any way. In fact, this output can be *any value* that encodes some information about failures.

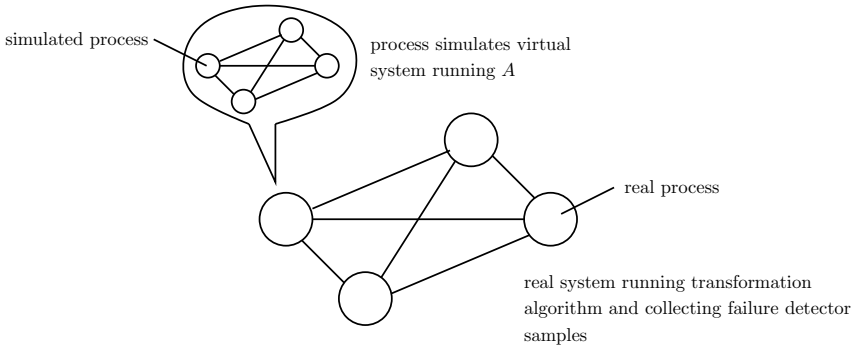

Fig. 4. Processes simulate runs of consensus in their own simulation environment taking stimuli from "reality".

Every process p_i locally uses the series of (consensus) decisions obtained by A in order to *extract* the leader process (Acts 3-5). The idea is to analyze which combination of values proposed to consensus leads to which decision values. In short, the process that is elected leader (i.e., extracted) is the one, the proposed value (or a particular step) of which makes a crucial difference in the pattern of decision values.

Act 1: The Exchange. The processes periodically query their failure detector modules of $\mathcal{D}$ and send the output to all other processes. As a result, every process knows more and more of the other processes' failure detector outputs and temporal relations between them.

All this information is pieced together in a single data structure, a directed acyclic graph (DAG). This DAG has some special properties that follow from its construction as we will discuss later. What is important to see for now is that the DAG can be used to derive simulation stimuli to A: it contains activation schedules and failure detector outputs for the processes to execute steps of A's instances. In fact, every path through this DAG is one such possible stimulus that can be used as an input to A in the simulation environment.

An example is depicted in Figure 5 in which the vertex $[p_1, d_4]$ (meaning that the failure detector at p_1 responded with d_4) is added to the DAG after having received $[p_2, d_2]$ and $[p_3, d_3]$. The DAG is transitively closed. Thus, every suffix of a path through the DAG is again a path through the DAG.

Of course, these schedules will always be finite, so they can be used to simulate only finite runs of A in the simulation environment (this means that A may not terminate for some processes). But this is not too bad. As long as the processes continue to exchange information, the DAG becomes larger and larger and provides more and more simulated terminating runs of A.

When the processes communicate, they send their own version of the DAG to each other. When receiving such a DAG, a process integrates it into its own DAG (it forms the union). In this way, the intersection of all correct processes' DAGs grows without bound too: without knowing exactly how large it is, the processes construct an ever-increasing "common sub-DAG".

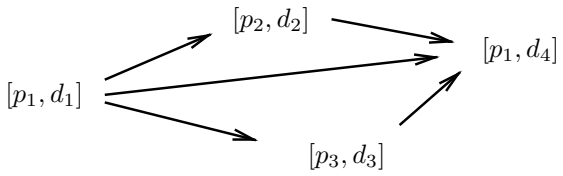

Fig. 5. Construction of the DAG.

Act 2: The Simulation. Let us forget the point about the common sub-DAG and the special DAG properties for a moment and just recall that the DAG can be used to extract stimuli for running A in a simulation environment. So assume that some process p_i has just constructed a new version G of its personal DAG. There are many paths through G and each path corresponds to an activation schedule for A. Process p_i simply starts a simulation environment *for every path* through G. Note that this might be a Herculean task in reality, but we are interested here in proving the existence of an algorithm, not necessarily an efficient one. Since G is finite, the number of paths through G is also finite and so we are not demanding the impossible.

So consider one such path π through G. We will use π to stimulate A in the simulation environment. But A is an algorithm that solves consensus, and so we need to provide input values for A. Which ones should we choose? The answer is simple: do not choose; rather run even more instances of A! So for the path π we run not one simulation environment, but rather $n+1$ simulation environments at the same time. (Remember that n is the number of processes in the system: $p_1, \ldots, p_n$.) The initial values given to each of these instances follow a certain pattern: in the first instance, all processes propose 0; in the second instance, process p_1 proposes 1 and all others propose 0; in the third instance, processes p_1 and p_2 propose 1, and all others propose 0, etc. In the final $(n+1)$-th instance, all processes propose 1. The input vectors to the instances are denoted $I_0, I_1, \ldots, I_n$, where I_i makes processes p_1 up to p_i propose 1 and the rest (processes p_{i+1} up to p_n) propose 0.

To recall: whenever its DAG changes, every process p_i runs a simulation environment for *every path through the DAG* and *for every input vector* $I_0, \ldots, I_n$. Each simulation is run "up to the end", i.e., until it runs out of activation stimuli from π found in the current DAG. When all simulations are finished, then p_i reconsiders incoming messages, constructs a new DAG and starts the simulations all over again from scratch. As mentioned above, this is a lot of work, but it is not impossible.

Act 3: The Tagging. At the end of the simulations (i.e., before constructing a new DAG and re-starting the simulations again), process p_i looks at the results of the runs of A. In some simulation environments, A might not have terminated since the

length of the input stimulus is not sufficiently large (the emulated processes have not been activated often enough). But in others, A might have terminated. In fact, since correct processes keep on exchanging messages with each other, there is eventually a path through the DAG in which such processes will appear regularly. So it is just a matter of waiting long enough, and launching new simulations, until there is some simulation stimulus that leads A to termination. Since A is a consensus algorithm, it gives us the decision value for the particular run of the simulation (either 0 or 1).

Process p_i now looks at all simulations that started from the same initial input vector. For example, it considers all simulations that started from I_0 and looks at the decision values of those simulations that have terminated. Of course, the decision value will be 0 for input vector I_0 since all processes proposed 0 and the decided value must be a proposed value (this is mandated by the validity property of consensus). Similar for I_n, only 1 can be decided. But for other input vectors like I_1 and I_{n-1} both 0 and 1 can be the result.

For every input vector $I_0, \ldots, I_n$, p_i determines the set of all decided values and "tags" the input vector with them. An input vector that is only tagged with 0 is called 0-*valent*. An input vector tagged only with 1 is called 1-*valent*. Otherwise, the input vector is called *bivalent*. (We disregard the case where none of the simulated runs decides, i.e., where there are no tags at all attached to an input vector; this happens only finitely many times.)

Now look at the list of input vectors $I_0, \ldots, I_n$: certainly, and as we pointed out, I_0 is 0-valent, but what about the others? Let us assume that I_1 and I_2 are 0-valent too, but not all input vectors can be 0-valent because I_n is 1-valent. This means that there exists some index i in the sequence of input vectors where I_{i-1} is 0-valent and I_i is either 1-valent or bivalent. Index i is called a *critical index*. Interestingly, the critical index can be used to make an educated guess about some correct process in the system.

Act 4: The Stabilization. An important fact to observe now is that there is a time after which the critical index does not change anymore and this index is the same at all processes. Since I_0 will always remain 0-valent, the critical index cannot decrease below 1; it will eventually stabilize to a value that is at least 1. Furthermore, all correct processes doing the simulation will stabilize to the same critical index. This is because the DAG is the only input to the simulations, and so the DAG alone inevitably determines the critical index. But as noted earlier, correct processes exchange and update their views of the DAG periodically, which implies a common ever-growing sub-DAG of all correct processes. Eventually, the common sub-DAG will be sufficiently large to allow A to terminate on the stimuli extracted from it. Hence, all correct processes will "stabilize" on the tags they attach to the input vectors: for example, if some process tags I_3 with 0, then eventually every other correct process will do the same (this tag is sticky, i.e., it will not vanish). As a result, once an input vector has become bivalent, it cannot switch back to 1-valent or 0-valent anymore. So the critical index cannot shift to the right, it can only shift to the left (toward I_0). Moreover, the critical index cannot shift left forever, it must stop at some point. And this point is eventually computed by every correct process. So while there is no way for the processes to "know" that their critical index has stabilized (unless it is 1 of course), stabilization will eventually happen.

Act 5: The Extraction. The final step is now to see what a critical index has to do with the identity of a correct process. This is maybe the point that is most difficult to understand, so we start by showing that a critical index has *some* relation to a correct process. If you can see that in some cases the critical index gives useful hints about a correct process, it might be easier for you to believe that it may do the same in the other (more complicated) cases.

The *simple* case we consider is, besides the assumption that the stabilization to a critical index i has taken place, that I_{i-1} is 0-valent, and that I_i is 1-valent. The claim now is that is that p_i is a correct process.

Assume by contradiction that p_i is faulty. In this case, p_i stops participating in the collective building of the DAG. So at some point, the DAG is being extended without adding vertexes with respect to p_i anymore. Now the special property of transitive closure of the DAG is important: consider any path π through the DAG in which p_i has stopped to participate. Since at some point p_i has stopped contributing to the DAG, there is a suffix π' of π in which there is no vertex regarding p_i. Because of transitive closure, π' is also a path through the DAG.

Because π' is also a path through the DAG, there must be a simulated run of A performed with π' as input stimulus for the input vector I_i. Since we assume that the critical index has stabilized, I_i remains 1-valent, and so running A on any extension of π' results in deciding value 1. To summarize, A has a run c_1 in which processes start from I_i, p_i takes no steps at all and the decision value is 1.

Now look at the same stimulus applied to A starting from I_{i-1}. Since I_{i-1} is 0-valent, the outcome of A must be 0. To summarize, A has a run c_2 in which processes start from I_{i-1}, p_i takes no step at all and the decision value is 0. But from the point of view of all the processes apart from p_i, the executions c_1 and c_2 are indistinguishable: the only difference is p_i's initial value that changes in both cases, but no process knows that value as p_i takes no step. Algorithm A is deterministic, so it should yield the same decision value in both cases, which it does not. This is the contradiction. So p_i must be a correct process.

The above line of reasoning shows that there is some non-trivial information about correct processes in the critical index. We have argued only for the case when I_i is 1-valent. The case when I_i is bivalent is a little trickier and involves reasoning about devices called *hooks* and *forks* (this is probably the place in the proof where readers might surrender). We will do some hand waving here and just ask the reader to believe that this part of the proof works. For those who still want to understand this final part of the proof, The appendix A provides a multi-page, high-level overview over this part of the proof.

The Epilogue. So let us put all CHT pieces together: we are given a distributed system, a failure detector $\mathcal{D}$, and an algorithm A that uses $\mathcal{D}$ to solve consensus. We design an algorithm T where the processes:

—periodically query $\mathcal{D}$;

—exchange the results in the form of DAGs;

—use the DAGs to locally simulate a large number of runs of A with all possible input vectors of proposal consensus values (every process runs these simulations, not necessarily the same set, but eventually every simulation that is done by one

correct process is also performed by all other correct processes);
—use the consensus decisions obtained from these runs to tag the input vectors;
—and finally use these tags to identify a critical index and elect the corresponding leader process.

To conclude, let us point out some extensions of CHT.

—CHT considers a distributed system model where the processes communicate by exchanging messages through reliable channels (e.g., in Act 1). The result has been extended by Lo and Hadzilacos [1994] and Guerraoui and Kouznetsov [2008] to a distributed system model where the processes communicate through registers. The result was also revisited in models with more sophisticated shared objects than registers, first by Neiger [1995], and later by Guerraoui and Kouznetsov [2008]. With these objects, the weakest failure detector to implement consensus is strictly weaker than Ω.
—The variant of consensus considered in CHT is sometimes called *strong* consensus: no process can decide 0 (resp. 1) if they all propose 1 (resp. 0) (this is fundamental in Act 3). This contrasts a weaker variant of consensus where we would require only that algorithm A has a run where 1 is decided and a run where 0 is decided. Considering such a weak variant impacts the result as pointed out by Chandra et al. [1996] and Guerraoui and Kuznetsov [2008].
—Finally, it is important to recall that CHT is *only* the *necessary* part of the proof that Ω is the weakest failure detector for consensus. The *sufficient* part goes through exhibiting an algorithm that implements consensus using Ω. Starting from the original Paxos algorithm by Lamport [1998], multiple variants of such algorithms have been proposed in the literature [Oki and Liskov 1988; Dwork et al. 1988; Chandra and Toueg 1996]. All these algorithms assume however a majority of correct processes, which is indeed necessary if Ω is the only failure detector available.
Determining the weakest failure detector for consensus without the correct majority assumption has been studied by Delporte-Gallet et al. [2003], and we discuss it in more detail in Section 3.2.4.

3.2 The weakest failure detector for a register

In this section, we sketch the proof that the quorum failure detector, denoted by Σ, is the weakest to implement atomic registers, regardless of when and where failures occur. Σ outputs a set of processes at each process, called *quorum*, and guarantees that every quorums (output at any times and by any processes) intersect, and eventually every output quorum consists of only correct processes.

The result was first obtained by Delporte-Gallet et al. [2003]. An alternative "CHT-like" proof, based on exchanging failure detector samples and using the samples as stimuli for locally simulated runs, was later presented by Eisler et al. [2004]. We review here the proof of Eisler et al. [2004], because it employs the simulation technique discussed in the previous section.

3.2.1 *Read/write shared memory.* A *register* is a shared object accessed through two operations: *read* and *write*. The write operation takes as an input parameter a specific value to be stored in the register and returns a simple indication *ok* that

the operation has been executed. The read operation takes no parameters and returns a value according to one of the following consistency criteria. A (single-writer, multi-reader) *safe* register ensures only that any read operation that does not overlap with any other operation returns the argument of the last write operation. A (stronger) *regular* register ensures that any read operation returns either a concurrently written value, or the value written by the last write operation. The (strongest) *atomic* register ensures that any operation appears to be executed instantaneously between its invocation and reply time events. (Precise definitions are given by Herlihy and Wing [1990] and Attiya and Welch [2004].)

The registers we consider are *fault-tolerant*: they ensure that, despite concurrent invocations and possible crashes of the processes, every correct process that invokes an operation eventually gets a reply (a value for the read and an *ok* indication for the write).

The classical results [Vitányi and Awerbuch 1986; Israeli and Li 1993] imply that if a failure detector $\mathcal{D}$ is sufficient to implement a safe one-writer one-reader register for any two processes, then $\mathcal{D}$ is sufficient to implement an atomic multi-writer multi-reader register. Thus, we do not need to specify here whether the register implemented using $\mathcal{D}$ is safe, regular or atomic, and how many readers and writers it can maintain: all these registers are computationally equivalent.

3.2.2 *The sufficiency part.* By a simple variation of the algorithm of Attiya et al. [1995] for implementing registers in a message-passing system with a majority of correct processes, we obtain an algorithm that implements registers using Σ, regardless on the number and location of failures. Where the original algorithm uses waiting until a majority responds to ensure that a read operation returns the most recently written value, we can use the quorums provided by Σ to the same effect.

3.2.3 *The reduction algorithm.* Now we need to show that any failure detector that can be used to implement registers can be transformed into Σ.

Let $\mathcal{D}$ be any failure detector that can be used to implement a set of atomic registers $\{X_p\}_{p\in\Pi}$, where for every $p \in \Pi$, X_p can be written by p and read by all processes. We present an algorithm that, using $\mathcal{D}$, implements Σ.

To extract Σ, we assign a particular protocol, i.e., a sequence of operations on the implemented registers, to every process. In this protocol, denoted $\mathcal{A}$, every process p first writes 1 in X_p, and then reads the registers $\{X_q\}_{q\in\Pi}$ (we assume that each X_q is initialized to 0). A run in which p is the only process that executes $\mathcal{A}$, is called a *p-solo run* of $\mathcal{A}$. A p-solo run in which p completes $\mathcal{A}$, is called a *complete p-solo run* of $\mathcal{A}$.

It is important to notice that in any run R of $\mathcal{A}$ in which two processes p and q both complete executing $\mathcal{A}$, either p reads 1 in X_q, or q reads 1 in X_p. Intuitively, this implies that the sets of processes "involved" in the executions of $\mathcal{A}$ at p and q intersect, which gives us a hint of how to extract Σ from $\mathcal{A}$ and $\mathcal{D}$.

As in CHT, the reduction algorithm consists of two tasks: the *communication* task and the *computation* task.

The communication task, in which each process p samples its local module of $\mathcal{D}$, exchanges the failure detector samples with the other processes, and assembles

```
Initially:
  Σ-output_p ← Π { Σ-output_p is the output of p's module of Σ }
while true do
  wait until p adds a new failure detector sample u to its DAG G_p
  repeat
    let G_p(u) be the subgraph induced by the descendants of u in G_p
    S ← set of simulated runs of A induced by G_p(u)
  until there is a complete p-solo run R ∈ S
  Σ-output_p ← set of all processes that take steps in the run R
```

Fig. 6. Extracting Σ: code for each process p

these samples in an ever-increasing directed acyclic graph G_p, is organized exactly as in Act 1 of Section 3.1. The computation task, in which p simulates runs of $\mathcal{A}$ and uses these runs to extract its current quorum (the output of its emulated module of Σ), is presented in Figure 6.

To compute its current quorum, process p first waits until enough "fresh" (not previously appeared) failure detector samples are collected in G_p. Eventually, G_p includes a sufficiently long fresh path g that can be used as a stimuli of a complete p-solo run R (we say that g *induces* R). The set of processes that take steps in R constitute the current quorum of p stored in variable Σ-$output_p$.

The correctness of the reduction algorithm follows immediately from the following two observations:

(1) Eventually, at every correct process p, Σ-$output_p$ contains only correct processes.
Indeed, there is a time after which faulty processes do not produce fresh failure detector samples and thus do not participate in fresh runs of $\mathcal{A}$ simulated by p.

(2) For all p and q, every two quorums computed by p and q in the algorithm of Figure 6 intersect.
Indeed, assume, by contradiction, that there exist $P, Q \subset \Pi$ such that $P \cap Q = \emptyset$, and, at some time t_1, p computes Σ-$output_p = P$ and, at some time t_2, q computes Σ-$output_q = Q$.
By the algorithm of Figure 6, $\mathcal{A}$ has a complete p-solo run R_p and a complete q-solo run R_q such that the sets of processes that participate in R_p and R_q are disjoint. But since the sets of processes taking part in R_p and R_q are independent of each other, the two runs can be composed in a single run R that is indistinguishable from R_p to p, and indistinguishable from R_q to q.
Hence, in R, both p and q complete $\mathcal{A}$ in R, i.e., p writes 1 in X_p and then reads all registers, and q writes 1 in X_q and then reads. Since R_p is a p-solo run, and p cannot distinguish R and R_p, p reads 0 from register X_q in R. Respectively, q reads 0 from register X_p in R. But this cannot happen in any register implementation: at least one of the processes p and q must read 1 in the register of the other process!
The contradiction implies that Σ-$output_p$ and Σ-$output_q$ always intersect.

Thus, Σ is indeed the weakest failure detector to implement atomic registers.

3.2.4 *Solving Consensus in All Environments.* Once we determined the weakest failure detector to implement atomic registers, it is straightforward to determine the weakest failure detector for solving (uniform) consensus, in all environments, i.e., regardless of when and where failures occur. This failure detector is (Ω, Σ), the composition of Ω and Σ.

Indeed, failure detector (Ω, Σ) can be used to solve consensus regardless of when and where failures occur, by first implementing registers out of Σ, and then consensus out of registers and Ω [Lo and Hadzilacos 1994].

On the other hand, consensus can be used to implement atomic registers [Lamport 1978; Schneider 1990], and thus to extract Σ. Combined with the fact that Ω is necessary to solve consensus [Chandra et al. 1996] (see Section 3.1), this implies that (Ω, Σ) is necessary to solve consensus.

Note that the *nonuniform* version of consensus that only requires *correct* processes to agree cannot be used to implement a register and, thus, does not allow for extracting Σ. Eisler et al. [2007] determined the weakest failure detector for solving nonuniform consensus and showed that it is strictly weaker than (Ω, Σ).

3.3 Solving Non-Blocking Atomic Commit

In this section, we discuss the weakest failure detector for solving Non-Blocking Atomic Commit (NBAC) [Delporte-Gallet et al. 2004]. This failure detector is $(\Psi, \mathcal{FS})$, the composition of Ψ, introduced by Delporte-Gallet et al. [2004], and $\mathcal{FS}$, the failure signal failure detector [Charron-Bost and Toueg 2001a; Guerraoui 2002] (see Section 2.2.10 for a definition).

3.3.1 *Failure detector Ψ.* Roughly speaking, Ψ behaves as follows: For an initial period of time the output of Ψ at each process is $\perp$. Eventually, however, Ψ behaves either like the failure detector (Ω, Σ) at all processes, or, *in case a failure previously occurred*, it *may* instead behave like the failure detector $\mathcal{FS}$ by outputting **red** at all processes. The switch from $\perp$ to (Ω, Σ) or $\mathcal{FS}$ need not occur simultaneously at all processes, but the same choice is made by all processes. Note that the switch from $\perp$ to $\mathcal{FS}$ is allowable *only* if a failure previously occurred. Furthermore, if a failure does occur processes are not *required* to switch from $\perp$ to $\mathcal{FS}$; they may still switch to (Ω, Σ).

3.3.2 *Using $(\Psi, \mathcal{FS})$ to solve NBAC.* The algorithm in Figure 7 uses $(\Psi, \mathcal{FS})$ to solve NBAC, regardless of the location and timing of failures. The algorithm is very similar to that of Fig. 3. Each process p sends its vote to all processes and then waits until the votes of all processes are received or $\mathcal{FS}$ detects a failure by outputting **red**. If the votes of all processes are received and are *yes*, then p sets the *myproposal* variable to 1. Otherwise, if some vote was *no* or a failure was detected by $\mathcal{FS}$, then p sets the *myproposal* variable to 0.

Then each process p waits until the output of Ψ becomes different from $\perp$. At that time, either Ψ starts behaving like $\mathcal{FS}$ or it starts behaving like (Ω, Σ). If Ψ starts behaving like $\mathcal{FS}$ (Ψ can do so *only* if a failure previously occurred), p returns *abort*. The remaining case is that Ψ starts behaving like (Ω, Σ). It was shown by Delporte-Gallet et al. [2003] that there is an algorithm that uses (Ω, Σ) to solve consensus (see also Sect. 3.2). Therefore, in this case, p proposes *myproposal* to that consensus algorithm and returns the value decided by that algorithm. If 1 is

```
NON-BLOCKING_ATOMIC_COMMIT(v): { v is yes or no }
1  send v to all
2  wait until [(for each process q in Π, received q's vote) or FS_p = red]
3  if the votes of all processes are received and are yes then
4     myproposal ← 1
5  else { some vote was no or there was a failure }
6     myproposal ← 0
7  wait until [Ψ_p ≠ ⊥]
8  if Ψ_p = red
9     then { henceforth Ψ behaves like FS }
10       return abort
11    else { henceforth Ψ behaves like (Ω, Σ) }
12       mydecision ← CONSENSUSPROPOSE(v) { use Ψ to run (Ω, Σ)-based consensus algorithm }
13 if mydecision = 1 then
14    return commit
15 else
16    return abort
```

Fig. 7. Using $(\Psi, \mathcal{FS})$ to solve NBAC: code for each process p

decided in the consensus algorithm, then p returns *commit*. If 0 is decided, then p returns *abort*.

The Agreement property of NBAC follows from the Agreement property of consensus and the fact that the output of Ψ switches uniformly from $\perp$ to (Ω, Σ) or $\mathcal{FS}$ at all processes. If there are no failures, then eventually p receives all the votes. If a failure occurs, then $\mathcal{FS}$ eventually outputs **red**. Hence, the wait statement in line 2 is non-blocking. The Termination property of consensus ensures that every correct process eventually decides.

Assume that p decides *commit*. By Validity of consensus some process q previously proposed 1. By the algorithm, q received the votes of all processes and all the votes were *yes*.

Assume now that p decides *abort*. Thus, either Ψ_p output **red**, i.e., a failure previously occurred, or the consensus algorithm (Ω, Σ) returned 0. By Validity of consensus some process q previously proposed 0. If some process q proposed 0, then either q received vote *no* from some process or a failure previously occurred and was detected by $\mathcal{FS}$. In both cases, Validity of NBAC is ensured.

3.3.3 *The weakest failure detector to solve NBAC.* Intuitively, $(\Psi, \mathcal{FS})$ precisely captures the semantics of NBAC. Indeed, if all processes propose 1 the only reason for an NBAC algorithm not to decide *commit* is a failure of some process. So repeatedly running the algorithm can be used for "anonymously" detecting failures, i.e., emulating $\mathcal{FS}$. Further, if processes agree on the fact that a failure previously occurred, then it is safe for them to return *abort* (the $\mathcal{FS}$ part of failure detector Ψ). Otherwise, processes must be able to reach agreement using there views of proposed values (the (Ω, Σ) part of failure detector Ψ).

Let $\mathcal{D}$ be any failure detector that solves NBAC, and let A be any algorithm that solves NBAC using $\mathcal{D}$.

There is a straightforward reduction algorithm that transforms $\mathcal{D}$ into $\mathcal{FS}$ [Charron-

Bost and Toueg 2001a; Guerraoui 2002]. Initially, at every process, the reduction algorithm outputs **green**. Processes run a series of instances of the NBAC algorithm A using $\mathcal{D}$ proposing *yes* in every instance, as long as *commit* is decided in every instance. If *abort* is decided, then the reduction algorithm switches its output to **red**. Clearly, **red** can only be output if a failure previously occurred, and if a failure occurs, eventually **red** is permanently output at every correct process.

Showing that $\mathcal{D}$ can be transformed into Ψ is based on a rather involved use of properties of NBAC and the technique of Chandra et al. [1996], and we refer to Delporte-Gallet et al. [2004] for the description of the corresponding reduction algorithm.

An immediate corollary to the result of Delporte-Gallet et al. [2004] is that consensus and NBAC are, from the failure detector perspective, incomparable (this observation was initially made by Charron-Bost and Toueg [2001b] and Guerraoui [2002]). We can easily show that, in general, (Ω, Σ) and $(\Psi, \mathcal{FS})$ cannot be reduced to each other. Indeed, in case there is failure $(\Psi, \mathcal{FS})$ can behave like $\mathcal{FS}$ and eventually output **red** at every correct process. Assuming a system of 3 or more processes, we can immediately see that Ω cannot be extracted from this information about failures, and, thus, consensus cannot be solved. On the other hand, the output of (Ω, Σ) does not allow for extracting $\mathcal{FS}$: at no point of time (Ω, Σ) can be used reliably detect that there is at least one failure.

3.4 The Set Agreement Quest and the Hierarchy of Distributed Tasks

The (n, k)-set agreement problem is a generalization of consensus in which n processes have to decide on at most k distinct proposed values (for $k = 1$, the problem is consensus). This problem is impossible if processes can only communicate using registers, k processes can crash, and no information about failures is available [Borowsky and Gafni 1993; Herlihy and Shavit 1999; Saks and Zaharoglou 2000]; the impossibility trivially then also applies also to message-passing systems. It was conjectured by Raynal and Travers [2006] that Ω_k, a generalization of Ω is the weakest failure detector for solving (n, k)-set agreement using registers, regardless of the failure pattern. Ω_k outputs a set of k processes and, eventually, the same set containing at least one correct process is permanently output at every correct process. It was shown by Raynal and Travers [2006] that Ω_k is sufficient to solve (n, k)-set agreement. In the proposed algorithm, Ω_k serves as an eventual *leader set* that eventually imposes at most k decision estimates to the rest of processes.

For the case $k = n - 1$, the conjecture was later disproved by Guerraoui et al. [2007] with a failure detector Υ. Υ outputs a non-empty set of processes, and eventually all correct processes stabilize on the same output set that is *not* the set of correct processes. Υ is strictly weaker than Ω_{n-1} but still strong enough to solve $(n, n-1)$-set agreement. In fact, Υ was shown to be the weakest *stable* failure detector to solve $(n, n-1)$-set agreement. (A failure detector is stable if its output eventually stabilizes at every correct process.) Later it was shown that there are unstable failure detectors that allow to solve $(n, n-1)$-set agreement and are weaker than Υ.

Finally, the weakest failure detector to solve $(n, n-1)$-set agreement in read-write shared memory systems was presented by Zielinski [2008]. This failure detector, denoted by anti-Ω, outputs a process identifier and guarantees that, eventually,

some correct process is never output. A generalization of anti-Ω, anti-Ω_k, outputs a set of $n-k$ processes, and eventually some correct process is never in the output sets. For the case $k = 1$, anti-Ω_k is equivalent to Ω, the weakest failure detector for consensus (or $(n, 1)$-agreement) in read-write shared memory systems [Lo and Hadzilacos 1994; Guerraoui and Kouznetsov 2008], which brings the conjecture that anti-Ω_k is also the weakest failure detector for solving (n, k)-agreement for $1 < k < n - 1$.

Interestingly, the reduction algorithms of Guerraoui et al. [2007] and Zielinski [2008] do not use the exact specification of set agreement. Indeed, the output of the desired failure detector is extracted from the very fact that a given failure detector circumvents some asynchronous impossibility. This provides an evidence for a hierarchy of n-process distributed tasks [Herlihy and Shavit 1999], based on the weakest failure detectors needed for solving them. The bottom level (level 0) in this hierarchy is populated by *trivial* tasks, tasks that can be solved asynchronously (e.g., $(2n-1)$-renaming [Attiya et al. 1990]). The top level (level $n-1$) is populated by *universal* tasks (e.g., consensus): if a failure detector solves a universal task, then it solves any task. The weakest failure detector to solve a universal task is Ω [Chandra et al. 1996; Guerraoui and Kuznetsov 2008]. Now level ℓ ($\ell = 1, \ldots, n-2$) is defined iteratively as follows. A task $\mathcal{T}$ belongs to level ℓ if and only if it does not belong to level $\ell - 1$ and any failure detector that solves a task that does not belong to level $\ell - 1$ also solves $\mathcal{T}$. Level 1 in the hierarchy is characterized by Zielinski [2008] where $(n, n-1)$-set agreement is shown to be the easiest non-trivial task and anti-Ω — the corresponding weakest failure detector. Filling the gap between levels 1 and $n - 1$ or disproving the conjectured hierarchy is an interesting open question.

3.5 Summary

Failure detectors are not only a helpful engineering abstraction but induce a hierarchization between problems in distributed computing. If the weakest failure detector to solve a problem M is strictly weaker than the weakest failure detector to solve a problem N, then M is (in terms of failure information) strictly easier to solve than M. Indirectly, this also allows to compare the synchrony requirements of problems in fault-tolerant computing. We discussed the underlying notion of a weakest failure detector and presented several examples of weakest failure detector proofs, starting from the seminal CHT proof of Chandra, Hadzilacos and Toueg. More examples can be found in the literature (see. e.g., [Aguilera. et al. 2000; Eisler et al. 2007; Delporte-Gallet et al. 2005; Zielinski 2007; Guerraoui et al. 2008; Guerraoui and Kouznetsov 2008; Delporte-Gallet et al. 2008]). Jayanti and Toueg [2008] fixed several glitches in the original formalism of Chandra et al. [1996] that allowed them to show that, strictly speaking, every distributed computing problem is matched with a corresponding weakest failure detector.

4. CONCLUDING REMARKS

Take the time-free (asynchronous) system model that is usually used when reasoning about fault-intolerant distributed algorithms, add the concept of failure detectors, and you get a system model that can be used to reason about fault-tolerant distributed algorithms. The failure detector abstraction has many virtues as an en-

gineering tool and as a computability benchmark. As we discussed in Sections 2.3 and 3.4, it also has some limitations in expressiveness and many research issues are left open.

The notion of a failure detector can be extended beyond the crash failure model (some examples of such extensions are described Section 2.3.2). However, unlike crash failure detectors discussed in this survey, a generic failure detector should be aware of the algorithm that each node in the system is supposed to be running. Thus, generic failure detectors cannot be compared independently of the algorithms that use them. As a result, it is tricky to give a meaningful definition of the weakest generic failure detector for a given problem, and it might be difficult to use failure detectors as a computability benchmark outside the crash failure model. On the other hand, failure detectors that account for more general classes of failures can still be an efficient engineering tool [Malkhi and Reiter 1997; Doudou et al. 1999; Haeberlen et al. 2007].

Acknowledgments

We wish to thank Martin Hutle for comments on a previous version of this paper and the anonymous reviewers for the constructive feedback.

REFERENCES

Afek, Y. and Nir, I. 2008. Failure detectors in loosely named systems. In *PODC*. 65–74.

Aguilera, Delporte-Gallet, Fauconnier, and Toueg. 2001. Stable leader election. In *Proceedings of the International Symposium on Distributed Computing (DISC)*. LNCS.

Aguilera, Delporte-Gallet, Fauconnier, and Toueg. 2003. On implementing omega with weak reliability and synchrony assumptions. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*.

Aguilera, M. and Toueg, S. 1998. Failure detection and randomization: A hybrid approach to solve consensus. *SIAM Journal on Computing 28*.

Aguilera, M. K., Chen, W., and Toueg, S. 1998. Failure detection and consensus in the crash-recovery model. In *Proceedings of the 12th International Symposium on Distributed Computing (DISC)*. 231–245.

Aguilera, M. K., Chen, W., and Toueg, S. 1999. Using the heartbeat failure detector for quiescent reliable communication and consensus in partitionable networks. *Theoretical Computer Science 220,* 1 (June), 3–30.

Aguilera, M. K., Chen, W., and Toueg, S. 2000a. Failure detection and consensus in the crash recovery model. *Distributed Computing 13,* 2 (Apr.), 99–125.

Aguilera, M. K., Chen, W., and Toueg, S. 2000b. On quiescent reliable communication. *SIAM Journal on Computing 29,* 6 (Dec.), 2040–2073.

Aguilera., M. K., Chen, W., and Toueg, S. 2000. On quiescent reliable communication. *SIAM J. Comput. 29,* 6, 2040–2073.

Aguilera, M. K., Delporte-Gallet, C., Fauconnier, H., and Toueg, S. 2000. Thrifty generic broadcast. In *Proceedings of the 14th International Symposium on Distributed Computing (DISC)*. Number 1914 in Lecture Notes in Computer Science. Springer-Verlag, Toledo, Spain, 268–282.

Aguilera, M. K., Le Lann, G., and Toueg, S. 2002. On the impact of fast failure detectors on real-time fault-tolerant systems. In *Proceedings of the International Symposium on Distributed Computing (DISC)*. 354–370.

Arora, A. and Kulkarni, S. S. 1998. Detectors and correctors: A theory of fault-tolerance components. In *Proceedings of the IEEE International Conference on Distributed Computing Systems*.

Attiya, H., Bar-Noy, A., and Dolev, D. 1995. Sharing memory robustly in message-passing systems. *Journal of the ACM 42,* 1, 124–142.

Attiya, H., Bar-Noy, A., Dolev, D., Peleg, D., and Reischuk, R. 1990. Renaming in an asynchronous environment. *J. ACM 37,* 3, 524–548.

Attiya, H. and Welch, J. L. 2004. *Distributed Computing: Fundamentals, Simulations and Advanced Topics (2nd edition).* Wiley.

Barborak, M., Dahbura, A., and Malek, M. 1993. The consensus problem in fault-tolerant computing. *ACM Computing Surveys 25,* 2 (June), 171–220.

Beauquier, J. and Kekkonen-Moneta, S. 1997. Fault-tolerance and self-stabilization: impossibility results and solutions using self-stabilizing failure detectors. *International Journal of System Science 28,* 11, 1177–1187.

Ben-Or, M. 1983. Another advantage of free choice: Completely asynchronous agreement protocols. In *Proc. Second Ann. ACM Symp. on Principles of Distributed Computing*. 27–30.

Bernstein, P., Hadzilacos, V., and Goodman, N. 1987. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, Reading, MA.

Borowsky, E. and Gafni, E. 1993. Generalized FLP impossibility result for t-resilient asynchronous computations. In *Proceedings of the 25th ACM Symposium on Theory of Computing (STOC)*. 91–100.

Brasileiro, F., Greve, F., Mostéfaoui, A., and Raynal, M. 2000. Consensus in one communication step. Tech. Rep. PI-1321, IRISA, Rennes, France.

Chandra, T. D., Hadzilacos, V., and Toueg, S. 1996. The weakest failure detector for solving consensus. *Journal of the ACM 43,* 4 (July), 685–722.

Chandra, T. D. and Toueg, S. 1996. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM 43,* 2 (Mar.), 225–267.

Chandy, K. M. and Misra, J. 1988. *Parallel Program Design: A Foundation*. Addison-Wesley, Reading, MA, Reading, Mass.

Charron-Bost, B., Guerraoui, R., and Schiper, A. 2000. Synchronous system and perfect failure detector: Solvability and efficiency issues. In *International Conference on Dependable Systems and Networks (IEEE Computer Society)*.

Charron-Bost, B. and Schiper, A. 2006. The "heard-of" model: Unifying all benign faults. Tech. rep., EPFL. June.

Charron-Bost, B. and Toueg, S. 2001a. Unpublished notes.

Charron-Bost, B. and Toueg, S. 2001b. Unpublished notes.

Chaudhuri, S. 1990. Agreement is harder than consensus: set consensus problems in totally asynchronous systems. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*.

Chen, W., Toueg, S., and Aguilera, M. K. 2000. On the quality of service of failure detectors. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN 2000)*. IEEE Computer Society Press, New York.

Chu, F. 1998. Reducing Ω to $\diamond W$. *Information Processing Letters 67*, 289–293.

Cristian, F. and Fetzer, C. 1999. The timed asynchronous distributed system model. *IEEE Transactions on Parallel and Distributed Systems 10,* 6 (June).

Delporte-Gallet, C., Fauconnier, G., and Freiling, F. C. 2005. Revisiting failure detection and consensus in omission failure environments. In *Theoretical Aspects of Computing - ICTAC 2005, Second International Colloquium, Hanoi, Vietnam*, D. V. Hung and M. Wirsing, Eds. Number 3722 in Lecture Notes in Computer Science. Springer-Verlag, 394–408.

Delporte-Gallet, C., Fauconnier, H., and Guerraoui, R. 2003. Shared memory vs message passing. Tech. Rep. IC/2003/77, EPFL. December. Available at http://icwww.epfl.ch/publications/.

Delporte-Gallet, C., Fauconnier, H., Guerraoui, R., Hadzilacos, V., Kouznetsov, P., and Toueg, S. 2004. The weakest failure detectors to solve certain fundamental problems in distributed computing. In *In Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*. 338–346.

Delporte-Gallet, C., Fauconnier, H., Guerraoui, R., and Kouznetsov, P. 2005. Mutual exclusion in asynchronous systems with failure detectors. *Journal of Parallel and Dustributed Computing (JPDC) 65,* 4 (April), 492–505.

Delporte-Gallet, C., Fauconnier, H., Guerraoui, R., and Tielmann, A. 2008. The weakest failure detector for message passing set-agreement. In *DISC*. 109–120.

Dijkstra, E. W. 1974. Self stabilizing systems in spite of distributed control. *Communications of the ACM 17,* 11, 643–644.

Dijkstra, E. W., Feijen, W. H. J., and van Gasteren, A. J. M. 1983. Derivation of a termination detection algorithm for distributed computations. *Information Processing Letters 16,* 5 (June), 217–219.

Dolev, D., Dwork, C., and Stockmeyer, L. 1987. On the minimal synchronism needed for distributed consensus. *Journal of the ACM 34,* 1 (Jan.), 77–97.

Dolev, D., Friedmann, R., Keidar, I., and Malkhi, D. 1997. Failure detectors in omission failure environments. In *Proceedings of the ACM Symposium on Principles of Distributed Computing, (PODC)e detectors in omission failure environments.*

Dolev, S. 2000. *Self-Stabilization.* MIT Press.

Doudou, A., Garbinato, B., and Guerraoui, R. 2002. Encapsulating failure detection: from crash to Byzantine failures. In *Proceedings of the Int. Conference on Reliable Software Technologies.* Vienna.

Doudou, A., Garbinato, B., and Guerraoui, R. 2005. Tolerating arbitrary failures with state machine replication. In *Dependable Computing Systems: Paradigms, Performance Issues and Applications*, First ed., H. Diab and A. Zomaya, Eds. Addison-Wesley, Reading, MA, Chapter 2.

Doudou, A., Garbinato, B., Guerraoui, R., and Schiper, A. 1999. Muteness failure detectors: specification and implementation. In *In Proceedings of the 3rd European Dependable Computing Conference (EDCC 99).* Number 1667 in Lecture Notes in Computer Science. Springer-Verlag, Prague, Czech Republic, 71–87.

Dwork, C., Lynch, N., and Stockmeyer, L. 1988. Consensus in the presence of partial synchrony. *Journal of the ACM 35,* 2 (Apr.), 288–323.

Eisler, J., Hadzilacos, V., and Toueg, S. 2004. The quorum failure detector and its relation to consensus and registers. Unpublished note.

Eisler, J., Hadzilacos, V., and Toueg, S. 2007. The weakest failure detector to solve nonuniform consensus. *Distributed Computing 19,* 4, 335–359.

Fetzer, C., Schmid, U., and Süsskraut, M. 2005. On the possibility of consensus in asynchronous systems with finite average response times. In *ICDCS*. IEEE Computer Society, 271–280.

Fischer, M. J., Lynch, N. A., and Paterson, M. S. 1985. Impossibility of distributed consensus with one faulty process. *Journal of the ACM 32,* 2 (Apr.), 374–382.

Freiling, F. C. and Völzer, H. 2006. Illustrating the impossibility of crash-tolerant consensus in asynchronous systems. *Operating Systems Review 40,* 2, 105–109.

Gafni, E. 1998. Round-by-round fault detectors: Unifying synchrony and asynchrony (extended abstract). In *PODC*. 143–152.

Garg, V. K. and Mitchell, J. R. 1998a. Distributed predicate detection in a faulty environment. In *Proceedings of the 18th IEEE International Conference on Distributed Computing Systems (ICDCS98).*

Garg, V. K. and Mitchell, J. R. 1998b. Implementable failure detectors in asynchronous systems. In *Proceedings of the 18th Conference on Foundations of Software Technology and Theoretical Computer Science.* Number 1530 in Lecture Notes in Computer Science. Springer-Verlag, Chennai, India.

Gärtner, F. C. and Kloppenburg, S. 2000. Consistent detection of global predicates under a weak fault assumption. In *Proceedings of the 19th IEEE Symposium on Reliable Distributed Systems (SRDS2000).* IEEE Computer Society Press, Nürnberg, Germany, 94–103.

Gärtner, F. C. and Pleisch, S. 2001. (Im)Possibilities of predicate detection in crash-affected systems. In *Proceedings of the 5th Workshop on Self-Stabilizing Systems (WSS2001).* Number 2194 in Lecture Notes in Computer Science. Springer-Verlag, Lisbon, Portugal, 98–113.

Gärtner, F. C. and Pleisch, S. 2002. Failure detection sequencers: Necessary and sufficient information about failures to solve predicate detection. In *Proceedings of the 16th International Symposium on DIStributed Computing (DISC 2002)*, D. Malkhi, Ed. Number 2508 in Lecture Notes in Computer Science. Springer-Verlag, Toulouse, France, 280–294.

Guerraoui, R. 2000. Indulgent algorithms. In *Proceedings of the 19th Annual ACM Symposium on Principles of Distributed Computing (PODC-00)*. ACM Press, NY, 289–298.

Guerraoui, R. 2002. Non-blocking atomic commitment in asynchronous systems with failure detectors. *Distributed Computing 15,* 1, 17–25.

Guerraoui, R., Herlihy, M., Kouznetsov, P., Lynch, N. A., and Newport, C. C. 2007. On the weakest failure detector ever. In *PODC*. 235–243.

Guerraoui, R., Hurfin, M., Mostéfaoui, A., Oliveira, R., Raynal, M., and Schiper, A. 1999. Consensus in asynchronous distributed systems: A concise guided tour. In *Advances in Distributed Systems*, S. Krakowiak and S. K. Shrivastava, Eds. Lecture Notes in Computer Science, vol. 1752. Springer, 33–47.

Guerraoui, R., Kapalka, M., and Kouznetsov, P. 2008. The weakest failure detectors to boost obstruction-freedom. *Distributed Computing 20,* 6, 415–433.

Guerraoui, R. and Kouznetsov, P. 2008. Failure detectors as type boosters. *Distributed Computing 20,* 5, 343–358.

Guerraoui, R. and Kuznetsov, P. 2008. The gap in circumventing the impossibility of consensus. *J. Comput. Syst. Sci. 74,* 5, 823–830.

Guerraoui, R. and Schiper, A. 1996. "Gamma-accurate" failure detectors. In *Distributed Algorithms, 10th International Workshop, WDAG '96*, Ö. Babaoglu and K. Marzullo, Eds. Lecture Notes in Computer Science, vol. 1151. Springer-Verlag, Bologna, Italy, 269–286.

Guerraoui, R. and Schiper, A. 1997. Genuine atomic multicast. In *Proceedings of the 11th International Workshop on Distributed Algorithms (WDAG97)*. Number 1320 in Lecture Notes in Computer Science. Springer-Verlag, 141–154.

Hadzilacos, V. 1984. Issues of fault tolerance in concurrent computations. Ph.D. thesis, Harvard University. also published as Technical Report TR11-84.

Hadzilacos, V. and Toueg, S. 1994. A modular approach to fault-tolerant broadcasts and related problems. Tech. Rep. TR94-1425, Cornell University, Computer Science Department. May.

Haeberlen, A., Kouznetsov, P., and Druschel, P. 2007. Peerreview: practical accountability for distributed systems. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP)*. 175–188.

Herlihy, M. and Shavit, N. 1999. The topological structure of asynchronous computability. *Journal of the ACM 46,* 6 (November), 858–923.

Herlihy, M. and Wing, J. M. 1990. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems 12,* 3 (June), 463–492.

Hermant, J. and Le Lann, G. 2002. Fast asynchronous uniform consensus in real-time distributed systems. *IEEE Transactions on Computers 51,* 8 (Aug.), 931–944.

Hermant, J.-F. and Widder, J. 2005. Implementing reliable distributed real-time systems with the theta-model. In *In Proceedings of the 9th International Conference on Principles of Distributed Systems (OPODIS 2005)*.

Hurfin, M., Mostéfaoui, A., and Raynal, M. 1998. Consensus in asynchronous systems where processes can crash and recover. In *Proceedings of the 17th IEEE Symposium on Reliable Distributed Systems (SRDS'98)*. IEEE Computer Society Press, West Lafayette, Indiana, 280–286.

Hurfin, M. and Raynal, M. 1999. A simple and fast asynchronous consensus protocol based on a weak failure detector. *Distributed Computing 12,* 4, 209–223.

Hutle, M. and Widder, J. 2005. On the possibility and the impossibility of message-driven self-stabilizing failure detection. In *Self-Stabilizing Systems, 7th International Symposium, SSS 2005, Barcelona, Spain*, T. Herman and S. Tixeuil, Eds. Lecture Notes in Computer Science, vol. 3764. Springer-Verlag, 153–170.

Israeli, A. and Li, M. 1993. Bounded time-stamps. *Distributed Computing 6,* 4 (July), 205–209.

Jayanti, P. and Toueg, S. 2008. Every problem has a weakest failure detector. In *PODC*. 75–84.

Kihlstrom, K. P., Moser, L. E., and Melliar-Smith, P. M. 2003. Byzantine fault detectors for solving consensus. *The Computer Journal 46,* 1.

Lamport, L. 1978. Time, clocks and the ordering of events in a distributed system. *Communications of the ACM 21,* 7 (July), 558–565.

Lamport, L. 1998. The part-time parliament. *ACM Transactions on Computer Systems 16,* 2 (May), 133–169.

Lamport, L., Shostak, R., and Pease, M. 1982. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems 4,* 3 (July), 382–401.

Larrea, M., Fernández, A., and Arévalo, S. 2000. Optimal implementation of the weakest failure detector for solving consensus. In *Proceedings of the 19th IEEE Symposium on Reliable Distributed Systems (SRDS2000)*. IEEE Computer Society Press, Nürnberg, Germany.

Larrea, M., Fernández, A., and Arvalo, S. 2000. Eventually consistent failure detectors. Tech. rep., Universidad Pública de Navarra, Spain. Apr. Presented as a brief announcement at DISC2000.

Lo, W.-K. and Hadzilacos, V. 1994. Using failure detectors to solve consensus in asynchronous shared-memory systems (extended abstract). In *Proceedings of the 8th International Workshop on Distributed Algorithms (WDAG94)*, G. Tel and P. M. B. Vitányi, Eds. Lecture Notes in Computer Science, vol. 857. Springer-Verlag, Terschelling, The Netherlands, 280–295.

Long, D. D. E., Carroll, J. L., and Park, C. J. 1991. A study of the reliability of Internet sites. In *Proceedings of the 10th IEEE Symposium on Reliable Distributed Systems (SRDS91)*. 177–186.

Malkhi, D. and Reiter, M. 1997. Unreliable intrusion detection in distributed computations. In *Proceedings of the 10th Computer Security Foundations Workshop (CSFW97)*. Rockport, MA, 116–124.

Matsui, H., Inoue, M., Masuzawa, T., and Fujiwara, H. 2000. Fault-tolerant and self-stabilizing protocols using an unreliable failure detector. *IEICE Transactions E83-D,* 10 (Oct.), 1831–1840.

Mittal, N., Freiling, F. C., Venkatesan, S., and Penso, L. D. 2005. Efficient reduction for wait-free termination detection in a crash-prone distributed system. In *DISC*. 93–107.

Mostéfaoui, A., Raynal, M., and Travers, C. 2006. Exploring Gafni's reduction land: From *mega* to wait-free adaptive (2p-[p/k])-renaming via k-set agreement. In *DISC*. 1–15.

Neiger, G. 1995. Failure detectors and the wait-free hierarchy. In *Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing (PODC'95)*. 100–109.

Oki, B. and Liskov, B. 1988. Viewstamped replication: A general primary copy method to support highly available distributed systems. In *Proceedings of the 7th Annual ACM Symposium on Principles of Distributed Computing (PODC'88)*. 8–17.

Oliveira, R., Guerraoui, R., and Schiper, A. 1997. Consensus in the crash-recover model. Tech. Rep. TR-97/239, EPFL – Départment d'Informatique, Lausanne, Switzerland. Aug.

Paxson, V. and Adams, A. 2002. Experiences with NIMI. In *Proceedings of the 2002 Symposium on Applications and the Internet*.

Pedone, F. and Schiper, A. 1999. Generic broadcast. In *Proceedings of the 13th International Symposium on Distributed Computing (DISC'99)*.

Powell, D. 1992. Failure mode assumptions and assumption coverage. In *Proceedings of the 22nd Annual International Symposium on Fault-Tolerant Computing (FTCS '92)*, D. K. Pradhan, Ed. IEEE Computer Society Press, Boston, MA, 386–395.

Raynal, M. 2002. Consensus in synchronous systems: A concise guided tour. In *PRDC '02: Proceedings of the 2002 Pacific Rim International Symposium on Dependable Computing*. IEEE Computer Society, Washington, DC, USA, 221.

Raynal, M. 2005. A short introduction to failure detectors for asynchronous distributed systems. *SIGACT News 36,* 1, 53–70.

Raynal, M. and Travers, C. 2006. In search of the holy grail: Looking for the weakest failure detector for wait-free set agreement. In *OPODIS*. 3–19.

Sabel, L. S. and Marzullo, K. 1995. Election vs. consensus in asynchronous systems. Tech. Rep. TR95-1488, Cornell University, Computer Science Department. Feb.

Saks, M. E. and Zaharoglou, F. 2000. Wait-free k-set agreement is impossible: The topology of public knowledge. *SIAM J. Comput. 29,* 5, 1449–1483.

Schiper, A. 1997a. Early consensus in an asynchronous system with a weak failure detector. *Distributed Computing 10,* 3, 149–157.

Schiper, A. 1997b. Erratum: Early consensus in an asynchronous system with a weak failure detector. *Distributed Computing 10*, 198.

Schlichting, R. D. and Schneider, F. B. 1983. Fail stop processors: An approach to designing fault-tolerant computing systems. *ACM Transactions on Computer Systems 1,* 3 (Aug.), 222–238.

Schneider, F. B. 1990. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys 22,* 4 (Dec.), 299–319.

Schneider, F. B. 1993. What good are models and what models are good? In *Distributed Systems*, Second ed., S. Mullender, Ed. Addison-Wesley, Reading, MA, Chapter 2, 17–26.

Sergent, N., Défago, X., and Schiper, A. 1999. Failure detectors: implementation issues and impact on consensus performance. Tech. Rep. SSC/1999/019, École Polytechnique Fédérale de Lausanne, Switzerland.

Tanenbaum, A. S. 1996. *Computer Networks*, Third ed. Prentice-Hall, Englewood Cliffs, NJ, USA.

Turek, J. and Shasha, D. 1992. The many faces of consensus in distributed systems. *IEEE Computer 25,* 6 (June), 8–17.

Vitányi, P. and Awerbuch, B. 1986. Atomic shared register access by asynchronous hardware. In *Proceedings of the 27th Symposium on Foundations of Computer Science*. 233–246.

Völzer, H. 2004. Randomization versus synchronization in distributed systems. In *Proceedings 31st Int. Colloquium on Automata, Languages, and Programming (ICALP 2004)*. Number 3142 in Lecture Notes in Computer Science. Springer Verlag, 1214–1226.

Völzer, H. 2005. On conspiracies and hyperfairness in distributed computing. In *Proceedings of the 19th International Symposium on Distributed Computing, DISC 2005*. Number 3724 in Lecture Notes in Computer Science. Springer-Verlag, 33–47.

Zielinski, P. 2007. Automatic classification of eventual failure detectors. In *DISC*. 465–479.

Zielinski, P. 2008. Anti-omega: the weakest failure detector for set agreement. In *PODC*. 55–64.

A. HANDLING A BIVALENT CRITICAL INDEX

In this appendix we fill a gap in the necessity part of CHT (Section 3.1). The basic question we answer here is the following: how can the identity of a correct process be determined if, for some stabilized critical index i, the input vector I_{i-1} is 0-valent and the vector I_i is bivalent? In contrast to the case studied before (where I_i was 1-valent), the correct process is not necessarily p_i. The proof is slightly more complicated than before. In fact, the proof for this part needs almost as many pages (and even more figures) to explain than the entire proof that was presented up to now. In some sense, we are advising the readers to read this appendix only if absolutely necessary.

A.1 Simulation Tree

Remember that a process runs a simulation (using the given consensus algorithm A) for all input vectors and for all paths through the current version of its DAG. Eventually, (provided the process does not crash) the result of this is a huge set of simulated runs of A. Let us only consider those runs that start with the (bivalent) input vector I_i (where i is the critical index). We can combine all of them into a

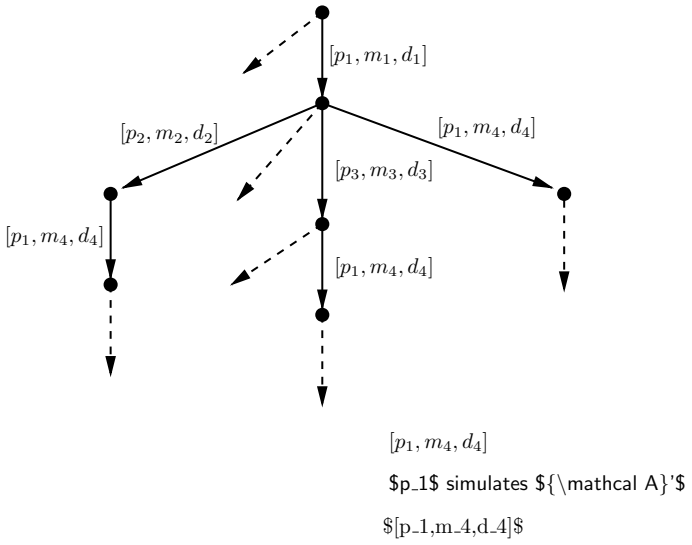

Fig. 8. Possible simulation tree constructed from the DAG in Fig. 5.

simulation tree. The root of the tree is the starting configuration. Every edge in the tree corresponds to a step taken by some process and every vertex corresponds to a configuration resulting from the steps leading to it. Runs of consensus that "share a prefix" (i.e., start in the same way) have the same prefix in the tree. Basically, the tree represents all choices of next steps during the simulation, much like a game tree in chess for example.

A step consists of three items and is represented as $[p, m, d]$. The value p denotes which process executed the step, m identifies which message that process consumed (if any) and d represents the output of the failure detector during this step. For example, the simulation tree using the input of the DAG in Fig. 5 could look like the one depicted in Fig. 8.

Note that there are many more choices that have not been represented in this tree: for example, the DAG will consist of increasingly many possible samples to choose from. Also there is a choice of the message m which a process should consume in the simulation. (In fact, a simulated process can consume an "empty" message in a step: this models arbitrary message delays.) So one "step" in the DAG offers multiple choices (and hence path continuations) in the simulation tree.

The individual states of the tree also carry a valency: if in that state the consensus algorithm has terminated with a decision of 0 or 1, then that state is 0-valent or 1-valent, respectively. For states, in which the consensus algorithm has not yet terminated, we have the following rule: if a state s has a followup state (a descendant in the tree) which is 1-valent, then s is also 1-valent. Similarly for 0-valent descendant states. However, if s has both a 1-valent and a 0-valent descendant state, then s is *bivalent*. Recall that the starting state of the simulation tree (the root) must be bivalent.

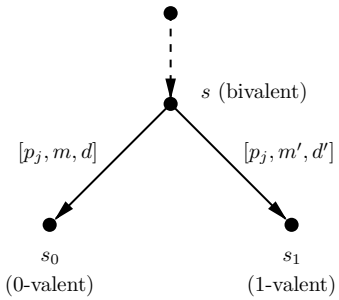

Fig. 9. A fork.

A.2 Determining a Correct Process: Hooks and Forks

As mentioned before, the correct process is in general not p_i anymore (where i is the critical index) but some other process. The idea is to use here certain patterns in the simulation tree to identify a correct process. These patterns are called *forks* and *hooks*. In the following, we first look at how hooks and forks allow the processes to determine a correct process. Then we argue that every simulation tree for I_i contains at least one hook or one fork which does not vanish. Let us consider forks first (this is the easier case).

A.2.1 *Forks.* The picture shown in Fig. 9 identifies a *fork*. A fork consists of a bivalent state s from which two *different* steps by the *same* process p_j are possible which, on the one hand, lead into a 0-valent state s_0 and, on the other hand, to a 1-valent state s_1.

The point to note now is that if we can find a fork in the simulation tree, then p_j must be a correct process. To see this, assume that p_j is faulty. Like in the main part of this text, we can now exploit a property of the DAG (its transitive closure). Starting from state s_0, there is a continuation π_0 in the simulation tree in which p_j never takes a step. Since s_0 is 0-valent, the processes decide 0 in π_0. Similarly, starting from state s_1 there is a continuation π_1 in which p_j also does not take any step, and in which the other processes take exactly the same steps as in π_0. Since s_1 is 1-valent, the processes decide 1.

There is a contradiction leaking here: the only difference between π_0 and π_1 is the state of p_j. But p_j is crashed and so processes should come to the same decision in π_0 and π_1, not different ones. So p_j must be correct.

A.2.2 *Hooks.* Let us look at hooks now. A *hook* in the simulation tree has the structure shown in Fig. 10. It consists of a bivalent state s, a state s' that is reached by executing a step of some process p_k, and two states s_0 and s_1 reached by executing *the same* step of process p_j. Additionally, state s_0 must be 0-valent and s_1 must be 1-valent (or vice versa; the order does not matter here).

If we can find a hook in the simulation tree, then we are done: in this case, p_k is a correct process. The reason for this can be derived by similar arguments as in the case of a fork: if p_k were faulty, then we could construct extensions of s_0 and s_1 in which p_k takes no step, but where all other processes take the same steps. Hence, they should reach the same decision. But the valencies of s_0 and s_1 are different —

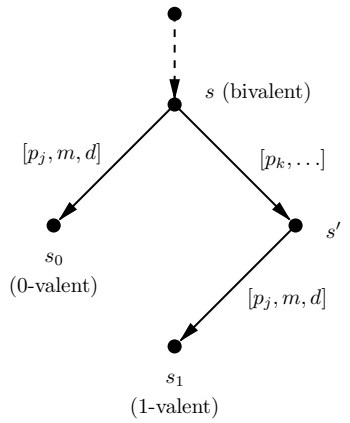

Fig. 10. A hook.

$s :=$ ⟨root of simulation tree⟩
while *true* **do**
  $j :=$ ⟨choose the next correct process in a round robin fashion⟩
  $m :=$ ⟨choose the oldest message for process p_j⟩
  if ⟨s has descendant s' (possibly $s = s'$) such that, for some d,
    s' extended with $[p_j, m, d]$ is a bivalent vertex of the tree ⟩
  then $s := s'$ extended with $[p_j, m, d]$
  else exit
end

Fig. 11. Finding a hook or a fork.

a contradiction! So p_k must be correct.

A.3 Existence of Hooks and Forks

If we have found a hook or a fork in the simulation tree, then we are lucky: we can extract the common correct leader. If there are many hooks and forks in the simulation tree, then we need to prevent different processes from looking at different hooks and outputting different values. This is done by defining the notion of a "first" hook or fork. Basically this is possible since the vertexes of the simulation are countable: the processes can then select the hook or fork with the "smallest" root state s and extract from it the leader.

But what if all newly computed hooks or forks disappear from the simulation tree and never re-appear? In fact, this cannot happen. Eventually, and as we argue below, there will be a hook or fork in the simulation tree that does not go away. Take an *infinite* bivalent simulation tree. We consider a hypothetical algorithm that goes through the simulation tree (see Fig. 11). The algorithm terminates only when a hook or a fork has been found.

Basically the algorithm locates a *fair path* through the simulation tree, i.e., a path in which all correct processes get scheduled infinitely often and every message sent to a correct process is eventually consumed. Additionally, this fair path goes through bivalent states only.

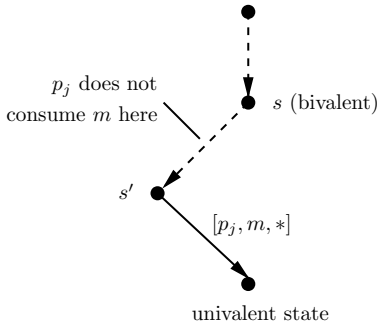

Fig. 12. Process p_j takes a step at descendant s'.

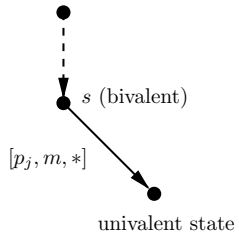

Fig. 13. Simplified tree of Fig. 12.

There are two questions to ask now: (1) Does this algorithm always terminate? (2) If it terminates, where is the hook or the fork?

A.3.1 *Terminating the Infinite Simulation Tree.* Suppose, by contradiction, that the algorithm does not terminate. That is, we can locate an *infinite* path in the simulation tree that goes through bivalent states only. Moreover, the path corresponds to a fair run of the consensus algorithm A: every correct process takes an infinite number of steps in the run and eventually receives every message sent to it. Note that a state in which some process decides a value $v \in \{0, 1\}$ cannot be bivalent, otherwise the agreement property of consensus would be violated. In this case, we end up with a fair run of algorithm A in which no process ever decides: a contradiction with the termination property of consensus.

A.3.2 *Identifying a Fork or a Hook.* So we have seen that at some point the algorithm terminates. What condition holds at that point? To find out, we simply have to negate the condition of the **if** statement. The negation reads as follows: state s has no descendant in the tree in which p_j takes a step consuming message m and where the resulting state is bivalent. In other words, any step of p_j consuming message m brings any descendant of s (including s itself) to either a 1-valent or a 0-valent state. Let us look at some extension of s in which p_j takes a step at some descendant s'. The situation is depicted in Fig. 12. Since the descendant can be state s itself, the structure shown in Fig. 13 is also a legal tree.

Without loss of generality, we assume that some step $[p_j, m, d']$ brings s to a 0-valent state. That is, our simulation tree contains the subtree shown if Fig. 14.

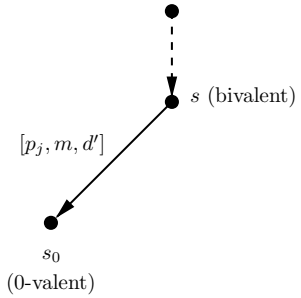

Fig. 14. Subtree containing 0-valent state s_0.

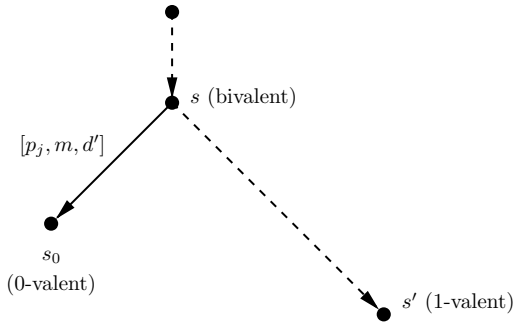

Fig. 15. Subtree containing 0-valent state s_0 and 1-valent state s'.

Since s is a bivalent state, it must have some descendant s' that is 1-valent. This is shown in Fig. 15. Let us assume that p_j takes no step between s and s' in which it consumes message m. This means that m is still unconsumed in state s' and some step of the form $[p_j, m, d]$ is applicable there. Since s' is 1-valent, the step $[p_j, m, d]$ applied to s' results in a 1-valent state. The resulting situation is shown in Fig. 16.

We can also come up with a situation like this even if p_j takes a step between s and s' in which it consumes m. So assume that p_j takes a step consuming m in some intermediate state, say, s'', on the path from s to s'. The resulting state cannot be 0-valent. Why? Because we know that s' is 1-valent, and so the resulting state must be at least bivalent! But it cannot be bivalent because the termination condition of the algorithm which identified s disallowed bivalent followup states of actions of p_j. So overall the situation then looks like the one depicted in Fig. 17. In both cases, our simulation tree contains the subtree shown in Fig. 18 where s''' is either s' or s''.

We claim that within the structure depicted in Fig. 18 there must be a fork or a hook somewhere. The argument is quite simple: the path between s and s''' consists of a finite sequence of steps $e_1, e_2, \ldots, e_m$. Since p_j does consume m between s and s''', the very same step $[p_j, m, d]$ can be applied in the first state following s on the way to s''', i.e., after executing step e_1. Similarly, it can be applied after executing step e_1 and e_2. In fact, it can be executed in *every* state on the way from s to s'''. We denote the corresponding intermediate states by $\sigma_0 = s, \sigma_1, \ldots, \sigma_{m-1}, \sigma_m = s'''$

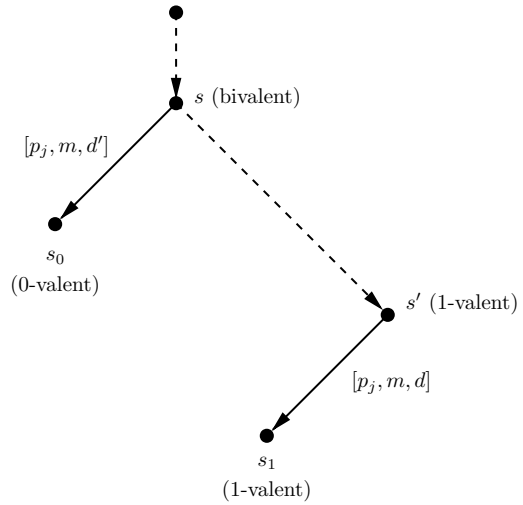

Fig. 16. Applying p_j's step to s or s'.

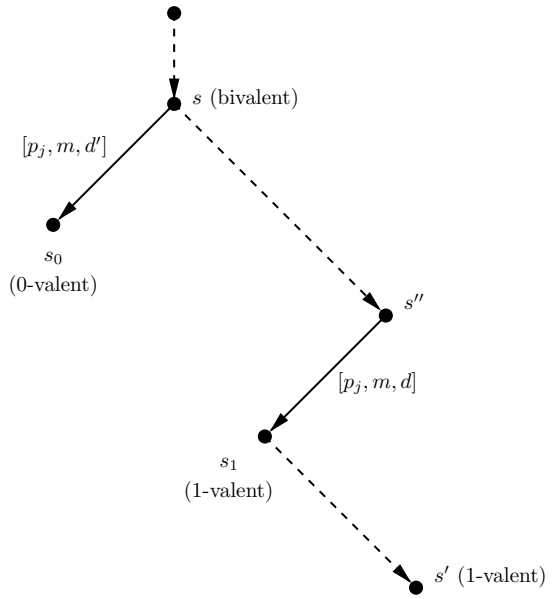

Fig. 17. Approaching a hook or fork.

(see Fig. 19).

The important point here is to see that, for any $k = 0, \ldots, m$, the step $[p_j, m, d]$ applied to σ_k results in a 0-valent or a 1-valent state. This is because the algorithm that identified s terminated and the termination condition stated that there is *no descendant of* s that is bivalent after applying that ominous step of p_j.

Let $k \in \{0, \ldots, m\}$ be the lowest index such that $[p_j, m, d]$ brings σ_k to a 1-valent

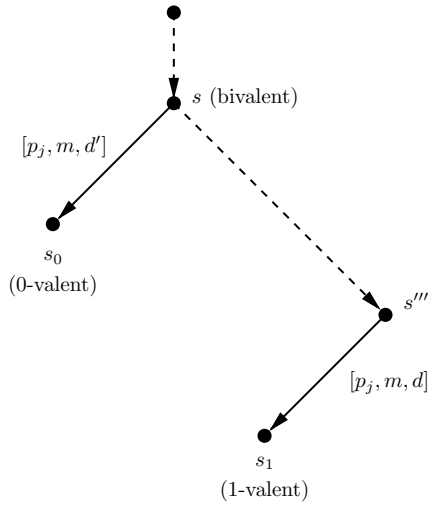

Fig. 18. Where is the hook or fork?

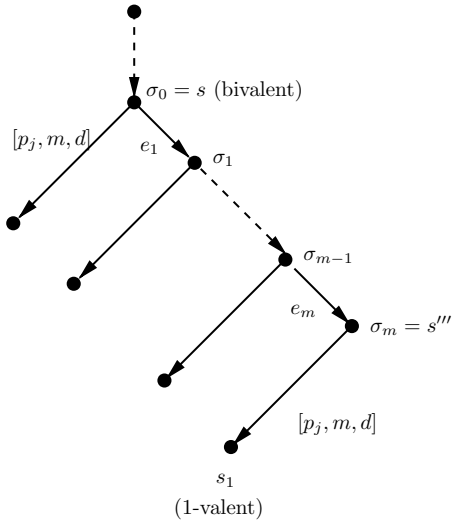

Fig. 19. Finding a fork.

state. We know that such an index exists, since s_1 is 1-valent and all such resulting states are either 0-valent or 1-valent.

Now we have the following two cases to consider: (1) $k = 0$, and (2) $k > 0$.

Let us assume that $k = 0$, i.e., $[p_j, m, d]$ applied to s brings it to a 1-valent state. But we know that there is a step $[p_j, m, d']$ that brings s to a 0-valent state. A fork is located! As a result, we identified p_j as a correct process.

If $k > 0$, we have the situation shown in Fig. 20. That's a hook! If e_k was executed by process p_k, then we have identified p_k as a correct process following the argument about hooks above.

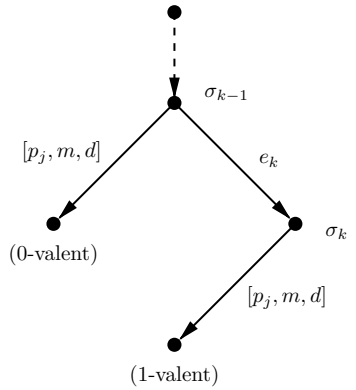

Fig. 20. A hook is found.

Thus, a bivalent infinite simulation tree has at least one *finite* subtree that allows us to compute a single correct process.