The Failure Detector Abstraction

Felix C. Freiling, University of Mannheim
and
Rachid Guerraoui, EPFL and MIT CSAIL
and
Petr Kouznetsov, MPI-SWS

University of Mannheim
Faculty of Mathematics and Computer Science
D-68131 Mannheim, Germany

Technical Report TR 2006-003
Feb. 10, 2006

This paper surveys the *failure detector* concept through two dimensions. First we study failure detectors as building blocks to simplify the design of reliable distributed algorithms. More specifically, 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 highlights 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 or machine. In the early times, abstractions like record, set or arrays helped the emancipation from assemblies and machine languages.

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 the possible differences between languages and operating systems on different machines, and encapsulate serialization and de-serialization mechanisms to transfer data over the wire.

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

This paper is structured into three parts which can be read independently: 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.

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 define here the notion of a failure detector in a precise manner. Our representation is pretty intuitive and is "time-free". It is in this sense more coarse-grained than in [Chandra et al. 1996]. Basically, our definition of a failure pattern is simpler and does not take into account *when* exactly failures occur. For the purpose of our survey, we believe it to be sufficient to consider a "time-free" representation.

In the third part (Section 4), we take some perspective to underly some limitations of the failure detector abstraction .

2. FAILURE DETECTOR AS A PROGRAMMING BUILDING BLOCK

Information about the operational state 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 standard methods of implementing failure detection (based on timeouts) and discuss their problems. In Section 2.2 we then informally introduce the failure detector abstraction and argue that it has several advantages over 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 allows to express information about failures in a way which 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 even has the potential of building more efficient applications.

2.1 Failure Detection using Timeouts

2.1.1 *Non-Blocking Atomic Commit.* Consider the omnipresent problem of non-blocking atomic commit in a distributed database [Bernstein et al. 1987, Chapter 7]. In a distributed database, data is stored at multiple sites, usually close to the location where it is used so that read and write operations on the data can be performed more efficiently. A distributed transaction groups a sequence of read and write operations together and ensures that either all are executed or none of them. For example, a bank might choose to store customer account data at the local branch closest to the customer's home. A debit of, say, $100 then involves a read and a write of the local data, but it may be accompanied by a log update for statistical purposes or the query of a customer's clearance at a central server. A transaction ensures that these operations are executed *atomically* despite site or communication failures. For simplicity, we will identify a *site* of the distributed database with the *process* of the database management system running on that site.

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 which 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 which does not crash a *correct* process.

2.1.2 *Three-Phase Commit. Three-phase commit* (3PC) is a well-known protocol to implement non-blocking atomic commit. 3PC refines the popular *two-phase commit* (2PC) protocol, which 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 which votes *yes*, and stops. Otherwise the coordinator sends a pre-commit message to all processes.

(4) A process which 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. 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 which 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*).

In the 3PC algorithm there are several places which should be enhanced with a timeout mechanism. For example, in step 2, processes wait for a vote request from the coordinator. The relevant part of the algorithm is depicted at the top of Fig. 1.

```
**wait for** ⟨vote request⟩ **from** $p_1$
```

```
$timeout := clock + \rho$
while $timeout > clock \wedge \neg\langle$vote request arrived from $p_1\rangle$ do
  skip
endwhile
if $\neg\langle$vote request arrived from $p_1\rangle$ then
  ⟨decide abort⟩
endif
```

Fig. 1. Adding explicit timeout actions to potentially blocking program statements in three-phase commit.

If a process fails to receive such a message, it can unilaterally decide *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. From the figure it should be clear that adding explicit timeouts to the algorithm quickly obscures the code and makes correctness arguments much more tricky.

Every statement in the 3PC algorithm which could potentially block needs to be enhanced with a similar timeout construct. Choosing the correct action upon timeout is rather tricky in some cases. For example, if a process times out on either a pre-commit message (in step 4) or the final commit message (in step 6) from the coordinator, it needs to communicate with the other processes since the coordinator has presumably failed. In this case, a termination protocol is invoked: All surviving processes elect a new coordinator which collects the individual states of all participating processes and enforces a consistent decision. Election is based on some pre-determined order on processes (like the process identifier). The termination protocol is reentrant: if the newly elected coordinator fails within the termination protocol, the same protocol is started anew. In the end, either one correct process will eventually survive and enforce a decision or all processes will fail.

2.1.4 *Correctness of Timeout-Based Solutions.* 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 which 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. 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. However, the usefulness and efficiency of the timeout-based 3PC algorithm above crucially depend on good reasoning via timeouts. Continued premature expiration of a too small timeout may prevent any transaction from commiting successfully. 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 shouldn't 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) and assume that time is measured in the number of *steps* which a process has executed. This 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 at some point k in the execution, then m must be delivered after at most Δ execution steps of the sending process following k.

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

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

First, it takes at most Δ steps of the sending process p for the message to travel from p to q. Then, after at most δ steps of process p the receiving process must have executed at least one step (which should include 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*. If a system can be characterized as synchronous, then timeout-based reasoning always leads to correct conclusions.

In practice, the beforementioned 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.* 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 which have no timing assumptions whatsoever. These systems are usually refered 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.

These points 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 at some point 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 usually modeled as an asynchronous system.

2.1.7 *Timeouts in the Asynchronous Model.* 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 which 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].

2.1.8 *Summary.* In many practical situations (like in 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. Algorithms which use explicit timeouts quickly become very ugly. Moreover, timeout-based reasoning is only valid in systems with strict timing guarantees. In other types of systems, timeout-based reasoning is uncertain or even impossible. This is unfortunate since this weakens the usefulness of timeout-based reasoning exactly in those (asynchronous) systems which are considered most common in practice. A path out of this dilemma is to introduce the abstraction of a *failure detector*, which we discuss in the following sections.

2.2 Failure Detectors as Useful Distributed Services

In the previous sections we have already talked about using the synchrony bounds for detecting crash failures of other processes. However, this perspective has mixed 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 could write a correct 3PC algorithm in systems without any explicit synchrony assumptions as long as we have a means to still detect process crashes. Separating these concerns is at the heart of the concept of *failure detectors*, which 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.

2.2.1 *Perfect Failure Detectors.* In the understanding of Chandra and Toueg [1996], a failure detector is composed of several failure detector modules, one at each process. To introduce the concept, consider a process p which is equipped with a failure detector module which indicates the operational state (*up*/*down*) of a process q. If the failure detector responds with *down*, we say that the failure detector at p *suspects* 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.

Obviously, the failure detector can be extended 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:
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:
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.

2.2.2 *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. Because of this, a failure detector is often called an *oracle*. 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 which encapsulates synchrony assumptions in an asynchronous interface.

2.2.3 *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. Now the code 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 refering 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 if the type of failure detector is used.

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

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

2.2.4 *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 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 which 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 which is only perfect after some finite time (before this time it can behave arbitrarily). The idea of the algorithm 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 *indugent* [Guerraoui 2000].

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] 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 and can be achieved by "extinguishing" all other values from the system. To lock a value, it is necessary that a majority of processes are correct.

It is rather easy to show that there is no indugent 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 which (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.

2.2.5 *Unreliable Failure Detectors.* The existence of a perfect failure detector is a very strong assumption which 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.

A systematic way to weaken the specification is to try different combinations of quantors. For example, where strong accuracy read "$\forall p : \forall q : \ldots$" we could write "$\forall p : \exists q : \ldots$". Among these weaker variants it turns out that the combination $\exists q : \forall p$ for safety and $\forall q : \exists p$ for liveness are useful for solving consensus. What do these combinations mean?

—$\exists$ a correct process $q : \forall p$: the failure detector module at p does not suspect q unless q has crashed.

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 suspect q.
In terms of liveness this means that for every crash, there is a process which 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 which are allowed to make mistakes fall into the category of *unreliable failure detectors*.

	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		weak $\mathcal{W}$		eventually weak $\diamond\mathcal{W}$

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

Interestingly (as will be discussed in Section 3), the eventually strong failure detector $\diamond\mathcal{S}$ is the *weakest* failure detector for solving consensus, given that a majority of processes are correct. Intuitively, this means that it provides the least level of timing information to make consensus solvable. Hence, systems which offer much less timing guarantees than a perfect failure detector can be used to implement consensus abstractions. Since the underlying failure detector $\diamond\mathcal{S}$ is in this sense connected to the problem of consensus, there are implementations of $\diamond\mathcal{S}$ which are optimized with respect to efficiently solving consensus [Larrea et al. 2000; Chen et al. 2000; Sergent et al. 1999]. This will be important later when we talk about combining different failure detectors in a single application.

2.2.6 *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 which 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* which is useful in designing protocols which are *quiescent*, i.e., which 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 anonymously perfect failure detector (denoted $?\mathcal{P}$) [Guerraoui 2002; Charron-Bost and Toueg 2001] is useful in the context of solving non-blocking atomic commit. This failure detector 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 this section.

2.2.7 *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.

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 ratio between the average length of a stable period to that of an unstable period is [[341 : 1]] on a standard local area network. The bottom line of this observation is that failure detection can be implemented perfectly "most of the time".

The average length of a stable period was [[x seconds,]] a time which is usually sufficient for an algorithm to terminate, e.g., a transaction to commit. Therefore, an arguably assumption is that the system is synchronous "forever" after an initial finite time of asynchrony. This is captured in what became known as the assumption of *partial synchrony* [Dwork et al. 1988].

There are two possible variants of partial synchrony, which 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.

Interestingly, both forms of partial synchrony allow consensus to be solved, even if both communication and processes are partially synchronous [Chandra and Toueg 1996] and even if merely the ratio between best-case and worst-case round trip delay is bounded [Jean-François Hermant and Josef Widder]. In terms of failure detectors, such partially synchronous systems allow to implement eventually perfect failure detectors [Chandra and Toueg 1996]. Hence, reasoning with eventually perfect failure detectors can be justified in practice.

2.2.8 *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 which 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 are 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 simultanteously. Dissemination of failure detection messages and keeping track of timeouts can be done centrally at a "middleware" layer which 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 which 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.

```
**non-blocking_atomic_commit**($vote_i$) **is**
  **send** $\langle p_i, vote_i \rangle$ **to all**
  **wait until** $\forall j \in \{1, \dots, n\}$ : **received** $\langle p_j, vote_j \rangle$ **or** $\top \in ?\mathcal{P}$
  **if** $\top \in ?\mathcal{P}$ **or** $\exists j \in \{1, \dots, n\} : vote_j = no$ **then**
    $outcome_i := consensus(abort)$
  **else**
    $outcome_i := consensus(commit)$
  **endif**
  **return**($outcome_i$)
**end**
```

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

Finally, we argue here that failure detectors also remedy the problems of asymmetric or differing timeouts within an application which was discussed at the end of the previous section. 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 different concerns can be captured using two different types of failure detectors. It the first phase, it is not important *which* process failed, so the anonymously perfect failure detector $?\mathcal{P}$ is sufficient (recall that $?\mathcal{P}$ outputs $\perp$ if no process has failed and $\top$ 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 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 which makes it easier to design, build and analyse fault-tolerant distributed systems. Secondly, implementation of the failure detection functionality can be done in a centralized, re-useable fashion which enables solutions which 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 which is more suitable to be used directly by application logic than explicit timing information.

3. FAILURE DETECTORS AS A COMPUTABILITY BENCHMARK

Failure detector is a means to abstract out time in distributed programming. A program assuming a failure detector $\mathcal{D}$ works in any physical model where $\mathcal{D}$ is implementable. In a sense, a failure detector hides the synchrony guarantees of the underlying model from the programmer. The principal theoretical question here is what are the minimal synchrony assumptions sufficient to solve a given problem M, or, in other words, the *weakest failure detector* to solve M.

In the next section (Section 3.1), we present formally the model of an asynchronous system with failure detectors and give a few examples of failure detectors. In the following sections (Sections 3.2, 3.3 and 3.4), we discuss the weakest failure detector question in the context of three fundamental problems in distributed computing: solving consensus (see Section 2.2.4 for a definition), implementing read-write shared memory in a message-passing system and solving non-blocking atomic commitment (NBAC).

3.1 Model

In this chapter, we describe the asynchronous message-passing model equipped with a failure detector [Chandra et al. 1996].

Processes. The system consists of a set of n processes $\Pi = \{p_1, p_2, \ldots, p_n\}$ ($n > 1$). Every pair of processes is connected by a reliable channel. Processes communicate by reliable message passing.

Failures and failure patterns. Processes are subject to *crash* failures. We do not consider Byzantine failures: a process either correctly executes the algorithm assigned to it, or crashes and stops forever executing any action. A process that does not crash in a given execution is called *correct.* A process that is not correct is called *faulty.* A *failure pattern* F, a proper subset of Π, describes the set of processes that are faulty in the same execution. An *environment* $\mathcal{E}$, a set of failure patterns, describes sets of processes that are allowed to fail in the same execution.

A *failure detector history* H *with range* $\mathcal{R}$ is an infinite sequence (p_{i_1}, d_1),(p_{i_2}, d_2),. . ., where for all $j \in \mathbb{N}$, $p_{i_j} \in \Pi$ and $d_j \in \mathcal{R}$. Informally, H represents the order in which processes p_{i_1},p_{i_2},... "see" failure detector outputs d_1,d_2,.... For a given failure pattern F, we say that a set S of failure detector histories is F-*complete* if S satisfies the following properties:

(1) For every infinite sequence σ of process identifiers $p_{i_1}, p_{i_2}, \ldots$ in which processes in F appear finitely often and processes in $\Pi \setminus F$ appear infinitely often, S contains a failure detector history $(p_{i_1}, d_1), (p_{i_2}, d_2), \ldots$.

(2) Let $H \in S$, and H' be any failure detector history obtained from H by removing finitely many elements. Then $H' \in S$.

Let $k \in \mathbb{N}$ and $H = (p_{i_1}, d_1), (p_{i_2}, d_2), \ldots$ be a failure detector history. We say that a process p is *faulty in step* k *of* H if p appears in H only before its k-th element, i.e., $\forall k' \geq k$, $p_{i_{k'}} \neq p$.

Failure detectors. A *failure detector* $\mathcal{D}$ with range $\mathcal{R}_\mathcal{D}$ is a function that maps each failure pattern F to an F-complete set of failure detector histories with range $\mathcal{R}_\mathcal{D}$. $\mathcal{D}(F)$ is thus the set of failure detector histories permitted by $\mathcal{D}$ for failure pattern F. Note that we do not make any assumption a priori on the range of a failure detector. When any process p performs a step of computation, it can *query* its *failure detector module* of $\mathcal{D}$, denoted $\mathcal{D}_p$, and obtain a value $d \in \mathcal{R}_\mathcal{D}$ that encodes some information about failures.

If $\mathcal{D}$ and $\mathcal{D}'$ are failure detectors, $(\mathcal{D}, \mathcal{D}')$ denotes the failure detector that outputs a vector with two components, the first being the output of $\mathcal{D}$ and the second being the output of $\mathcal{D}'$. Formally, $\mathcal{R}_{(\mathcal{D},\mathcal{D}')} = \mathcal{R}_\mathcal{D} \times \mathcal{R}_{\mathcal{D}'}$, and for each F, $\tilde{H} \in (\mathcal{D}, \mathcal{D}')(F)$ $\Leftrightarrow$ $\tilde{H} = (H, H')$, $H \in \mathcal{D}(F)$, $H' \in \mathcal{D}'(F)$.

Now we introduce a few popular failure detectors in the model we just described:

—The *perfect* failure detector $\mathcal{P}$ [Chandra and Toueg 1996] outputs a set of *suspected* processes at each process. $\mathcal{P}$ ensures *strong completeness*: every crashed process is eventually suspected by every correct process, and *strong accuracy*: no process is suspected before it crashes.

Formally, $R_\mathcal{P} = 2^\Pi$ and, for each failure pattern F, and each history $H = (p_{i_1}, d_1), (p_{i_2}, d_2), \ldots \in \mathcal{P}(F) \Leftrightarrow$

$$\Big(\exists k \in \mathbb{N}\ \forall p \in F\ \forall k' \geq k :\ p \in d_{k'}\Big) \wedge$$
$$\Big(\forall k \in \mathbb{N}\ ((p \in d_k) \Rightarrow (\forall k' \geq k : p \neq p_{i_{k'}}))\Big)$$

—The *eventually perfect* failure detector $\diamond\mathcal{P}$ [Chandra and Toueg 1996] also outputs a set of suspected processes at each process. But the guarantees provided by $\diamond\mathcal{P}$ are weaker than those of $\mathcal{P}$. There is a time after which $\diamond\mathcal{P}$ outputs the set of all faulty processes at every non-faulty process. More precisely, $\diamond\mathcal{P}$ satisfies strong completeness and *eventual strong accuracy*: there is a time after which no correct process is ever suspected.

Formally, $R_{\diamond\mathcal{P}} = 2^\Pi$ and, for each failure pattern F, and each history $H = (p_{i_1}, d_1), (p_{i_2}, d_2), \ldots \in \diamond\mathcal{P}(F) \quad \Leftrightarrow$

$$\exists k \in \mathbb{N}\ \forall k' \geq k :\ d_{k'} = F$$

—The *leader failure detector* Ω [Chandra et al. 1996] outputs the identifier of a process at each process. There is a time after which it outputs the identifier of the same non-faulty process at all non-faulty processes.
Formally, $R_\Omega = \Pi$ and, for each failure pattern F, and each history $H = (p_{i_1}, d_1), (p_{i_2}, d_2), \ldots \in \Omega(F) \quad \Leftrightarrow$

$$\exists k \in \mathbb{N}\ \exists q \in \Pi \setminus F\ \forall k' \geq k:\ d_{k'} = q$$

—The *quorum failure detector* Σ [Delporte-Gallet et al. 2003] outputs a set of processes at each process. Any two sets (output at any times and at any processes) intersect, and eventually every set consists of only non-faulty processes.
Formally, $R_\Sigma = \Pi$ and, for each failure pattern F, and each history $H = (p_{i_1}, d_1), (p_{i_2}, d_2), \ldots \in \Sigma(F) \quad \Leftrightarrow$

$$\begin{aligned}&\big(\forall k, k' \in \mathbb{N}\ d_k \cap d_{k'} \neq \emptyset\big) \wedge \\ &\big(\exists k \in \mathbb{N}\ \forall k' \geq k\ d_{k'} \subseteq \Pi \setminus F\big).\end{aligned}$$

—The *failure signal* failure detector $\mathcal{FS}$ [Delporte-Gallet et al. 2004], originally called *anonymously perfect failure detector* in [Guerraoui 2002] outputs **green** or **red** at each process. As long as there are no failures, $\mathcal{FS}$ outputs **green** at every process; after a failure occurs, and only if it does, $\mathcal{FS}$ must eventually output **red** permanently at every non-faulty process.
Formally, $R_{\mathcal{FS}} = \{\mathbf{green}, \mathbf{red}\}$ and, for each failure pattern F, and each history $H = (p_{i_1}, d_1), (p_{i_2}, d_2), \ldots \in \mathcal{FS}(F) \quad \Leftrightarrow$

$$\begin{aligned}&\big(\forall k \in \mathbb{N}\ \big(d_k = \mathbf{red} \Rightarrow (\exists p \in F\ \forall k' \geq k:\ p \neq p_{i_{k'}})\big)\big) \wedge \\ &\big(F \neq \emptyset \Rightarrow \exists k \in \mathbb{N}\ \forall k' \geq k\ d_{k'} = \mathbf{red}\big).\end{aligned}$$

Algorithms. The asynchronous communication channels are modeled as a message buffer which contains messages not yet received by their destinations. An *algorithm* $\mathcal{A}$ is a collection of n (possibly infinite state) deterministic automata, one for each process. $\mathcal{A}(p)$ denotes the automaton on which process p is running algorithm $\mathcal{A}$. Computation proceeds in *steps* of $\mathcal{A}$. In each step of $\mathcal{A}$, process p performs atomically the following three actions:

(i) p receives a single message addressed to p from the message buffer, or a null message, denoted λ (*receive* phase);
(ii) p queries and receives a value from its failure detector module (*query* phase);
(iii) p changes its state and sends a message to a single process, according to the automaton $\mathcal{A}(p)$ (*send* phase).

Note that the received message is chosen *non-deterministically* from the messages in the message buffer destined to p, or the null message λ.

Configurations, schedules, and runs. A *configuration* defines the current state of each process in the system and the set of messages currently in the message buffer. Initially, the message buffer is empty. A step (p, m, d) of an algorithm $\mathcal{A}$ is uniquely determined by the identity of the process p that takes the step, the message m received by p during the step (m might be the null message λ), and the failure detector value d seen by p during the step. We assume that messages are uniquely identified.

We say that a step $e = (p, m, d)$ *is applicable* to a configuration C if and only if $m = \lambda$ or m is in the message buffer of C. For a step e applicable to C, $e(C)$ denotes the unique configuration that results from applying e to C.

A *schedule* S of algorithm $\mathcal{A}$ is a (finite or infinite) sequence of steps of $\mathcal{A}$. $S_\perp$ denotes the empty schedule. We say that a schedule S *is applicable to a configuration* C if and only if (a) $S = S_\perp$, or (b) $S[1]$ is applicable to C, $S[2]$ is applicable to $S[1](C)$, etc. For a finite schedule S applicable to C, $S(C)$ denotes the unique configuration that results from applying S to C.

A *partial run of algorithm* $\mathcal{A}$ *in an environment* $\mathcal{E}$ *using a failure detector* $\mathcal{D}$ is a tuple $\langle F, I, S\rangle$ where $F \in \mathcal{E}$, I is an initial configuration of $\mathcal{A}$, $S = (p_{i_1}, m_1, d_1)$, (p_{i_1}, m_1, d_1), …, (p_{i_k}, m_k, d_k) is a finite schedule of $\mathcal{A}$, applicable to I, such that (p_{i_1}, d_1), (p_{i_1}, d_1), …, (p_{i_k}, d_k) is a prefix of a failure detector history $H \in \mathcal{D}(F)$.

A *run of algorithm* $\mathcal{A}$ *in an environment* $\mathcal{E}$ *using a failure detector* $\mathcal{D}$ is a tuple $\langle F, I, S\rangle$ where $F \in \mathcal{E}$, I is an initial configuration of $\mathcal{A}$, and $S = (p_{i_1}, m_1, d_1)$, (p_{i_1}, m_1, d_1), … is an *infinite* schedule of $\mathcal{A}$, applicable to I, such that (p_{i_1}, d_1), (p_{i_1}, d_1), … is a failure detector history in $\mathcal{D}(F)$, and every correct process receives every message sent to it.

Problems and solvability. A *problem* is a predicate on a set of runs (usually defined by a set of properties that these runs should satisfy). An algorithm $\mathcal{A}$ *solves a problem* $\mathcal{M}$ *in an environment* $\mathcal{E}$ *using a failure detector* $\mathcal{D}$ if the set of all runs of $\mathcal{A}$ in $\mathcal{E}$ satisfies $\mathcal{M}$. We say that a failure detector $\mathcal{D}$ *solves problem* $\mathcal{M}$ in $\mathcal{E}$ if there is an algorithm $\mathcal{A}$ which solves $\mathcal{M}$ in $\mathcal{E}$ using $\mathcal{D}$.

Reducibility. Let $\mathcal{D}$ and $\mathcal{D}'$ be failure detectors, and $\mathcal{E}$ be an environment. If, for failure detectors $\mathcal{D}$ and $\mathcal{D}'$, there is an algorithm $T_{\mathcal{D}'\to\mathcal{D}}$ that transforms $\mathcal{D}'$ into $\mathcal{D}$ in $\mathcal{E}$, we say that $\mathcal{D}$ *is weaker than* $\mathcal{D}'$ *in* $\mathcal{E}$.

If $\mathcal{D} \preceq_\mathcal{E} \mathcal{D}'$ but $\mathcal{D}' \not\preceq_\mathcal{E} \mathcal{D}$, we say that $\mathcal{D}$ *is strictly weaker than* $\mathcal{D}'$ *in* $\mathcal{E}$. If $\mathcal{D} \preceq_\mathcal{E} \mathcal{D}'$ and $\mathcal{D}' \preceq_\mathcal{E} \mathcal{D}$, we say that $\mathcal{D}$ *and* $\mathcal{D}'$ *are equivalent in* $\mathcal{E}$. If $\mathcal{D} \not\preceq_\mathcal{E} \mathcal{D}'$ and $\mathcal{D}' \not\preceq_\mathcal{E} \mathcal{D}$, we say that $\mathcal{D}$ *and* $\mathcal{D}'$ *are incomparable in* $\mathcal{E}$.

Algorithm $T_{\mathcal{D}'\to\mathcal{D}}$ that emulates histories of $\mathcal{D}$ using histories of $\mathcal{D}'$ is called a *reduction algorithm.* Note that $T_{\mathcal{D}'\to\mathcal{D}}$ does not need to emulate *all* histories of $\mathcal{D}$; it is required that all the histories it emulates be histories of $\mathcal{D}$.

A weakest failure detector. We say that a failure detector $\mathcal{D}$ is *the weakest failure detector to solve a problem* $\mathcal{M}$ *in an environment* $\mathcal{E}$ if the following conditions are satisfied:

(a) $\mathcal{D}$ is *sufficient* to solve $\mathcal{M}$ in $\mathcal{E}$, i.e., $\mathcal{D}$ solves $\mathcal{M}$ in $\mathcal{E}$, and

(b) $\mathcal{D}$ is *necessary* to solve $\mathcal{M}$ in $\mathcal{E}$, i.e., if a failure detector $\mathcal{D}'$ solves $\mathcal{M}$ in $\mathcal{E}$, then $\mathcal{D}$ is weaker than $\mathcal{D}'$ in $\mathcal{E}$.

There might be a number of distinct failure detectors satisfying these conditions. (Though all such failure detectors are in the just defined sense equivalent.) Hence, it would be more technically correct to talk about *a* weakest failure detector to solve $\mathcal{M}$ in $\mathcal{E}$.

3.2 The weakest failure detector for consensus

In this section, we discuss the seminal "CHT result" obtained by Chandra et al. [1996]. We show that failure detector Ω is necessary for solving consensus in asynchronous message-passing systems in all environments, i.e., for all assumptions on when and where failures might occur. Combined with the algorithm that solves consensus using a failure detector equivalent to Ω given a majority of correct processes [Chandra and Toueg 1996], this result implies that Ω is the weakest failure detector for solving consensus given a majority of correct processes.

3.2.1 *Overview of the reduction algorithm.* Let $\mathcal{E}$ be any environment, $\mathcal{D}$ be any failure detector that can be used to solve consensus in $\mathcal{E}$, and $\mathcal{A}$ be any algorithm that solves consensus in $\mathcal{E}$ using $\mathcal{D}$. Our goal is to determine a reduction algorithm $T_{\mathcal{D}\to\Omega}$ that, using failure detector $\mathcal{D}$ and algorithm $\mathcal{A}$, implements Ω in $\mathcal{E}$. Recall that implementing Ω means outputting, at every process, the identifier of a process so that eventually, the identifier of the same correct process is output permanently at all correct processes.

The basic idea underlying $T_{\mathcal{D}\to\Omega}$ is to have each process locally *simulate* the overall distributed system in which the processes execute several runs of $\mathcal{A}$ that *could have happened* in the current failure pattern and failure detector history. Every process then uses these runs to extract Ω.

In the local simulations, every process p feeds algorithm $\mathcal{A}$ with a set of proposed values, one for each process of the system. Then all automata composing $\mathcal{A}$ are triggered locally by p which emulates, for every simulated run of $\mathcal{A}$, 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, which we call the stimuli of algorithm $\mathcal{A}$, process p periodically queries its failure detector module and exchanges the failure detector information with the other processes.

The reduction algorithm $T_{\mathcal{D}\to\Omega}$ consists of two tasks that are run in parallel at every process: the *commmuncation task* and the *computation task*. In the communication task, every process maintains ever-growing stimuli of algorithm $\mathcal{A}$ by periodically querying its failure detector module and sending the output to all other processes. In the computation task, every process periodically feeds the stimuli to algorithm $\mathcal{A}$, simulates several runs of $\mathcal{A}$, and computes the current emulated output of Ω.

3.2.2 *Building a DAG.* The communication task of algorithm $T_{\mathcal{D}\to\Omega}$ is presented in Figure 4. Executing this task, p knows more and more of the 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) G_p. Informally, every vertex $[q, d, k]$ of G_p is a failure detector value "seen" by q in its k-th query of its failure detector module. An edge $([q, d, k], [q', d', k'])$ can be interpreted as "q saw failure detector value d (in its k-th query) *before* q' saw failure detector value d' (in its k'-th query)".

DAG G_p has some special properties which follow from its construction. Let $F \in \mathcal{E}$ be the set of faulty processes in the current execution. Then:

```
$G_p \leftarrow$ empty graph
$k_p \leftarrow 0$
**while** true **do**
   receive message $m$
   $d_p \leftarrow$ query failure detector $\mathcal{D}$
   $k_p \leftarrow k_p + 1$
   **if** $m$ is of the form $(q, G_q, p)$ **then** $G_p \leftarrow G_p \cup G_q$
   add $[p, d_p, k_p]$ and edges from all vertices of $G_p$ to $[p, d_p, k_p]$ to $G_p$
   send $(p, G_p, q)$ to all $q \in \Pi$
```

Fig. 4. Building a DAG: process p

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

(2) If $v' = [q, d, k]$ and $v'' = [q, d', k']$ are vertices of G_p, and $k < k'$, then (v, v') is an edge of G_p.

(3) G_p is transitively closed: if (v, v') and (v', v'') are edges of G_p, then (v, v'') is also an edge of G_p.

(4) Let $g = [q_1, d_1, k_1] \rightarrow [q_2, d_2, k_2] \rightarrow \ldots$ be any path in G. Then $H = (q_1, d_1) \rightarrow (q_2, d_2) \rightarrow \ldots$ is a partial failure detector history in $\mathcal{D}(F)$.

(5) For all correct processes p and q, and for every vertex v of G_p, there is a $d \in \mathcal{R}_\mathcal{D}$ and a $k \in \mathbb{N}$ such that eventually $(v, [q, d, k])$ is an edge of G_q.

Note that properties (1)–(5) imply that, for every correct process p, $k \in \mathbb{N}$, and set of paths V in G_p, eventually G_p contains a path g, such that (a) every correct process appears at least k times in g, and (b) for every path g' in V, $g' \cdot g$ is also a path in G_p.

3.2.3 *Simulation trees.* Now DAG G_p can be used to simulate runs of $\mathcal{A}$. Any path $g = [q_1, d_1, k_1], [q_2, d_2, k_2], \ldots, [q_s, d_s, k_s]$ through G_p gives the order in which processes $q_1, q_2, \ldots, q_s$ "see", respectively, failure detector values $d_1, d_1, d_2, \ldots, d_s$. That is, g contains an activation schedule and failure detector outputs for the processes to execute steps of $\mathcal{A}$'s instances. Let I be any initial configuration of $\mathcal{A}$. Consider a schedule S that is applicable to I and *compatible with* g, i.e., $|S| = s$ and $\forall k \in \{1, 2, \ldots, s\}$, $S[k] = (q_k, m_k, d_k)$, where m_k is a message addressed to q_k (or the null message λ).

All schedules that are applicable to I and compatible with paths in G_p can be represented as a tree Υ_p^I, called the *simulation tree induced by G_p and I*. The set of vertices of Υ_p^I is the set of all schedules S that are applicable to I and compatible with paths in G_p. The root of Υ_p^I is the empty schedule $S_\perp$. There is an edge from S to S' if and only if $S' = S \cdot e$ for a step e; the edge is labeled e. Thus, every vertex S of Υ_p^I is associated with a sequence of steps $e_1\, e_2\, \ldots\, e_s$ consisting of labels of the edges on the path from $S_\perp$ to S. In addition, every descendant of S in Υ_p^I corresponds to an extension of $e_1\, e_2\, \ldots\, e_s$.

The construction of Υ_p^I implies that, for any vertex S of Υ_p^I, there exists a partial run $\langle F, I, S \rangle$ of $\mathcal{A}$ where F is the current failure pattern. Thus, if correct processes appear sufficiently often in S and receive sufficiently many messages sent to them, then every correct (in F) process decides in $S(I)$.

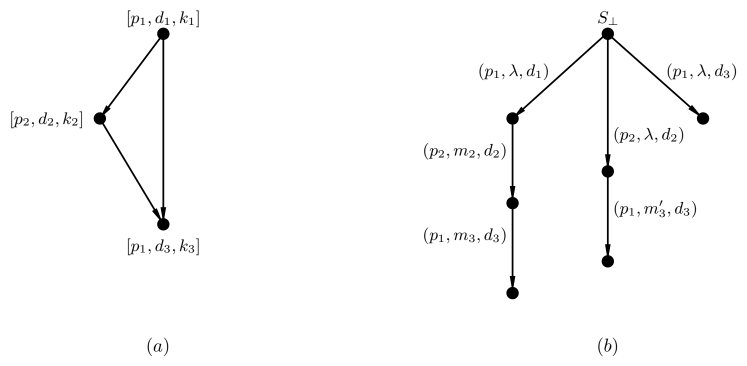

Fig. 5. A DAG and a tree

In the example depicted in Figure 5, a DAG (a) induces a simulation tree a portion of which is shown in (b). There are three non-trivial paths in the DAG: $[p_1, d_1, k_1] \rightarrow [p_2, d_2, k_2] \rightarrow [p_1, d_3, k_3]$, $[p_2, d_2, k_2] \rightarrow [p_1, d_3, k_3]$ and $[p_1, d_1, k_1] \rightarrow [p_1, d_3, k_3]$. Every path through the DAG and an initial configuration I induce at least one schedule in the simulation tree. Hence, the simulation tree has at least three leaves: (p_1, λ, d_1) (p_2, m_2, d_2) (p_1, m_3, d_3), (p_2, λ, d_2) (p_1, m'_3, d_3), and (p_1, λ, d_3). Recall that λ is the empty message: since the message buffer is empty in I, no non-empty message can be received in the first step of any schedule.

3.2.4 *Tags and valences.* Let I^i, $i \in \{0, 1, \ldots, n\}$ denote the initial configuration of $\mathcal{A}$ in which processes $p_1, \ldots, p_i$ propose 1 and the rest (processes $p_{i+1}, \ldots, p_n$) propose 0. In the computation task of the reduction algorithm, every process p maintains an ever-growing *simulation forest* $\Upsilon_p = \{\Upsilon_p^0, \Upsilon_p^1, \ldots, \Upsilon_p^n\}$ where Υ_p^i $(0 \leq i \leq n)$ denotes the simulation trees induced by G_p and initial configurations I^i.

For every vertex of the simulation forest, p assigns a set of *tags*. Vertex S of tree Υ_p^i is assigned a tag v if and only if S has a descendant S' in Υ_p^i such that p decides v in $S'(I^i)$. We call the set tags the *valence* of the vertex. By definition, if S has a descendant with a tag v, then S has tag v. Validity of consensus ensures that the set of tags is a subset of $\{0, 1\}$.

Of course, at a given time, some vertices of the simulation forest Υ_p might not have any tags because the simulation stimuli are not sufficiently long yet. But this is just a matter of time: if p is correct, then every vertex of p's simulation forest will eventually have an extension in which correct processes appear sufficiently often for p to take a decision.

A vertex S of Υ_p^i is 0-*valent* if it has exactly one tag $\{0\}$ (only 0 can be decided in S's extensions in Υ_p^i). A 1-valent vertex is analogously defined. If a vertex S has both tags 0 and 1 (both 0 and 1 can be decided in S's extensions), then we say that S is *bivalent*.[1]

[1]The notion of valence was first defined by Fischer et al. [1985] as the set of values than are decided in *all* extensions of a given execution. Here we define the valence as only a subset of these values, defined by the simulation tree.

It immediately follows from Validity of consensus that the root of Υ_p^0 can at most be 0-valent, and the root of Υ_p^n can at most be 1-valent (the roots of Υ_p^0 and Υ_p^n cannot be bivalent).

3.2.5 *Stabilization.* Note that the simulation trees can only grow with time, they never shrink. As a result, once a vertex of the simulation forest Υ_p gets a tag v, it cannot lose it later. Thus, eventually every vertex of Υ_p stabilizes being 0-valent, 1-valent, or bivalent. Since correct processes keep continuously exchanging the failure detector samples and updating their simulation forests, every simulation tree computed by a correct process at any given time will eventually be a subtree of the simulation forest of every correct process.

Formally, let p be any correct process, i be any index in $\{0, 1, \ldots, n\}$, and S be any vertex of Υ_p^i. Then:

(i) There exists a non-empty $V \subseteq \{0, 1\}$ such that eventually the valence of S is permanently V. (We say that the valence of S *stabilizes* on V at p.)

(ii) If the valence of S stabilizes on V at p, then for every correct process q, eventually S is a vertex of Υ_q^i and the valence of S stabilizes on V at q.

Hence, the correct processes eventually agree on the same tagged simulation subtrees. In discussing the stabilized tagged simulation forest, it is thus convenient to consider the *limit* infinite DAG G and the *limit* infinite simulation forest $\Upsilon = \{\Upsilon^0, \Upsilon^1, \ldots, \Upsilon^n\}$ such that for all $i \in \{0, 1, \ldots, n\}$ and all correct processes p, G_p tends to G and Υ_p^i tends to $Upsilon^i$.

3.2.6 *Critical index.* Let p be any correct process. We say that index $i \in \{1, 2, \ldots, n\}$ is *critical* if *either* the root of Υ^i is bivalent *or* the root of Υ^{i-1} is 0-valent and the root of Υ^i is 1-valent. In the first case, we say that i is *bivalent critical.* In the second case, we say that i is *univalent critical.*

LEMMA 3.1. *There is at least one critical index in* $\{1, 2, \ldots, n\}$.

PROOF. Indeed, by the Validity property of consensus, the root of Υ^0 is 0-valent, and the root of Υ^1 is 1-valent. Thus, there must be an index $i \in \{1, 2, \ldots, n\}$ such that the root of Υ^{i-1} is 0-valent, and Υ^i is either 1-valent or bivalent. □

Since tagged simulation forests computed at the correct processes tend to the same infinite tagged simulation forest, eventually, all correct processes compute the same *smallest* critical index i of the same type (univalent or bivalent). Now we have two cases to consider for the smallest critical index: (1) i is univalent critical, or (2) i is bivalent critical.

3.2.7 *Handling a univalent critical index*

LEMMA 3.2. *If i is univalent critical, then p_i is correct.*

PROOF. By contradiction, assume that p_i is faulty. Then G contains an infinite path g in which p_i does not participate and every correct process participates infinitely often. Then Υ^i contains a vertex S such that p_i does not take steps in S and some correct process p decides in $S(I^i)$. Since i is 1-valent, p decides 1 in $S(I^i)$. But p_i is the only process that has different states in I^{i-1} and I^i, and

p_i does not take part in S. Thus, S is also a vertex of Υ^{i-1} and p decides 1 in $S(I^{i-1})$. But the root of Υ^{i-1} is 0-valent — a contradiction. □

3.2.8 *Handling a bivalent critical index.* Assume now that the root of Υ^i is *bivalent*. Below we show that Υ^i then contains a *decision gadget*, i.e., a finite subtree which is either a *fork* or a *hook* (Figure 6).

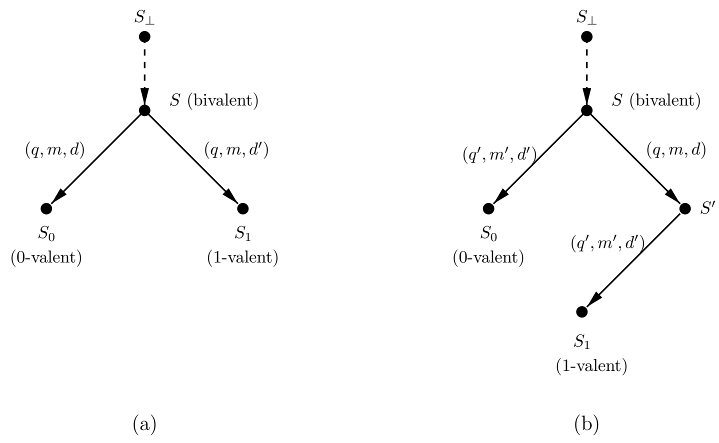

Fig. 6. A fork and a hook

A fork (case (a) in Figure 6) consists of a bivalent vertex S from which two *different* steps by the *same* process q, consuming the same message m, are possible which lead, on the one hand, to a 0-valent vertex S_0 and, on the other hand, to a 1-valent vertex S_1.

A hook (case (b) in Figure 6) consists of a bivalent vertex S, a vertex S' which is reached by executing a step of some process q, and two vertices S_0 and S_1 reached by applying *the same* step of process q' to, respectively, S and S'. Additionally, S_0 must be 0-valent and S_1 must be 1-valent (or vice versa; the order does not matter here).

In both cases, we say that q is the *deciding process*, and S is the *pivot* of the decision gadget.

Lemma 3.3. *The deciding process of a decision gadget is correct.*

Proof. Consider any decision gadget γ defined by a pivot S, vertices S_0 and S_1 of opposite valence and a deciding process q. By contradiction, assume that q is faulty. Let g, g_0 and g_1 be the simulation stimuli of, respectively, S, S_0 and S_1. Then G contains an infinite path $\tilde{g}$ such that (a) $g \cdot \tilde{g}$, $g_0 \cdot \tilde{g}$, $g_1 \cdot \tilde{g}$ are paths in G, and (b) q does not appear and the correct processes appear infinitely often in g.

Let γ be a fork (case (a) in Figure 6). Then there is a finite schedule $\tilde{S}$ compatible with a prefix of $\tilde{g}$ and applicable to $S(I^i)$ such that some correct process p decides in $S \cdot \tilde{S}(I^i)$; without loss of generality, assume that p decides 0. Since q is the only process that can distinguish $S(I^i)$ and $S_1(I^i)$, and q does not appear in $\tilde{S}$, $\tilde{S}$ is also applicable to $S_1(I^i)$. Since $g_1 \cdot \tilde{g}$ is a path of G and $\tilde{S}$ is compatible with a prefix of $\tilde{g}$, it follows that $S_1 \cdot \tilde{S}$ is a vertex of Υ^i. Hence, p also decides 0 in $S_1 \cdot \tilde{S}(I^i)$. But S_1 is 1-valent — a contradiction.

Let γ be a hook (case (b) in Figure 6). Then there is a finite schedule $\tilde{S}$ compatible with a prefix of g and applicable to $S_0(I^i)$ such that some correct process p decides in $S_0 \cdot \tilde{S}(I^i)$. Without loss of generality, assume that S_0 is 0-valent, and hence p decides 0 in $S_0 \cdot \tilde{S}(I^i)$. Since q is the only process that can distinguish $S_0(I^i)$ and $S_1(I^i)$, and q does not appear in $\tilde{S}$, $\tilde{S}$ is also applicable to $S_1(I^i)$. Since $g_1 \cdot \tilde{g}$ is a path of G and $\tilde{S}$ is compatible with a prefix of $\tilde{g}$, it follows that $S_1 \cdot \tilde{S}$ is a vertex of Υ^i. Hence, p also decides 0 in $S_1 \cdot \tilde{S}(I^i)$. But S_1 is 1-valent — a contradiction. □

Now we need to show that any bivalent simulation tree Υ^i contains at least one decision gadget γ.

Lemma 3.4. *If i is bivalent critical, then Υ^i contains a decision gadget.*

Proof. Let i be a bivalent critical index. In Figure 7, we present a procedure which goes through Υ^i. The algorithm starts from the bivalent root of Υ^i and terminates when a hook or a fork has been found.

```
$S \leftarrow S_\perp$
while *true* do
  $p \leftarrow$ ⟨choose the next correct process in a round robin fashion⟩
  $m \leftarrow$ ⟨choose the oldest undelivered message addressed to $p$ in $S(I^i)$⟩
  if ⟨$S$ has a descendant $S'$ in $\Upsilon^i$ (possibly $S = S'$) such that, for some $d$,
        $S' \cdot (p, m, d)$ is a bivalent vertex of $\Upsilon^i$⟩
  then $S \leftarrow S' \cdot (p, m, d)$
  else exit
```

Fig. 7. Locating a decision gadget

We show that the algorithm indeed terminates. Suppose not. Then the algorithm locates an infinite *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. But no correct process can decide in a bivalent state $S(I^i)$ (otherwise we would violate the Agreement property of consensus). As a result, we constructed a run of $\mathcal{A}$ in which no correct process ever decides — a contradiction.

Thus, the algorithm in Figure 7 terminates. That is, there exists a bivalent vertex S, a correct process p, and a message m addressed to p in $S(I^i)$ such that

$$\begin{array}{c}\text{For all descendants } S' \text{ of } S \text{ (including } S' = S) \text{ and all } d, \\ S' \cdot (p, m, d) \text{ is } \textit{not} \text{ a bivalent vertex of } \Upsilon^i.\end{array} \tag{1}$$

In other words, any step of p consuming message m brings any descendant of S (including S itself) to either a 1-valent or a 0-valent state. Without loss of generality, assume that, for some d, $S \cdot (p, m, d)$ is a 0-valent vertex of Υ^i. Since S is bivalent, it must have a 1-valent descendant S''.

If S'' includes a step in which p consumes m, then we define S' as the vertex of Υ^i such that, for some d', $S' \cdot (p, m, d')$ is a prefix of S''. If S'' includes no step in which p consumes m, then we define $S' = S''$. Since p is correct, for some d', $S' \cdot (p, m, d')$ is a vertex of Υ^i. In both cases, we obtain S' such that for some d', $S' \cdot (p, m, d')$ is a 1-valent vertex of Υ^i.

Let the path from S to S' go through the vertices $\sigma_0 = S, \sigma_1, \ldots, \sigma_{m-1}, \sigma_m = S'$. By transitivity of G, for all $k \in \{0, 1, \ldots, m\}$, $\sigma_k \cdot (p, m, d')$ is a vertex of Υ^i. By statement 2, $\sigma_k \cdot (p, m, d')$ is either 0-valent or 1-valent vertex of Υ^i.

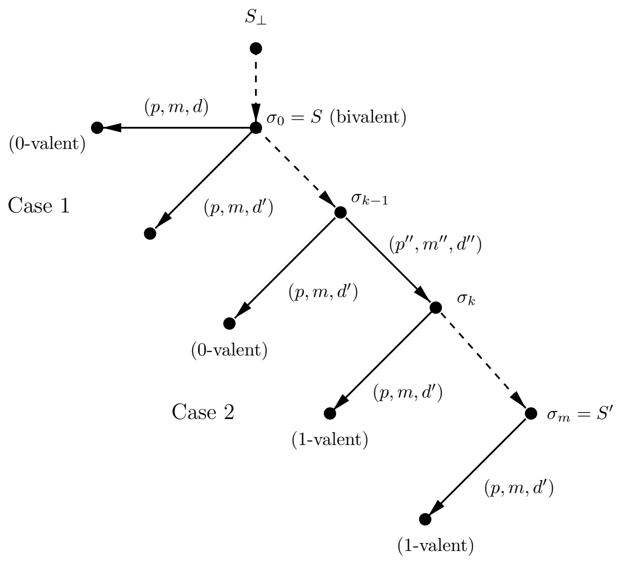

Fig. 8. Locating a fork (Case 1) or a hook (Case 2)

Let $k \in \{0, \ldots, m\}$ be the lowest index such that (p, m, d') brings σ_k to a 1-valent state. We know that such an index exists, since $\sigma_m \cdot (p, m, d')$ 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) $k > 0$.

Assume that $k = 0$, i.e., (p, m, d') applied to S brings it to a 1-valent state. But we know that there is a step (p, m, d) that brings S to a 0-valent state (Case 1 in Figure 8). That is, a fork is located!

If $k > 0$, we have the following situation. Step (p, m, d') brings σ_{k-1} to a 0-valent state, and $\sigma_k = \sigma_{k-1} \cdot (p', m', d'')$ to a 1-valent state (Case 2 in Figure 8). But that is a hook!

As a result, any bivalent infinite simulation tree has at least one decision gadget. $\square$

3.2.9 *The reduction algorithm.* Now we are ready to complete the description of $T_{\mathcal{D} \to \Omega}$. In the computation task (Figure 9), every process p periodically extracts the current *leader* from its simulation forest, so that eventually the correct processes agree on the same correct leader. The current leader is stored in variable Ω-$output_p$.

```
Initially:
  for $i = 0, 1, \ldots, n$: $\Upsilon_p^i \leftarrow$ empty graph
  $\Omega$-$output_p \leftarrow p$

while true do

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

  { Select a process from the tagged simulation forest }
  if there is a critical index then
    $i \leftarrow$ the smallest critical index
    if $i$ is univalent critical then $\Omega$-$output_p \leftarrow p_i$
    if $\Upsilon_p^i$ has a decision gadget then
      $\Omega$-$output_p \leftarrow$ the deciding process of the smallest decision gadget in $\Upsilon_p^i$
```

Fig. 9. Extracting a correct leader: code for each process p

Initially, p elects itself as a leader. Periodically, p updates its simulation forest Υ_p by incorporating more simulation stimuli from G_p. If the forest has a univalent critical index i, then p outputs p_i as the current leader estimate. If the forest has a bivalent critical index i and Υ_p^i contains a decision gadget, then p outputs the deciding process of *the smallest* decision gadget in Υ_p^i (the "smallest" can be well-defined, since the vertices of the simulation tree are countable).

Eventually, the correct processes locate the same *stable* critical index i. Now we have two cases to consider:

(i) i is univalent critical. By Lemma 3.2, p_i is correct.

(ii) i is bivalent critical. By Lemma 3.4, the limit simulation tree Υ^i contains a decision gadget. Eventually, the correct processes locate the same decision gadget γ in Υ_i and compute the deciding process q of γ. By Lemma 3.3, q is correct.

Thus, eventually, the correct processes elect the same correct leader — Ω is emulated!

3.3 The weakest failure detector for a register

In this section, we show that the quorum failure detector Σ is the weakest failure detector to implement atomic registers in all environments. Combined with earlier work on state machine replication [Lamport 1978; Schneider 1990], this implies that (Ω, Σ), the composition of Ω and Σ, is the weakest failure detector for solving consensus in *all* environments, i.e., for all assumptions on when and where failures might occur..

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.

3.3.1 *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; 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: all these registers are computationally equivalent.

3.3.2 *The sufficiency part.* Recall that the quorum failure detector Σ outputs a set of processes at each process. Any two sets (output at any times and by any processes) intersect, and eventually every set consists of only correct processes.

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 in any environment using Σ. 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.

3.3.3 *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 $\mathcal{E}$ be any environment. Let $\mathcal{D}$ be any failure detector that can be used to implement in $\mathcal{E}$ 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

```
Initially:
    $\Sigma$-$output_p \leftarrow \Pi$ { $\Sigma$-$output_p$ is the output of $p$'s module of $\Sigma$ }
    $I \leftarrow$ the initial configuration of $\mathcal{A}$

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 $G_p$
        $\mathcal{S} \leftarrow$ set of all schedules of $\mathcal{A}$ compatible with some path
            of $G_p(u)$ and applicable to $I$
    until there is a schedule $S \in \mathcal{S}$ of a complete $p$-solo run of $\mathcal{A}$
    $\Sigma$-$output_p \leftarrow$ set of all processes that take steps in the schedule $S$
```

Fig. 10. Extracting Σ: code for each process p

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 $\mathcal{D}$.

Again, 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 these samples in an ever-increasing directed acyclic graph G_p, which is organized exactly as in Section 3.2 (Figure 4). 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 10.

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 such that there is a schedule S of a complete p-solo run of $\mathcal{A}$, compatible with g. The set of processes that take steps in S 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 schedules of $\mathcal{A}$ simulated by p.

(2) For all p and q, Σ-$output_p$ and Σ-$output_q$ always 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 10, $\mathcal{A}$ has a complete p-solo run $R_p = \langle F, I, S_p \rangle$

and a complete q-solo run $R_q = \langle F, I, S_q \rangle$ such that the sets of processes that participate in S_p and S_q are disjoint.

But the two runs R_p and R_q can be composed in a single run $R = \langle F, I, S \rangle$ that is indistinguishable from R_p to p, and indistinguishable from R_q to q.

Hence, both p and q complete $\mathcal{A}$ in R. 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, for all environments $\mathcal{E}$, Σ is the weakest failure detector to implement atomic registers in $\mathcal{E}$.

3.3.4 *Solving consensus in all environments.* Once we determined the weakest failure detector to implement atomic registers, determining the weakest failure detector for solving consensus in all environments is straightforward. This failure detector is (Ω, Σ), the composition of Ω and Σ.

Indeed, failure detector (Ω, Σ) can be used to solve consensus in all environments, 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 in any environment [Lamport 1978; Schneider 1990], and thus to extract Σ. Combined with the fact that Ω is necessary to solve consensus in *any* environment [Chandra et al. 1996] (see Section 3.2), this implies that (Ω, Σ) is necessary to solve consensus in any environment.

3.4 Solving Non-Blocking Atomic Commit

In this section, we discuss determining the weakest failure detector for solving Non-Blocking Atomic Commit (NBAC) in all environments. 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 2001; Guerraoui 2002].

3.4.1 *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).

3.4.2 *Using* $(\Psi, \mathcal{FS})$ *to solve NBAC.* The algorithm in Figure 11 uses $(\Psi, \mathcal{FS})$ to solve NBAC in any environment $\mathcal{E}$. 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

```
NON-BLOCKING_ATOMIC_COMMIT($v$): { $v$ is yes or no }
send $v$ to all
wait until [(for each process $q$ in $\Pi$, received $q$'s vote) or $\mathcal{FS}_p$ = red]
if the votes of all processes are received and are yes then
   myproposal $\leftarrow 1$
else { some vote was no or there was a failure }
   myproposal $\leftarrow 0$
wait until [$\Psi_p \neq \perp$]
if $\Psi_p$ = red
   then { henceforth $\Psi$ behaves like $\mathcal{FS}$ }
      return abort
   else { henceforth $\Psi$ behaves like $(\Omega, \Sigma)$ }
      mydecision $\leftarrow$ CONSPROPOSE($v$) { use $\Psi$ to run $(\Omega, \Sigma)$-based consensus algorithm }
if mydecision = 1 then
   return commit
else return abort
```

Fig. 11. Using $(\Psi, \mathcal{FS})$ to solveNBAC: code for each process p

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 is shown in [Delporte-Gallet et al. 2003] that there is an algorithm that uses (Ω, Σ) to solve consensus in any environment (see also Sect. 3.3). Therefore, in this case, p proposes **myproposal** to that consensus algorithm and returns the value decided by that algorithm. If 1 is 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.

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

There is a straightforward reduction algorithm that transforms $\mathcal{D}$ into $\mathcal{FS}$ [Charron-Bost and Toueg 2001; 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.

3.5 Summary

Failure detectors are not only a helpful engineering abstraction but also allow to compare the synchrony requirements of problems in fault-tolerant computing. In this section we discussed the associated problem of the *weakest* failure detector and presented three examples of weakest failure detector proofs.

4. LIMITATIONS OF FAILURE DETECTORS

The failure detector abstraction has many practical and theoretical virtues, but we do not want to close this survey without also investigating the limitations of this abstraction which have been frequent sources of misunderstandings and misconceptions in fault-tolerant algorithms. We have grouped the discussion about the limitations around four basic questions which we discuss and put into context.

4.1 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 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.

Gärtner and Pleisch [2002] 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 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 which can be defined as the result of the causal structure of a computation can be computed. As another result, they 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). So at least a perfect failure detector can be regarded as an abstraction of process synchrony, not of channel synchrony.

4.2 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 many other failure models in the literature. Among those 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 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 which 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, consider the *crash-recovery* failure model where processes can crash and later start execution again from a predefined point in their program. Since the failure model offers a new type of behavior, 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 which 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].

Fixing the crash model, 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].

The failure detector abstraction cannot easily be adapted to Byzantine failures since it is not possible to derive a clean failure detector interface which 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. The closest we can get are so-called "muteness" detectors [Doudou et al. 1999; Doudou et al. 2002]. But this approach usually assumes that processes send certain messages continuously and relay messages to all others if they receive them for the first time. Kihlstrom et al. [2003] distinguishes between detectable and undetectable Byzantine failures. Non-detectable failures are either unobservable (a Byzantine process spontaneously changes his input value) or undiagnosable (they cannot be tagged to a specific process, e.g. if a process claims that some other process sent him something). Byzantine failure detectors can only report detectable faults, which can be further classified into commisson 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].

4.3 Can Randomization be used to implement Failure Detectors?

In 1983, Ben-Or [1983] presented an algorithm which 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 seperately 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 [2005] studied the relationship between fairness and randomization. He showed that to solve crash-tolerant consensus it is sufficient to postulate a certain kind of fairness called *hyperfairness*. Briefly spoken, hyperfairness looks at situations in which certain resources are needed (e.g., for a process to terminate) but which can be independently made available and withdrawn to a process. Hyperfairness means that eventually these resources will be "synchronized" and available to that process. This special type of fairness can be implemented using randomization and partial synchrony.

4.4 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 in terms of explicit time failure detectors do offer.

Strong failure detectors have 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 which 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 which 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 guarantees 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 which detect failures in a time which is orders of magnitude less than a round trip delay [Aguilera et al. 2002].

5. SUMMARY

Take the time-free system model which is usually used when reasoning about fault-intolerant distributed algorithms, add the concept of unreliable failure detectors, and you get a system model which can be used to reason about fault-tolerant distributed algorithms. The failure detector abstraction has many virtues as an engineering tool and as a computability benchmark, but also has some limitations in expressiveness.

REFERENCES

Aguilera, Delporte-Gallet, Fauconnier, and Toueg. 2001. Stable leader election. In *DISC: International Symposium on Distributed Computing*. LNCS.

Aguilera, Delporte-Gallet, Fauconnier, and Toueg. 2003. On implementing omega with weak reliability and synchrony assumptions. In *PODC: 22th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*.

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., 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 16th International Symposium on Distributed Computing (DISC'02)*. 354–370.

Arora, A. and Kulkarni, S. S. 1998. Detectors and correctors: A theory of fault-tolerance components. In *Proceedings of the 18th IEEE International Conference on Distributed Computing Systems (ICDCS98)*.

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. 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.

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 Toueg, S. 2001. Unpublished notes.

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 *PODC*. 338–346.

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., FRIEDMANN, R., KEIDAR, I., AND MALKHI, D. 1997. Failure detectors in omission failure environments. In *Proceedings of the 16th Annual ACM Symposium on Principles of Distributed Computing (PODC97)*.

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., GUERRAOUI, R., AND SCHIPER, A. 1999. Muteness failure detectors: specification and implementation. In *Proc. 3rd European Dependable Computing Conference*. 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.

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.

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 *Proc. 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. 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.

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.

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.

Jean-François Hermant and Josef Widder, title = Implementing Reliable Distributed Real-Time Systems with the Theta-Model, O. . . O. . . b. . I. O. . . y. . . O. . . O. . . O. . . O. . . O. . . m. . d. O. . . O. . . O. . . O. . .

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., 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 Arv́alo, 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.

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.

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.

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

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.

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. 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.