A Gentle Introduction to Failure Detectors and Related Subjects

Felix C. Gärtner
Darmstadt University of Technology

Department of Computer Science
Technical Report TUD-BS-2001-01

April 19, 2001

The concept of *unreliable failure detectors* was introduced into the field of computer science by Chandra and Toueg in 1991 [18, 19]. Failure detectors are abstract devices that offer information about the functional state of processes in a distributed system. There is much evidence that failure detectors play a vital role in the design and analysis of fault-tolerant algorithms and systems. Unfortunately, the literature on this subject is rather diverse and mostly very technical. Rather than striving for new results about failure detectors, this text tries to offer a more gentle introduction into the field by explaining the underlying ideas of the concept using as little formalism as possible. It is aimed at people who are new to the concept, want to quickly understand the basic principles and get started in the area.

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: asynchronous system, agreement problem, consensus problem, fault tolerance, liveness, message passing, safety, synchronous system

---

This work was supported by the Deutsche Forschungsgemeinschaft (DFG) as part of the "Graduiertenkolleg ISIA" at Darmstadt University of Technology.
Address: Department of Computer Science, Alexanderstr. 10, D-64283 Darmstadt, Germany, Email: `fcg@acm.org`

A Gentle Introduction to Failure Detectors and Related Subjects

Felix Gärtner

Abstract. The concept of *unreliable failure detectors* was introduced into the field of computer science by Chandra and Toueg in 1991 [18, 19]. Failure detectors are abstract devices that offer information about the functional state of processes in a distributed system. There is much evidence that failure detectors play a vital role in the design and analysis of fault-tolerant algorithms and systems. Unfortunately, the literature on this subject is rather diverse and mostly very technical. Rather than striving for new results about failure detectors, this text tries to offer a more gentle introduction into the field by explaining the underlying ideas of the concept using as little formalism as possible. It is aimed at people who are new to the concept, want to quickly understand the basic principles and get started in the area.

1 Introduction

Fault tolerance in distributed systems is an interesting area of research because, firstly, it poses a lot of difficult theoretical questions, and secondly, it is an area of eminent practical importance. Generally, fault tolerance means "correctness in the presence of faults," and correctness arguments usually involve (formal) reasoning about the system model which is assumed. The choice of the right (i.e., a powerfull but yet tractable) model for certain application areas is crucial to the design and analysis of algorithms.

In the literature on "normal" (i.e., non-fault-tolerant) distributed algorithms you usually encounter a system model which is often termed the *time-free* message passing model. In this model, processes run local algorithms and synchronize each other exclusively by sending messages through communication channels. The model is so popular because of its simplicity: it neither makes assumptions about the processing speeds of the processes nor does it restrict the message transmission time. Because of these very weak assumptions, algorithms designed for the time-free model will also work correctly in environments where specific timing assumptions exist (e.g., a maximum message transmission delay).

However, the time-free model is not a good model for studying fault-tolerant distributed systems. This is because in the time-free model many important problems of fault-tolerant computing are not solvable if processes are allowed to fail in simple but realistic ways. This is usually attributed to the fact that it is impossible for processes to derive information about such failures of other processes in the time-free system model. To circumvent these impossibilities, Chandra and Toueg suggested to augment the time-free model by adding *unreliable failure detectors* [19]. Basicly, failure detectors are abstract devices that offer information about the functional state of processes in a distributed system. Using failure detectors it is possible to solve those problems which were originally impossible to solve.

There is much evidence that failure detectors play a vital role in the design and analysis of fault-tolerant algorithms and systems. Unfortunately, the literature on this subject is rather diverse and mostly very technical. This text aims at people who want to obtain a general understanding of the field without having to read the rather technical literature which already exists. It tries to focus on the underlying ideas of the concept using as little formalism as possible. We assume that readers have a basic knowledge of distributed systems and algorithms, i.e., we will not explain terms like safety, liveness, problem specification, trace, etc. Wherever we felt it to be appropriate, remarks that are more technical and give hints to the literature have been inserted. These remarks can be skipped at first reading.

1.1 Remark (What is not covered) This is the first such technical remark: Apart from failure detection there exist several other means to circumvent the impossibility, e.g., randomization [6, 14, 24] or refined notions of fairness [69]. To streamline the presentation we will only be covering deterministic algorithms and systems and only briefly touch these mechanisms at the end of this article. (End of Remark)

While this text touches many of the issues of the area, it should however be noted that it is by no means a replacement of the excellent original work in the area. While being rather technical and formal, many of the original articles on failure detection are very well-readable and naturally are the best source of information on the subject matter.

Overview. We will first start by explaining the basic assumptions of the time-free model and show in more detail why it is not a good model for fault tolerance (Section 2). Then we will show, on the one hand, how these problems can be solved by introducing an explicit notion of real-time, and, on the other hand, that introducing explicit time has its drawbacks (Section 3). This will lead the way to the main section on failure detectors (Section 4) and related aspects. A final Section on alternative system models and a discussion concludes this article.

2 What is the problem with the time-free model?

2.1 Basic (time-free) model

The basic time-free model can be characterized by the following basic statements:

- A system is modeled as a set of processes connected by communication channels.
- Usually it is assumed that the network is fully connected, i.e., every process can directly send messages to every other process.
- Communication is by point-to-point message passing using *send* and *receive* primitives.
- 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.

The attribute "time-free" mainly refers to the last item. It results 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_i to take a single step, another process p_j can take any finite number of steps.

2.1 Remark The term "time-free" refering to the above type of system model is not yet a standard one. While it has been used before, the first written reference to the term which we are aware of is a paper by Cristian and Fetzer [25]. Previously, the system model has also been called the *asynchronous* system model [63]. (End of Remark)

Usually, all components of the system are assumed to be reliable. In the context of fault tolerance, the time-free model is combined with an assumption about how the individual parts of the system are allowed to fail. This is the *failure model* (sometimes also called *failure assumption*, *failure semantics* or *failure mode*).

There are different common types of failure models. One of the most prominent ones is the *crash* failure model. It assumes that processes fail by halting, i.e., by simply stopping to execute steps. Sometimes the failure model also assumes an upper bound on the number of processes which are allowed to crash. Those processes which eventually crash are called *faulty*; the non-faulty processes are called *correct*.

2.2 Remark (Other failure models) There are many other different failure models in the literature. Among those that refer to the incorrect behavior of processes are *fail-stop* [62], *crash-recovery* [1, 49, 56], *send/receive omission* [46, 47] and *Byzantine* [51]. 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 (at least in terms of internal actions and what type of messages it sends; note that in the Byzantine failure model introduced by Lamport et al. [51] processes are still required to take steps and send messages in every round of an algorithm, whereas the Byzantine model in the understanding of Hadzilacos and Toueg [47] includes the crash failure model). (End of Remark)

2.3 Remark (channel failures and classifications) Failure models may also refer to channels and allow them to be unreliable, meaning that messages may be lost, duplicated, altered, or that spurious messages may appear within the channel out of nowhere [12]. Classifications of failure models as well as mechanisms to describe the relations between them have been presented by Hadzilacos and Toueg [47] and by Echtle and Masum [34]. (End of Remark)

2.2 The consensus problem

Most of the fundamental problems of interest in fault-tolerant computing are *agreement problems*. They always appear when a set of processes must make a consistent decision. For example, if a database is replicated (for sake of performance or fault tolerance), all replicas have to agree whether to commit or abort some given transaction. Many different agreement problems have been condensed into one basic problem called *consensus*.

The consensus problem is defined using two primitives called *propose* and *decide*. Both take an argument from a fixed set of decision values. 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 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). Note that Agreement and Validity are safety properties and that Termination is a liveness property.

2.4 Remark The above definition of consensus is general in the sense that it is not tied to a specific failure model. Given a specific failure model the specification usually must be weakened to still be solvable. To see this, consider the crash failure model and the above consensus specification. The specification requires that every process eventually decides. However, if some process crashes before it decides, the Termination condition can never be met. A weaker variant which is usually used is the following:

- Termination: Every *correct* process eventually decides.

The resulting consensus specification is called *uniform consensus* [47] because it mandates that all processes (i.e., even the faulty ones) do not disagree on the decision value. (End of Remark)

2.5 Remark An even weaker variant of consensus may only restrict the agreement to the set of correct processes. However, since it is difficult to know beforehand which processes will not crash in an execution, it is important to precisely define what this means. For example, Charron-Bost, Toueg and Basu [22] have observed that unfortunate formalizations can turn the agreement property into a liveness property: assuming two nodes have decided differently, agreement can still be satisfied if one of both nodes crashes. (End of Remark)

2.6 Remark (Literature on consensus) A near-to-complete survey of literature on the consensus problem is a paper by Barborak et al. [11]. Another good reference on different variants of consensus and other agreement problems is the book by Lynch [54], while Guerraoui and Schiper [44] address common prejudices and misconceptions about the consensus problem. (End of Remark)

2.3 Why is consensus impossible?

In general, only system models where it is possible to solve consensus are really useful for fault tolerance. Unfortunately, the time-free model together with the crash failure model does not allow any solution to consensus. To approach the intuition of this fact, consider the following example.

2.7 Example (The consensus dilemma) Consider a time-free distributed system consisting of two processes p_1 and p_2 connected by a bidirectional communication channel. The channel is reliable but processes may fail by crashing, i.e., at any point within their execution they may simply stop to execute steps of their local algorithm.

Assume that at some point in the execution of a consensus algorithm, one process, say p_1, is waiting for a message m from p_2. Message m contains some vital information needed by p_1 before it can proceed in its execution. As long as m has not arrived, p_1 is in a dilemma: How long should p_1 continue to wait for m? It has two choices:

1. It can continue to wait for m.
2. At some point in its execution, p_1 may become impatient and assume that message m will never arrive because p_2 crashed before sending it.

Always choosing the first alternative is not a good idea if p_2 has actually crashed, because if p_2 has crashed before sending m, then p_1 may wait infinitely long. Thus, a computation in which p_1 is involved may never terminate.

Choosing the second alternative is okay as long as the assumption about p_2 is correct. For example, p_1 could proceed to decide the value which it has previously proposed. However, it is obvious that the same argument can be restated in a symmetric way taking p_2 as the process which is waiting for a vital message from p_1. Again, p_2 is in the same dilemma and may also erroneously assume that p_1 has crashed. Thus, p_2 will also decide some value, but according to Murphy's law, this value will probably be different from the one which p_1 has decided. Consequently, taking this alternative may result in executions where processes do not agree. (End of Example)

Intuitively, problems which require all processes to participate in a non-trivial way (such as consensus) are impossible in the time-free model where process crashes can happen [8, 15]. The example however also indicates that one way out of the dilemma is to allow processes to distinguish between situations where other processes have crashed or are merely very slow.

2.8 Remark (Proof of impossibility) The original formal impossibility proof appears in a widely cited paper by Fischer et al. [37]. The intuition behind the impossibility is discussed by Dolev et al. [28], while Tel [66] and Turek and Shasha [67] rephrase the main lines of the proof with a little less technical detail. (End of Remark)

3 Introducing explicit synchrony

One obvious way to try and make consensus solvable in the time-free system is to abandon the attribute "time-free" and explicitly strengthen the assumptions about the synchrony of processors and communication channels.

3.1 Fully synchronous systems

So let us drop the asynchrony assumption of the time-free model and assume that there exist known bounds on the time it takes for a message to travel from one process to the other and on the relative processing speeds. We will denote these bounds by δ (processing speed bound) and Δ (message delivery delay bound). They 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.

3.1 Remark (Relations to real-time) The beforementioned synchrony conditions can also be 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 [25]. That local clocks advance at a steady rate does not mean that processes advance equally within their local algorithms. (End of Remark)

If bounds δ and Δ exist, then it is obviously easy to detect the crash of some process.

3.2 Example (Detecting a process crash using δ and Δ) Take two processes p_1 and p_2 again and consider the protocol depicted in Figure 1. Process p_1 sends a message requesting an immediate reply to p_2. Now p_1 waits for $2\Delta + \delta$ steps. This is exactly the number of steps after which p_2 must have received, processed and returned the message to p_1. If an answer does not arrive within this bound, then p_2 must have crashed. (End of Example)

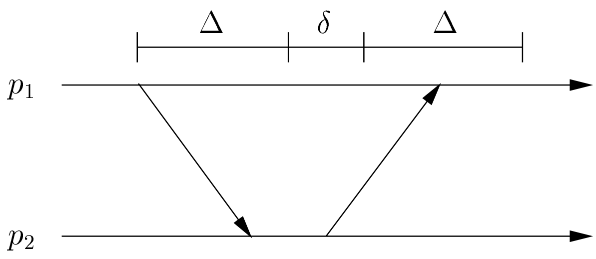

Figure 1: Detecting process crashes in synchronous systems.

Using similar protocols it is possible to build consensus algorithms where all process may crash [28]. Note that the execution of a consensus algorithm in which all processes crash can still be a valid execution regarding the problem specification of consensus (see Section 2.2).

3.3 Remark (Relations to real-time) If the bounds are expressed in terms of real time (see Remark 3.1) then the detection mechanism can be implemented using a real-time timer. This is the way in which much of the failure detection in practice is done [68]. Note that there exist a variety of failure detection algorithms [65] that differ substantially in the quality of service they deliver [23]. (End of Remark)

3.4 Remark If receiving and sending a message are two separate atomic steps, then both bounds δ and Δ are necessary. Dolev et al. [28] have shown that one or both bounds can be omitted if other system parameters are favorable: For example, if it is possible to broadcast a message to all processes in one atomic step and if messages are delivered in the real-time order in which they were sent, then consensus is solvable without the bounds on explicit synchrony. (End of Remark)

3.2 Partially synchronous systems

The assumptions on explicit timing bounds is rather strong, so people have explored ways to weaken these bounds and still have a model which is powerful enough to solve consensus in. This resulted in the concept of *partial synchrony* [33].

There are two possible variants of partial synchrony, which we will exemplify using the communication bound Δ:

1. Either Δ exists but is not known, or
2. Δ is known but holds only eventually.

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. When modeling practical systems, it is usually safe to assume that some upper bound on message delivery time exists. The question of how large it actually is motivates the first form of partial synchrony. Equally, it is a common observation that periods of instability of a system (i.e., periods in which timing parameters are not met) are usually followed by "long" periods of stability. This motivates the second form.

Interestingly, both forms of partial synchrony allow consensus to be solved, even if both communication and processes are partially synchronous. Why is this so?

Let us start with the first form of partially synchronous communication, i.e., where an upper bound Δ exists but is not known. The idea now is to start with an educated guess Δ' of Δ. Obviously, a wrong guess will lead to false suspicions, i.e., a process p_1 could detect the crash of another process p_2 without p_2 having actually crashed. This is not good since it may lead to a violation of safety as discussed in Example 2.7. However, it is possible to construct algorithms in which such wrong suspicions merely affect liveness, not safety, as we now explain.

3.5 Example (A consensus algorithm which is always safe) Consider an asynchronous system with three processes p_1, p_2 and p_3 and at most one process may crash. The algorithm proceeds in asynchronous rounds, each of which is subdivided into four phases. At any time, every process has an *estimate* of the value it thinks will become the decision value. Initially, this estimate is the proposed value of every process. Figure 2 shows the pseudocode of the algorithm.

Every message is tagged with the round number of the process when it is sent. Thus, messages from different rounds can be distinguished. In every round, one process plays a distinct *coordinator* role: In round 1 this is process p_1, in round 2 it is p_2 and so on in round robin fashion. Here is what happens in the phases of the first round:

- In phase 1, every process sends its current estimate to the coordinator p_1.

```
Process p_i:

  variables:
    estimate = ⟨proposed value⟩
    round = 1                                  {* current round number *}
    coord = 1                                  {* current coordinator *}
    update = 1      {* last round number in which estimate was updated *}

  algorithm:
    while ⟨not yet decided⟩
                                                          {* phase 1: *}
      send (estimate, round, update) to p_coord
                                                          {* phase 2: *}
      if i = coord then
        receive (e, r, u) from ⟨a non-coordinator⟩
        estimate := ⟨either estimate or e depending on update and u⟩
        send (estimate, round) to all
                                                          {* phase 3: *}
      wait until ⟨(e, r) is received from p_coord⟩ or ⟨p_coord is suspected⟩
      if ⟨(e, r) received⟩ then
        estimate := e
        update := u
        send ack to p_coord
      else
        send nack to p_coord
                                                          {* phase 4: *}
      if i = coord then
        wait for ⟨at least one answer from a non-coordinator⟩
        if ⟨received at least one ack⟩ then
          ⟨decide on estimate, tell the others and exit⟩
                                              {* switch to next round: *}
      round := round + 1
      coord := (round mod 3) + 1
    end
```

Figure 2: Pseudocode of the algorithm from Example 3.5 [19, 33].

- In phase 2, p_1 waits for the estimate of at least one other process (i.e., p_2 or p_3). It then chooses a new estimate among the collected values (including its own one), and subsequently broadcasts this new estimate back to p_2 and p_3. (Note that as long as p_1 has not crashed, this phase will terminate since channels are reliable and at most one process may crash.)

- In phase 3, all processes receiving this new estimate (p_2 and/or p_3) adopt it as their own estimate and send back a positive acknowledgment to p_1.

- In phase 4, if p_1 receives at least one positive acknowledgment from p_2 or p_3, then it can decide on the current estimate and tell the other processes to do the same.

Note that the only place where asynchrony needs to be tackled is phase 3. In that phase, all non-coordinator processes wait for a new estimate from the coordinator. If the coordinator has crashed before sending the new estimate, processes could wait forever. So here it is useful to bound the waiting time using some form of timeout (e.g., based on some assumption Δ' of Δ). If a process "times out" on the coordinator, it simply sends a negative acknowledgment back to the coordinator and switches to the next round. Consequently, if the coordinator receives no positive acknowledgment in phase 4 but instead one or two negative acknowledgments, it does not decide and also switches to the next round. Thus, a wrong assumption Δ' will inhibit the algorithm from making progress in this round. (End of Example)

Why does the algorithm ensure safety? To violate safety, two processes must decide on different values. A process only decides on a value v during some round if a majority of processes have adopted v as their current estimate during that round. In different rounds, however, different majorities can be formed for different values. So the description of the algorithm above is not precise enough. The way the algorithm deals with the problem is to state phase 2 more precisely as:

- The coordinator only takes estimate messages with the "newest" round number tag into consideration as the new estimate.

Thus, if a majority of processes have adopted a new estimate v and sent a positive acknowledgment to the coordinator, then no other value can become the decision value. This is because among the values collected by any subsequent coordinator there will be at least one estimate v and this estimate will be the one with the latest round number.

To ensure liveness, the protocol must guarantee that eventually no false suspicions occur. In cases of partially synchronous communication, this will be the case if $\Delta' \geq \Delta$. So in every round which does not come to a deciding end, the protocol will increase the bound Δ' by some constant. This ensures that eventually Δ' will supercede Δ, thus leading to protocol termination.

The case where the given bound Δ only holds after a certain period of time is even easier to handle. In this case, we simply run the algorithm as though Δ holds already. However, if this is not the case, the worst that can happen is that the algorithm switches to the next round. So as long as it keeps on trying, eventually a decision will be reached.

The important thing to note here is that solutions to consensus in the presence of partial synchrony require a majority of processes to be correct. In a sense this is the price you have to pay for making mistakes in detecting crashes.

3.6 Remark (Proving the lower bound) It is probably easier to show that there can be no consensus algorithm in a partially synchronous system if at least half of the processes can fail. Such an argument follows the lines of Example 2.7, where safety was violated if both processes wrongly suspected the other to have crashed. This example can be generalized: A set of n processes can become "virtually partitioned" by information resulting from wrong synchrony assumptions (i.e., false crash suspicions). 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.

Note that if we let such an algorithm run in a partially synchronous system where more than half of the processes may crash, the algorithm will only terminate if it can reach a decision before half of the processes have crashed. If more than half the processes have crashed too soon, the algorithm is not able to ensure termination because there is no majority of processes anymore which could participate in a deciding run. The problem is that a process never knows when its suspicions have become correct. Thus, it always is in fear of virtual partitions, even if they cannot occur anymore. (End of Remark)

3.7 Remark (Synchrony vs. partial synchrony) It is also interesting to see what happens if we let an algorithm designed for the partially synchronous case run in the asynchronous system model. The situation is similar to the one sketched in the previous remark: The algorithm might terminate nicely, but this is not the case in all scenarios. In contrast to the partially synchronous model, suspicions based on timing bounds such as Δ will never become trustworthy in the asynchronous case. Hence, even though a majority of processes is correct, the algorithm might keep on suspecting all the other processes forever, thus violating liveness. (End of Remark)

4 Failure detectors

4.1 Introducing perfect failure detectors

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.

Obviously, we could also build consensus algorithms 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 pioneered by Chandra and Toueg [19]. Failure detectors are simply (possibly unreliable) oracles on the functional state of processes.

4.1 Example (One-process failure detector) In the understanding of Chandra and Toueg [19], a failure detector is composed of several failure detector modules, one at each process. To introduce the concept, consider the example scenario in Figure 3: Here, process p_1 is equipped with a failure detector module which indicates what functional state p_2 is in.

The failure detector module at p_1 should guarantee two things:

- It never suspects p_2 unless p_2 has actually crashed (safety), and
- if p_2 has crashed, then the failure detector module at p_1 will eventually suspect p_2 to have crashed (liveness).

(End of Example)

4.2 Remark (Relation to other types of detectors) In general, failure detectors are specified in the same way as other detectors in distributed systems (such as termination detectors [27] or general predicate detectors [10, 20]). Such a detector is specified by two predicates X and Y: We say that a detector satisfies "X detects Y" if and only if

- X does not hold unless Y holds (safety), and
- if Y holds, then eventually X will hold (liveness).

The failure detector in Example 4.1 can be seen as an instance of a general detector where

- $X \equiv$ "Failure detector module of p_1 suspects p_2 to have crashed"
- $Y \equiv$ "p_2 has actually crashed"

(End of Remark)

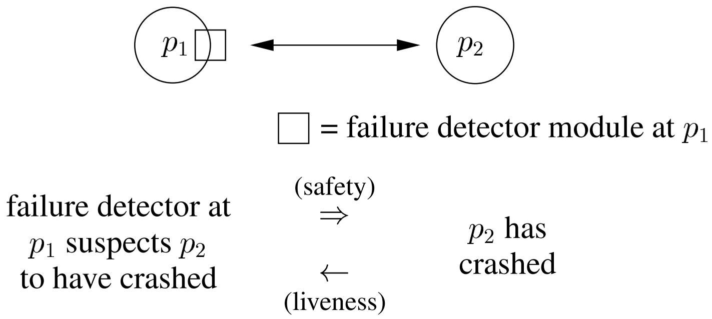

Figure 3: Safety and liveness of a simple one-process failure detector.

Obviously, the failure detector in Example 4.1 can be extended to the two process case by simply adding a failure detector module to p_2 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 functional 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 interface used by Chandra and Toueg [19] is that of a *suspicion list*, i.e., a list of identifiers signifying processes which are currently suspected, but other more event driven interfaces are also possible [39].

For an n process system, the safety 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 suspect q.

When referring to failure detectors, Chandra and Toueg [19] 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*. A perfect failure detector makes no wrong suspicions and eventually detects every crash. Obviously, if the time-free model is augmented by assuming the existence of a perfect failure detector then consensus becomes solvable.

4.2 Imperfect failure detectors

The existence of a perfect failure detector is a very strong assumption and its existence makes the model no more realistic than one where explicit synchrony bounds are added. 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 "$\forall p : \exists q : \ldots$". All such combinations are visualized in Figure 4. The quantified statements read as follows:

- ... the failure detector module at p does not suspect q unless q has crashed (safety/accuracy).
- ... if q crashes then the failure detector module at p will eventually suspect q (liveness/completeness).

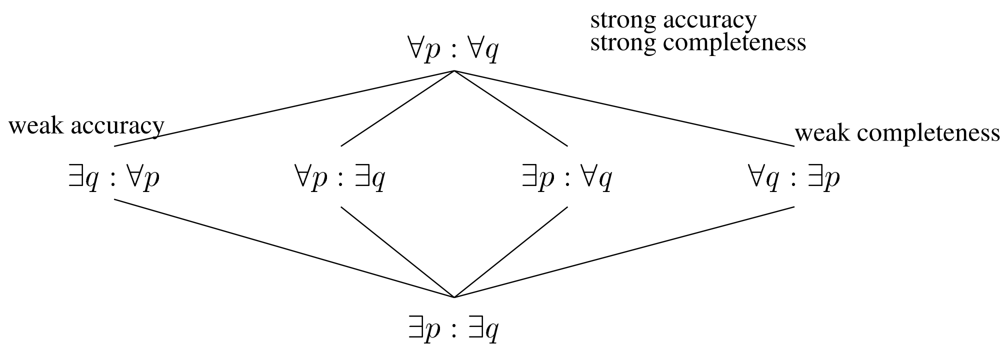

Figure 4: Variants of safety and liveness properties for failure detectors.

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* [19] 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* [19]. 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 eventually weak accuracy is called an *eventually weak failure detector*. Chandra and Toueg [19] presented an algorithm which solves consensus in asynchronous systems using an eventually weak failure detector. At first sight, this result was quite surprising since, in this model, processes could keep on making false suspicions of other processes forever. However, the properties of the failure detectors were implicitly tailored to the nature of the consensus problem, as we now explain.

The algorithm [19] works in a similar manner as the one we have presented in Example 3.5. The approach of dividing the computation into asynchronous rounds and using an implicitly defined coordinator per round is called the *rotating coordinator paradigm*. To achieve liveness, eventually one coordinator must succeed in imposing a decision value onto the system. As in the algorithm from Example 3.5, false suspicions of the coordinator merely inhibit liveness but not safety. The eventually weak accuracy property of the failure detector ensures that eventually one process will succeed in being a (non-suspected) coordinator. This effectively leads to a decision being reached but also shows the necessity to restrict the existentially quantified processes to be correct. Again, it is necessary that there is a majority of correct processes if failure detectors are allowed to make false suspicions.

Due to the precise specification of failure detector properties, it is now possible to formally prove that the algorithm actually solves consensus. However, although the solution seems clear, the formal proofs remain cumbersome.

4.3 Remark (Comparing failure detector classes) Failure detectors which satisfy the same safety and liveness properties can be grouped into failure detector classes. Can these classes be compared?

To answer this question it is necessary to define what it means for one failure detector class to be "weaker" than the other. Chandra and Toueg suggested to use a notion of *reducibility* [19] of one class to another: A failure detector class F_1 is weaker than failure detector class F_2 (denoted $F_1 \leq F_2$) if there exists a distributed algorithm which "emulates" some instance f_1 from F_1 using some instance f_2 from F_2.

To give an example: As mentioned above, a failure detector f_2 satisfying weak completeness can be used to emulate a failure detector f_1 which satisfies strong completeness using a reliable dissemination means. Such means are available in asynchronous systems with crash failures in the form of reliable broadcast algorithms [47]. So, in the sense defined above, $f_1 \leq f_2$ and thus $F_1 \leq F_2$. However, obviously a failure detector satisfying strong completeness also satisfies weak completeness, and so $F_2 \leq F_1$. In this case, both classes are said to be *equivalent*.

On the other hand, weak accuracy cannot be strengthened to strong accuracy using a similar approach for obvious reasons, so weak accuracy is strictly weaker than strong accuracy in this understanding. For a comparison between all the eight classes see the original work [19]. (End of Remark)

4.4 Remark (What is a failure detector in terms of real time?) While being methodologically inconvenient it is sometimes helpful to figure out what kind of synchrony in terms of explicit time failure detectors do offer.

Perfect failure detectors allow building a system close to the fully synchronous one. Communication can be devised in rounds; in every round, every correct process sends a message to every process. The round ends at some process if for every other process p either it has received a message from p or it suspects p to have crashed. Conversely, fully synchronous systems allow an implementation of perfect failure detectors. However, as observed by Charron-Bost, Guerraoui and Schiper [21], adding perfect failure detectors does not offer *all* the expressiveness inherent in a synchronous system because it is not possible to know whether there still is a message in transit from the crashed process [39].

Eventually perfect failure detectors can be built in partially synchronous systems, i.e., systems in which synchrony bounds exist but are not known, or bounds exist but only hold eventually. It is even possible to implement eventually perfect failure detectors in systems where synchrony bounds exist, but they are not known *and* hold only eventually [19]. Consequently, a system using an eventually perfect failure detector can be viewed as a fully synchronous system which eventually stabilizes. The fact that it is not possible to know when the system has stabilized mirrors the fact that the bounds are not known.

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.

It is difficult to express these synchrony features in terms of explicit synchrony bounds. This is an advantage of the failure detector model over the explicit synchrony models. (End of Remark)

4.3 Extensions to other problems and failure models

Failure detectors were introduced to circumvent the impossibility of consensus in asynchronous systems. They describe several intermediate models between fully synchronous systems and fully asynchronous systems that are difficult to specify using explicit synchrony bounds. However, the "classic" failure detectors have been *crash* failure detectors, i.e., they were tailored to the crash failure model.

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 [3] output a vector of unbounded counters hinting on how often a process has been suspected.

Finding out how a failure detector should convey its information is much harder since we would like to require the failure detector properties to be as weak as possible. The eventual weak accuracy property of the crash failure detectors was shown to be necessary to solve asynchronous consensus [17] which is a clear indication that a central correct process seems to be inherent in any attempt to solve consensus (even if stronger types of failure detectors are available [52]). Finding adequate failure detector specifications for specific situations is a highly non-trivial task.

4.5 Remark (Work on other problems and failure models) Nevertheless it is possible to learn from previous work.

Problems which are directly related to consensus can be solved using the same type of failure detectors (e.g., *atomic broadcast* [19] or *non blocking atomic commitment* [41, 42]).

Failure detector specifications have been extended to environments where the network may partition [2, 43], partly due to communication links suffering from failures [29, 30]. Lossy links are a usual assumption in work on consensus in the *crash-recovery* failure model [1, 49, 56]. In this model, questions of to what extent stable storage is necessary are also important. Lo and Hadzilacos [53] study failure detection and consensus in a shared memory setting. Failure detectors in a Byzantine environment are the focus of Doudou and Schiper [32].

Other problems than consensus have been studied adapting the failure detection approach. Sabel and Marzullo [59] consider the *election* problem (see also Larrea et al. [52]) while Matsui et al. [55] investigate *eventual leader election of k processes* (which they call *k-consensus*). Issues of group communication have also been considered (e.g., *atomic multicast* [45] and *generic broadcast* [5, 57]). Predicate detection in faulty environments is investigated by Garg and Mitchell [38] and Gärtner and Kloppenburg [39]. (End of Remark)

4.6 Remark (Efficiency issues and performance metrics) If questions of solvability in a time-free setting have been answered, it is natural to compare different solutions regarding performance metrics. Algorithms which improve the original algorithm of Chandra and Toueg [19] with respect to the "time" it takes for consensus to terminate in favorable settings have been proposed by Schiper [60, 61], Hurfin and Raynal [50] and by Brasileiro et al. [16]. Sergent, Défago and Schiper [65] study the impact of different failure detection algorithms on the performance of consensus algorithms. (End of Remark)

4.7 Remark (Relation to stabilization research) Safety properties which hold only eventually are sometimes investigated under the heading of *stabilization* [9, 26]. Stabilization usually assumes *transient* faults that may impact the entire state of the system and postulates that the system converges to specified behavior after fault occurrences cease. In this context, obviously no safety property can be maintained, however, stabilizing algorithms exist for a wide variety of problems [31, 64]. Failure detectors whose accuracy holds only eventually can be regarded as stabilizing program components and implementations have been investigated explicitly under the heading of stabilization by Beauquier and Kekkonen-Moneta [13]. Others [8, 40] have studied the solvability of problems in environments which combine crash and arbitrary transient failures. (End of Remark)

4.8 Remark (Quiescence of algorithms) Aguilera, Chen and Toueg [2, 4] turn their attention to *quiescent algorithms*, i.e., algorithms which eventually stop sending messages. Quiescence can be regarded as a stabilization-like property lying between termination and non-termination. 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*) and terminating ones (like consensus) at higher layers. The idea of this approach is to use the failure detection service as the source of activation of higher level algorithms. (End of Remark)

5 Discussion

This section briefly touches other prominent system models for fault tolerance in Section 5.1 and then summarizes and concludes the discussion in Section 5.2.

5.1 Other system models

Apart from the introduction of explicit synchrony and the use of failure detectors, there have been a few other attempts to describe and formalize models for the design and evaluation of fault-tolerant systems. The most prominent ones are the *timed asynchronous system model* of Cristian and Fetzer [25] and the *quasi-synchronous model* of Almeida et al. [7]. While the former is very close to the time-free system, the latter is very close to fully synchronous systems.

5.1 Remark (The timed asynchronous model [25]) The timed asynchronous system model takes the time-free model as a basis. It contains, however, an explicit notion of real-time through the assumption that processes have access to local hardware clocks and that these clocks run within a linear envelope of real-time. This means that it is always possible to state a real-time bound on the maximum clock difference between each pair of clocks. Note that this does not mean that bounds exist on processing speeds or message delivery delays. On the contrary, the model explicitly assumes that communication is via message passing and there is no bound on message delivery delay. Just like in the time-free model, there is no bound on the relative processing speed of processes.

The failure model assumed in the timed asynchronous model is a little different from those we have come across above. Communication is assumed to be unreliable, i.e., channels may lose messages. Also, processes may crash and subsequently recover. They can use stable storage to store parts of their state and retrieve it after a crash. The model explicitly does not assume any bound on the frequency of failures that can occur within the system.

Through the notion of real-time provided by the hardware clocks it is possible to define real-time bounds for all services provided in a timed asynchronous system. In fact, this part of a service specification is mandatory. This means that while it is still impossible to reliably detect a process crash, it is now possible to tell whether a reply has not met its real-time deadline. (This is called *fail-awareness* [36].) Hence, solving consensus is not possible in the basic timed asynchronous system. More generally, since there are no bounds on fault occurrences, in this model it is not possible to ensure any liveness property at all.

But if consensus is not solvable in the timed asynchronous system model, why use it as a system model for fault tolerance? First of all, the system model is arguably very realistic. It is based on a substantial body of simulation data which is presented in the original article [25]. Also, because it does not assume explicit bounds on fault occurrences, the model has a high assumption coverage [58] meaning the probability of the assumptions holding in practice is rather high. The assumption about the existence of hardware clocks is also reflected in practice: All workstations currently on the market have hardware clocks with bounded drift rate (the bound is somewhere between 10^{-4} and 10^{-6}).

But even a very realistic model is of no use if consensus remains unsolvable. To address this question, Cristian and Fetzer made the following observation: In practice, systems alternate between long periods of stability and short periods of instability. The measurements they give in their article [25] which were made in a local area network environment show that the average distance between unstable periods is 218 seconds, while the average length of an unstable period was about 340 milliseconds (this gives a ratio of $641 : 1$). This observation allows to formulate *progress assumptions* of the form: "There exists a constant c, such that infinitely often there will be a stable period of length at least c." (More generally, this can be formulated for a "stable partition" of processes.) In other words, this means that infinitely often the system will be synchronous for at least c time.

Using progress assumptions it is now possible to specify liveness properties in the following way: "Assuming that some stability predicate holds, then the system will eventually perform an action." The stability predicate S can be, e.g., "infinitely often a majority of processes is synchronous for at least 2 seconds." Liveness properties of form L are transformed into weaker variants of the form $S \Rightarrow L$ called *conditional timeliness properties* [25]. For example, the termination property of consensus is rephrased as:

- Given S, then eventually every correct process decides.

Weakening of the liveness properties is nessesarily essential because there is no bound on the failure rate. (Such a bound is introduced by the additional condition.) In the failure detector model, a bound on the failures is part of the model. So it looks as though the liveness properties are equivalent. However, predicate S specifies a minimum duration c of stability. It is necessary that algorithms can exploit this duration to make progress. In the timed-asynchronous model, algorithms which sometimes cannot progress within c time are thus not guaranteed to achieve liveness. In the failure detector model, they are. This aspect has been often noted in work on asynchronous models [19, 28]: failure detector/synchrony properties are not really assumed to hold forever; they must hold for "long enough". The timed asynchronous model gives an indication whether this assumption is justified. (End of Remark)

5.2 Remark (The quasi synchronous model [7]) The quasi synchronous model is at heart a fully synchronous model, i.e., it assumes that bounds on all relevant timing parameters exist and are known (bounds on processing delays, message delivery delays, hardware clock drift rate, differences of hardware clocks). To make the model more realistic, the timing bounds are set very conservatively (i.e., made very large), so that the assumed coverage of these bounds is 1. The failure model in the quasi synchronous model are similar to the asynchronous model, i.e., message omission and process crashes may occur, however, there are bounds on both. Obviously, in such a system it is possible to implement a perfect failure detector and, hence, consensus is solvable. But why is it a good model if it is fully synchronous?

The model allows to assume shorter timing bounds. The shorter these bounds become, the smaller the assumed coverage will be. To prevent "unsafe" states (e.g., due to false crash suspicions) when time bounds are not met, the model assumes the existence of a *perfect timing failure detector*. It is defined as follows:

- any timing failure is detected by all correct processes within a maximum real-time delay, and
- an event that occurs within its deadline is never considered a timing failure by any correct process.

The timing failure detector is dependent on the specified shorter bounds on timing parameters. An application may request timing failure detection with regard to longer or shorter bounds. Using this idea, it is possible to dynamically switch to different levels of synchrony within the system, e.g., assume larger bounds on message delivery delay if many messages arrive late. (This is called *coverage stability* [7].) In consequence, liveness is achieved by switching bounds to lower and lower values until the liveness property holds (in the worst case this may happen only when the bound values approach the existing bounds and the assumption coverage is 1). Since fault occurrences are bounded, the system can be equipped with enough redundancy such that liveness is always guaranteed.

Obviously, to be practical, the perfect timing failure detector must be implemented. From the things said up to now in this article, this seems rather difficult. However, Almeida et al. [7] argue that today there exists network technologies which make perfect timing failure detection feasible (e.g., specialized real-time network technologies like ATM, ISDN or GSM). To implement a timing failure detector, it suffices if only a part of the entire network is in fact fully synchronous (see Figure 5). This part can even have a much smaller bandwidth that the rest of the network as long as it is fully synchronous. Thus, where the failure detector model assumed an *eventually* "one-way synchronous" unfixed subnet, the quasi-synchronous model exploits a *fixed* and always synchronous subnet.

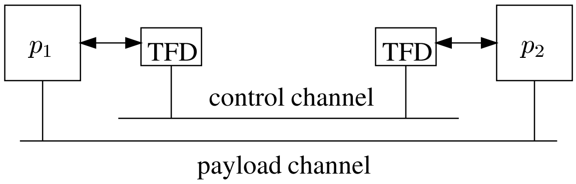

Figure 5: Timing failure detection using a dedicated control channel.

(End of Remark)

5.3 Remark (Randomization and fairness) In 1983, Ben-Or [14] 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. Later, Aguilera and Toueg [6] combined this approach with failure detection.

Recently, Völzer [69] studied the relationship between fairness and randomization. He showed that to solve consensus it is sufficient to postulate a certain kind of fairness called *synchronization fairness*. Briefly spoken, synchronization fairness 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. Synchronization fairness means that eventually these resources will be "synchronized" and available to that process. This special type of fairness can be implemented using randomization. This new concept shows that there are many different abstractions (see also Halpern and Ricciardi [48]) which can be added to a system model and help solve problems like consensus. Showing the precise interrelations between these abstractions and comparing their usefulness towards the design of fault tolerant systems is still a largely uncharted area.
(End of Remark)

5.2 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. This text has attempted to introduce this concept to an unprepared but interested audience. In doing this, we have tried to show the merits of the failure detector model. It is theoretically sound, meaning that it can be fully formalized and lends itself well to analytical studies of algorithms, their verification and to proving impossibility results. Failure detectors are also methodologically advantageous, since they separate the concerns of abstract functionality and functionality implementation. The advantages of the failure detector model mostly stem from not having to deal with real-time issues when reasoning about distributed algorithms and systems. However, failure detectors must be adjusted to each new failure model considered.

5.4 Remark (Comparison with other system models) In contrast to the failure detector model, the basic timed asynchronous model was constructed having real-world systems in mind. Because the failure model in the timed asynchronous model is usually weaker than that of the failure detector model, the timed asynchronous model offers a higher fault assumption coverage, although this same fact makes it impossible to achieve any form of liveness. However, due to hardware clocks and requiring services to be timed, it is possible to detect service violations and thus always remain in a safe state. This makes the timed asynchronous model a good model for building fail-safe systems in practice (which has been done [35]). The model is in this sense very conservative.

The timed-asynchronous model also separates the assumptions on failure bounds from the basic model and explicitly places them into the liveness part of the protocol specification. This is again useful in practical settings to communicate fault-tolerance properties and calculate coverage probabilities.

The quasi synchronous model is the "most synchronous" model discussed here, since it is the only one where bounds on all timing parameters are assumed to exist and to be known and where a priori perfect failure detectors exist. Arguably, it is also very realistic because it can exploit timeliness specifications of real networks (e.g., ATM, GSM). Since it places bounds on the occurrence of faults, the failure assumption coverage is not as high as in the timed asynchronous model. However, the quasi synchronous model is the only model in which "real-time liveness" can be dealt with (i.e., it can be specified and proved to hold). Through the concept of parameterized timing failure detectors it is also capable of dealing with real-time quality of service specifications. Thus, it is a good model for fault-tolerant applications with hard and soft real-time deadlines.

(End of Remark)

Acknowledgments

The author wishes to thank Rachid Guerraoui for helpful discussions and Xavier Défago and Stefan Pleisch for their comments on an earlier version of this paper.

References

[1] Marcos Kawazoe Aguilera, Wei Chen, and Sam Toueg. Failure detection and consensus in the crash-recovery model. In *Proceedings of the 12th International Symposium on Distributed Computing (DISC)*, pages 231–245, September 1998.

[2] Marcos Kawazoe Aguilera, Wei Chen, and Sam Toueg. Using the heartbeat failure detector for quiescent reliable communication and consensus in partitionable networks. *Theoretical Computer Science*, 220(1):3–30, June 1999.

[3] Marcos Kawazoe Aguilera, Wei Chen, and Sam Toueg. Failure detection and consensus in the crash recovery model. *Distributed Computing*, 13(2):99–125, April 2000.

[4] Marcos Kawazoe Aguilera, Wei Chen, and Sam Toueg. On quiescent reliable communication. *SIAM Journal on Computing*, 29(6):2040–2073, December 2000.

[5] Marcos Kawazoe Aguilera, Carole Delporte-Gallet, Hugues Fauconnier, and Sam Toueg. Thrifty generic broadcast. In *Proceedings of the 14th International Symposium on Distributed Computing (DISC)*, number 1914 in Lecture Notes in Computer Science, pages 268–282, Toledo, Spain, October 2000. Springer-Verlag.

[6] Marcos Kawazoe Aguilera and Sam Toueg. Randomization and failure detection: A hybrid approach to solve consensus. In Özalp Babaoglu and Keith Marzullo, editors, *Proceedings of the 10th International Workshop on Distributed Algorithms (WDAG96)*, volume 1151 of *Lecture Notes in Computer Science*, pages 29–39, Bologna, Italy, 9–11 October 1996. Springer-Verlag.

[7] Carlos Almeida, Paulo Veríssimo, and António Casimiro. The quasi-synchronous approach to fault-tolerant and real-time communication and processing. Technical Report CTI RT-98-04, Instituto Superior Técnico, Lisboa, Portugal, July 1998.

[8] Efthymios Anagnostou and Vassos Hadzilacos. Tolerating transient and permanent failures. In *Proceedings of the 7th International Workshop on Distributed Algorithms (WDAG93)*, pages 174–188, 1993.

[9] Anish Arora. Stabilization. In Partha Dasgupta and Joseph E. Urban, editors, *Encyclopedia of Distributed Computing*. Kluwer, 2000. to appear.

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

[11] Michael Barborak, Anton Dahbura, and Miroslaw Malek. The consensus problem in fault-tolerant computing. *ACM Computing Surveys*, 25(2):171–220, June 1993.

[12] Anindya Basu, Bernadette Charron-Bost, and Sam Toueg. Simulating reliable links with unreliable links in the presence of process crashes. In *Proceedings of the 10th International Workshop on Distributed Algorithms (WDAG96)*, pages 105–122, 1996.

[13] Joffroy Beauquier and Synnöve Kekkonen-Moneta. Fault-tolerance and self-stabilization: impossibility results and solutions using self-stabilizing failure detectors. *International Journal of System Science*, 28(11):1177–1187, 1997.

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

[15] Ofer Biran, Shlomo Moran, and Shmuel Zaks. A combinatorial characterization of the distributed 1-solvable tasks. *Journal of Algorithms*, 11(3):420–440, September 1990.

[16] Francisco Brasileiro, Fabíola Greve, Achour Mostéfaoui, and Michel Raynal. Consensus in one communication step. Technical Report PI-1321, IRISA, Rennes, France, 2000.

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

[18] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for asynchronous systems. In *Proceedings of the 10th Annual ACM Symposium on Principles of Distributed Computing (PODC'91)*, pages 325–340, 1991.

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

[20] K. Mani Chandy and Jayadev Misra. *Parallel Program Design: A Foundation*. Addison-Wesley, Reading, Mass., 1988.

[21] Bernadette Charron-Bost, Rachid Guerraoui, and André Schiper. Synchronous system and perfect failure detector: Solvability and efficiency issues. In *International Conference on Dependable Systems and Networks (IEEE Computer Society)*, 2000.

[22] Bernadette Charron-Bost, Sam Toueg, and Anindya Basu. Revisiting safety and liveness in the context of failures. In C. Palamidessi, editor, *Proceedings of CONCUR2000 – Concurrency Theory, 11th Int. Conference*, number 1877 in Lecture Notes in Computer Science, pages 552–565, University Park, PA, August 2000. Springer-Verlag.

[23] Wei Chen, Sam Toueg, and Marcos Kawazoe Aguilera. On the quality of service of failure detectors. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN 2000)*, New York, June 2000. IEEE Computer Society Press.

[24] Flaviu Cristian. Probabilistic clock synchronization. *Distributed Computing*, 3:146–158, 1989.

[25] Flaviu Cristian and Christof Fetzer. The timed asynchronous distributed system model. *IEEE Transactions on Parallel and Distributed Systems*, 10(6), June 1999.

[26] Edsger W. Dijkstra. Self stabilizing systems in spite of distributed control. *Communications of the ACM*, 17(11):643–644, 1974.

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

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

[29] Danny Dolev, Roy Friedman, Idit Keidar, and Dahlia Malkhi. Failure detectors in omission failure environments. Technical Report TR96-1608, Cornell University, Computer Science Department, September 1996.

[30] Danny Dolev, Roy Friedmann, Idit Keidar, and Dahlia Malkhi. Failure detectors in omission failure environments. In *Proceedings of the 16th Annual ACM Symposium on Principles of Distributed Computing (PODC97)*, 1997.

[31] Shlomi Dolev. *Self-Stabilization*. MIT Press, 2000.

[32] Assai Doudou and André Schiper. Muteness detectors for consensus with Byzantine processes. Technical report, EPFL – Département d'Informatique, Lausanne, Switzerland, 1997.

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

[34] Klaus Echtle and Asif Masum. Understanding cooperative byzantine failures: A novel failure classification to enable efficient fault-tolerant protocols. In *Proceedings of the Annual IEEE Workshop on Fault-Tolerant Parallel and Distributed Systems (FTPDS'99)*, San Juan, Puerto Rico, USA, April 1999. Kluwer.

[35] Didier Essame, Jean Arlat, and David Powell. Padre: A protocol for asymmetric duplex redundancy. In *Proceedings of the Seventh IFIP International Working Conference on Dependable Computing for Critical Applications*, San Jose, USA, January 1999.

[36] Christof Fetzer and Flaviu Cristian. Fail-awareness: An approach to construct fail-safe applications. In *Proceedings of The Twenty-Seventh Annual International Symposium on Fault-Tolerant Computing (FTCS'97)*, pages 282–291. IEEE, June 1997.

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

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

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

[40] Ajei S. Gopal and Kenneth J. Perry. Unifying self-stabilization and fault-tolerance. In *Proceedings of the 12th Annual ACM Symposium on Principles of Distributed Computing (PODC'93)*, pages 195–206. ACM Press, 1993.

[41] Rachid Guerraoui. Revisiting the relationship between non-blocking atomic commitment and consensus. In Jean-Michel Hélary and Michel Raynal, editors, *Proceedings of the 9th International Workshop on Distributed Algorithms (WDAG95)*, volume 972 of *Lecture Notes in Computer Science*, pages 87–100, Le Mont-Saint-Michel, France, 13–15 September 1995. Springer-Verlag.

[42] Rachid Guerraoui, Mikel Larrea, and André Schiper. Non blocking atomic commitment with an unreliable failure detector. In *Proceedings of the 14th IEEE Symposium on Reliable Distributed Systems (SRDS95)*, September 1995.

[43] Rachid Guerraoui and André Schiper. "Gamma-accurate" failure detectors. In Özalp Babaoglu and Keith Marzullo, editors, *Distributed Algorithms, 10th International Workshop, WDAG '96*, volume 1151 of *Lecture Notes in Computer Science*, pages 269–286, Bologna, Italy, 9–11 October 1996. Springer-Verlag.

[44] Rachid Guerraoui and André Schiper. Consensus: the big misunderstanding. In *Proceedings of the 6th Workshop on Future Trends of Distributed Computing Systems (FTDCS-6)*, October 1997.

[45] Rachid Guerraoui and André Schiper. Genuine atomic multicast. In *Proceedings of the 11th International Workshop on Distributed Algorithms (WDAG97)*, number 1320 in Lecture Notes in Computer Science, pages 141–154. Springer-Verlag, September 1997.

[46] Vassos Hadzilacos. *Issues of Fault Tolerance in Concurrent Computations*. PhD thesis, Harvard University, 1984. also published as Technical Report TR11-84.

[47] Vassos Hadzilacos and Sam Toueg. A modular approach to fault-tolerant broadcasts and related problems. Technical Report TR94-1425, Cornell University, Computer Science Department, May 1994.

[48] Joseph Y. Halpern and Aleta Ricciardi. A knowledge-theoretic analysis of uniform distributed coordination and failure detectors. In *Proceedings of the 18th Annual ACM Symposium on Principles of Distributed Computing (PODC'99)*, pages 73–82, 1999.

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

[50] Michel Hurfin and Michel Raynal. A simple and fast asynchronous consensus protocol based on a weak failure detector. *Distributed Computing*, 12(4):209–223, 1999.

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

[52] Mikel Larrea, Antonio Fernández, and Sergio Arv́alo. Eventually consistent failure detectors. Technical report, Universidad Pública de Navarra, Spain, April 2000. Presented as a brief announcement at DISC2000.

[53] Wai-Kau Lo and Vassos Hadzilacos. Using failure detectors to solve consensus in asynchronous shared-memory systems (extended abstract). In Gerard Tel and Paul M. B. Vitányi, editors, *Proceedings of the 8th International Workshop on Distributed Algorithms (WDAG94)*, volume 857 of *Lecture Notes in Computer Science*, pages 280–295, Terschelling, The Netherlands, 29 September–1 October 1994. Springer-Verlag.

[54] Nancy Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.

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

[56] Rui Oliveira, Rachid Guerraoui, and André Schiper. Consensus in the crash-recover model. Technical Report TR-97/239, EPFL – Départment d'Informatique, Lausanne, Switzerland, August 1997.

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

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

[59] Laura S. Sabel and Keith Marzullo. Election vs. consensus in asynchronous systems. Technical Report TR95-1488, Cornell University, Computer Science Department, February 1995.

[60] André Schiper. Early consensus in an asynchronous system with a weak failure detector. *Distributed Computing*, 10(3):149–157, 1997.

[61] André Schiper. Erratum: Early consensus in an asynchronous system with a weak failure detector. *Distributed Computing*, 10:198, 1997.

[62] Richard D. Schlichting and Fred B. Schneider. Fail stop processors: An approach to designing fault-tolerant computing systems. *ACM Transactions on Computer Systems*, 1(3):222–238, August 1983.

[63] Fred B. Schneider. What good are models and what models are good? In Sape Mullender, editor, *Distributed Systems*, chapter 2, pages 17–26. Addison-Wesley, second edition, 1993.

[64] Marco Schneider. Self-stabilization. *ACM Computing Surveys*, 25(1):45–67, 1993.

[65] Nicole Sergent, Xavier Défago, and André Schiper. Failure detectors: implementation issues and impact on consensus performance. Technical Report SSC/1999/019, École Polytechnique Fédérale de Lausanne, Switzerland, 1999.

[66] Gerard Tel. *Introduction to Distributed Algorithms*. Cambridge University Press, 1994.

[67] John Turek and Dennis Shasha. The many faces of consensus in distributed systems. *IEEE Computer*, 25(6):8–17, June 1992.

[68] Werner Vogels. World wide failures. In *Proceedings of the ACM SIGOPS European Workshop*, Connemara, Ireland, September 1996.

[69] Hagen Völzer. *Fairness, Randomisierung und Konspiration in verteilten Algorithmen*. PhD thesis, Humboldt Universität zu Berlin, Fakultät für Informatik, December 2000.