Failure Detection Sequencers: Necessary and Sufficient Information about Failures to Solve Predicate Detection

Felix C. Gärtner[1] and Stefan Pleisch[2]

[1] Department of Computer Science, Darmstadt University of Technology, D-64283 Darmstadt, Germany, felix@informatik.tu-darmstadt.de
[2] IBM Research, Zurich Research Laboratory CH-8803 Rüschlikon, Switzerland, spl@zurich.ibm.com

Abstract. This paper investigates the amount of information about failures needed to solve the predicate detection problem in asynchronous systems with crash failures. In particular, we show that predicate detection cannot be solved with traditional failure detectors, which are only functions of failures. In analogy to the definition of failure detectors, we define a *failure detection sequencer*, which can be regarded as a generalization of a failure detector. More specifically, our failure detection sequencer Σ outputs information about failures *and* about the final state of the crashed process. We show that Σ is necessary and sufficient to solve predicate detection. Moreover, Σ can be implemented in synchronous systems. Finally, we relate sequencers to perfect failure detectors and characterize the amount of knowledge about failures they additionally offer.

1 Introduction

Predicate detection in distributed settings is a well-understood problem and many techniques together with their detection semantics have been proposed [6]. Most of these techniques address predicate detection with the assumption that no faults occur in the system. However, it is desirable to also detect predicates which refer to the operational state of processes, e.g., predicates such as "$x_i = 1 \wedge \mathit{crashed}_i$", where $\mathit{crashed}_i$ is a predicate that is true iff (if and only if) process p_i has crashed. Since $x_i = 1$ might indicate the presence of a lock, the given predicate can be used to formalize special conditions such as "process p_i crashed while holding a lock", which is useful in the context of databases.

In the context of *crash* failures and the *consensus* problem, *failure detectors* have been devised to provide information about failures [3], but they offer "solely" information about failures. To detect general predicates such as the example predicate above, failure detection information needs to be combined with additional information about the internal state of a process. Indeed, while a failure detector may capture the predicate $\mathit{crashed}_i$, it gives no information about the value of x_i.

Ideally, a predicate detection algorithm never erroneously detects a predicate and does not miss any occurrence of the predicate in the underlying computation. As shown in [10], the quality of predicate detection critically depends on the

quality of failure detection. This explains why work in [8, 16, 18] puts a restriction on the type of detectable predicates, or [9] weakens the semantics of predicate detection.

In previous work [10], we have investigated predicate detection in an asynchronous system with crash failures and found that it is impossible to solve predicate detection even with a very strong failure detector, the *perfect failure detector* [3]. In this paper, we show that predicate detection cannot be solved with *any* failure detector (as defined in [3]), no matter how strong it is. For example, consider a "real-time perfect" failure detector which makes no mistakes and flags the occurrence of a crash *immediately*. Even this failure detector is insufficient to solve predicate detection. The reason for this impossibility is that failure detectors are only functions of failures. Our proof is a generalization of previous impossibility proofs by the present authors [10] and by Charron-Bost, Guerraoui and Schiper [5]. We attempt to remedy the unpleasant situation caused by the result and (in analogy to the definition of failure detectors) define a *failure detection sequencer*. A failure detection sequencer is a generalization of a failure detector in that it conveys information that is a function of the failures *and* the current history of the system which is under observation. To solve predicate detection, we define a particular failure detection sequencer class Σ, that only gives one additional piece of information: for every crashed process it gives the latest state of the process before this one crashes. We show that Σ is necessary and sufficient to solve predicate detection and consequently is the "weakest failure detection sequencer" to solve predicate detection.

Although Σ is in a sense "stronger" than a perfect failure detector, it is still possible to implement Σ in synchronous systems. Moreover, we argue that using Σ it is possible to implement a *synchronizer* for asynchronous crash-affected systems which makes these systems equivalent to purely synchronous systems in terms of the solvability of *time-free* [5] problems. We finally argue that while perfect failure detectors can be viewed as capturing the synchrony of processes, failure detection sequencers in addition also capture the synchrony of communication.

After presenting the system model and defining the problem of predicate detection in Sections 2 and 3, we present our contributions in the following order: First, we show that it is impossible to achieve predicate detection with any failure detector in the sense of Chandra and Toueg [3] in Section 4. Section 5 introduces the failure detection sequencer abstraction and shows that a particular sequencer Σ is equivalent to predicate detection. In Section 6, we show how to implement Σ and then discuss the strength of Σ in Section 7. Finally, Section 8 concludes the paper. For the full proofs, the reader is referred to [11].

2 Model

We consider an asynchronous distributed system in which processes communicate via message passing. This means that no bounds on message transmission time nor on relative process speeds exist. Message delivery is reliable, i.e., a sent

message is eventually delivered, no spurious messages are delivered, and messages are not altered. Processes can fail by crashing, i.e., they simply stop to execute steps. Crashed processes do not recover any more. Processes which do not crash are called *correct*.

2.1 Distributed Computations

A distributed system, called the *application system*, consists of a finite set Π of n processes $p_1, p_2, \ldots, p_n$ (called *application processes*). Each process p_i has a local state s_i (defined by the values assigned to its local variables) and performs atomic state transitions according to a local algorithm A. Such a state transition is also called an *event*. Sending and receiving a message also results in a state change. If a process p_i sends a message in state s_i which is received by process p_j resulting in state s_j, we say that s_i and s_j *correspond*.

We define a relation of *potential causality* (denoted "$\rightarrow$") [2] on local states as the transitive closure of the following two relations:

- $s \rightarrow s'$ if s and s' happen on the same process and s happens before s'.
- $s \rightarrow s'$ if s and s' happen on different processes and s and s' correspond.

A *local history of* p_i is an (infinite) sequence $s_1, s_2, \ldots$ of states. A *distributed computation* is defined as a set of local histories, one for every process. A *global state* of the computation is a vector $G = (s_1, s_2, \ldots, s_n)$ of local states, one for each process. Each local state identifies a point in the local history of a process and thus is equivalent to the set of all local states the process went through to reach its "current" local state. A global state G is *consistent* if the union of these sets (of all local states in G) is left-closed with respect to $\rightarrow$, i.e., if a state s is in this set and $s' \rightarrow s$, then s' must also be in this set. The set of all global states of a computation together with $\rightarrow$ define a lattice [14].

We assume the existence of a discrete global clock. Processes do not have access to this global clock; it is merely a fictional device to simplify presentation. Let $\mathcal{T}$ denote the range of output values of the global clock. For simplicity we think of $\mathcal{T}$ to be the set of natural numbers.

2.2 Failure Detectors

A *failure detector* is a device that can be queried at any time $t \in \mathcal{T}$ and outputs the set of processes that it suspects to have crashed at time t.

We adopt the formal definitions of failure detectors by Chandra and Toueg [3]. A *failure pattern* F is a mapping from $\mathcal{T}$ to the powerset of Π. The value of $F(t)$ specifies the set of application processes that have crashed until time $t \in \mathcal{T}$. We define $\mathit{crashed}(F) = \bigcup_{t \in \mathcal{T}} F(t)$ and $\mathit{correct}(F) = \Pi \setminus \mathit{crashed}(F)$. A *failure detector history* H is a mapping from $\Pi \times \mathcal{T}$ to the powerset of Π. The value of $H(p, t)$ denotes the return value of the failure detector module for process p at time t, i.e., if p queries the failure detector at time t, $H(p, t)$ contains the set of processes suspected at that time.

A *failure detector* $\mathcal{D}$ maps a failure pattern F to a set of failure detector histories. The set of histories returned by the failure detector satisfy certain accuracy and completeness properties. A perfect failure detector satisfies *strong accuracy* and *strong completeness*:

- Strong accuracy: no process is suspected before it crashes. Formally:

$$\forall F. \forall H \in \mathcal{D}(F). \forall t \in \mathcal{T}. \forall p, q \in \Pi \setminus F(t). p \notin H(q, t)$$

- Strong completeness: a crashed process is eventually permanently suspected by every correct process. Formally:

$$\forall F. \forall H \in \mathcal{D}(F). \exists t \in \mathcal{T}. \forall p \in \mathit{crashed}(F). \forall q \in \mathit{correct}(F). \forall t' \geq t. p \in H(q, t')$$

The set of all perfect failure detectors is denoted by $\mathcal{P}$. In the following, we will sometimes use the symbol $\mathcal{P}$ as a shorthand for any failure detector from $\mathcal{P}$.

2.3 Runs and Steps

Chandra and Toueg [3] define a computation (which they call a *run*) to be a tuple $R = (F, \mathcal{D}, I, S, T)$, where S is a sequence of algorithm steps and T is a sequence of increasing time values when these steps are taken. Steps are defined with respect to an algorithm which in turn is a collection of deterministic automata. We define a run in a slightly different but equivalent manner. Instead of S and T we use two functions: a *step function* S_s from $\mathcal{T}$ to the set of all algorithm steps, and a process function S_p from $\mathcal{T}$ to Π. Briefly spoken, $S_p(t)$ denotes the process which takes a step at time t and $S_s(t)$ identifies the step which was taken. Without loss of generality, we assume that at any instance of time at most one process takes a step. If no process takes a step at time t, both functions evaluate to $\perp$. A computation then is a tuple $R = (F, \mathcal{D}, I, S_s, S_p)$.

In predicate detection, which is defined in the following section, we wish to detect whether a predicate holds on the state of processes. We assume that the state resulting from an algorithm step contains enough information to infer the validity of the predicate. For instance, a predicate referring to the number of events that have occurred in the system requires that an event counter is part of the local state. In general, the state must allow to infer the events that have happened and are of interest for the predicate. We assume that there is at least a correspondence between the most recent step of a process and the state resulting from executing that step. In this paper, we use the terms *state* and *step* interchangeably.

3 Predicate Detection

To detect predicates in the application system (see Section 2.1), the application system is extended with a set Φ of m *monitor processes* $b_1, \ldots, b_m$. The sets Π and Φ together form the *observation system*.

While application processes may crash we assume, for simplicity, that monitor processes do not[1]. Crashes of application processes do not change the local state of the process[9]. However, the operational state of a process p_i is modeled by a virtual boolean variable $crashed_i$ on every monitor. The global state of the system together with the vector of *crashed* variables defines the *extended global state* of the system.

The task of the monitor processes is to observe the application processes and invoke a special primitive *detected* if the state of the system satisfies a certain predicate. A predicate ϕ is a boolean function on the extended global state of the application system. For example, the predicate $x_i = 2 \wedge crashed_i$ is true in a global state if the variable x_i of p_i equals 2 and p_i has crashed. We say that ϕ *holds* in a computation c iff there exists a consistent global state in c such that ϕ is true in that state.

In our version of predicate detection, monitors can observe multiple predicates simultaneously. More specifically, the predicate detection algorithm maintains a set S of currently active predicates. A special primitive $fork(\phi)$ can be used to add a predicate ϕ to this set. Whenever some $\phi \in S$ is found to hold in the computation, the predicate detection algorithm indicates this by pointing to ϕ, i.e., by calling $detected(\phi)$. Formally, detecting any $\phi \in S$ corresponds to detecting the disjunction of all such ϕ. This formulation of predicate detection has the important advantage of allowing us to increase the set of observed predicates at runtime. In other words, it does not matter when a predicate ϕ is added to S. Even if ϕ held "early" in the computation and $fork(\phi)$ is invoked very late (e.g., after hours), then still the algorithm must eventually invoke $detected(\phi)$ [2]. In this sense, our predicate detection concept is *adaptive* and thus slightly more general than other definitions of predicate detection (e.g., *perfect predicate detection* [10]). This is reflected in the following definition:

Definition 1 (predicate detection). *A predicate detection algorithm is a distributed algorithm running on the observation system with an input operation fork() and an output operation detected(). Using fork(ϕ) a new predicate can be added to an initially empty set S of predicates. The algorithm must satisfy the following properties:*

- *(Safety) If a monitor invokes detected(ϕ) then ϕ holds in the computation and $\phi \in S$.*
- *(Liveness) If $\phi \in S$ and ϕ holds in the computation, then eventually a monitor must invoke detected(ϕ).*

Our definition of predicate detection makes no reference to a specific implementation. Generally, one expects application processes to use causal broadcast [2] to consistently disseminate information about every local state change to all

[1] Our results remain valid even if some monitors fail. The possibility results in Sect. 5 merely require that at least one monitor is correct.

[2] Later in Section 5.2 we show how this property can be implemented in asynchronous systems.

monitor processes. But this is not required by the specification. Furthermore, there is no indication how monitors keep track of the changes of *crashed* values of processes, i.e., we do not postulate the existence of a special type of failure detector in the specification. However, failure detection can be considered a special case of predicate detection on the extended state space where the predicate to be detected consists only of the *crashed* variables of processes. This highlights the close relationship between failure detection and predicate detection which is studied in the following sections.

Note that the meaning of "ϕ holds in the computation" corresponds to the detection modality *possibly*(ϕ) [7, 13]. Detecting *possibly*(ϕ) involves constructing the entire computation lattice in the general case. The lattice represents all possible observations; hence, an observation is a path through the lattice. For simplicity we restrict our attention to *observer-independent predicates* [4]. For these types of predicates it is sufficient to construct a single observation, i.e., a single path through the lattice, to check whether ϕ holds in the observation. For example, stable predicates are observer-independent (a predicate is stable iff once it holds it holds forever). However, not all observer independent predicates are stable. For example, predicates which are local to a single process are observer-independent but may not be stable.

4 Impossibility of Predicate Detection in a Faulty Environment using Failure Detectors

In this section we show that predicate detection cannot be solved with any failure detector in the sense of Chandra and Toueg [3]. This is because failure detectors are "functions of failures", i.e., the failure detector $\mathcal{D}$ is a function which maps a failure pattern F to some element of an arbitrary range $\mathcal{G}$. The proof is based on the assumption that apart from using failure detectors and (asynchronous) messages, no information can flow between processes. Messages sent by application processes to monitor processes for the sake of predicate detection are called *control messages*. The impossibility holds even if we assume that state changes on the application processes and the broadcast of control messages happen atomically.

Theorem 1. *It is impossible to solve predicate detection with any failure detector $\mathcal{D}$.*

Proof. The proof is by contradiction and thus assumes that there exists an algorithm A which solves predicate detection using some failure detector $\mathcal{D}$. Now consider a run $R_1 = (F, \mathcal{D}(F), I, S_s, S_p)$ in which p crashes without executing a single step and consider A detecting the predicate $\phi \equiv$"p crashed in initial state". Since A solves predicate detection, A will eventually detect ϕ (e.g., at time t_1). Now consider a run R_2 with the same failure pattern F, but different S_s and S_p where p executes a step s_1 before it crashes. Since A is correct and ϕ never holds, A will never detect ϕ. Since we assume that the only means of communication are control messages, A must receive the message m about the

occurrence of s_1 in R_2 before t_1. If it would receive m later, A must act like in R_1 since it has no means to distinguish R_1 and R_2 (not even the failure detector can help here). But postulating that m is received before t_1 violates the asynchrony of communication, a contradiction.

In the next section we consider a way of circumventing the impossibility by extending the concept of a failure detector to a component which we call a *failure detection sequencer*.

5 Failure Detection Sequencers

The failure detector abstraction was introduced by Chandra and Toueg [3] to characterize different system models with respect to the solvability of problems in fault-tolerant computing. We take a similar approach and devise an oracle that encodes enough information to solve predicate detection in asynchronous systems with process crashes. As shown in the previous section, information about failures alone is not sufficient. Hence, our oracle also needs to provide information about the state of the process at the moment this process has crashed.

5.1 Definition

We now define a *failure detection sequencer* Σ, which consists of a set of passive modules, one for each monitor process. The sequencer can be queried by the monitor and returns an array of size n. The value at index i of the array is either $\perp$ or contains a predicate φ on the local state of process p_i. Informally spoken, the latter means that p_i has crashed and that its final state satisfied φ. The predicate φ may have different forms, e.g., indicate a unique sequence number of the step last performed by p_i. Let $\mathcal{A}$ denote the set of all possible array values, i.e., combinations of $\perp$ and local predicates, which can be returned by Σ. Formally, Σ is defined as follows:

A *sequencer history* H_Σ is a mapping from $\Phi \times \mathcal{T}$ to $\mathcal{A}$. The value of $H_\Sigma(m,t)$ indicates the return value of Σ at monitor b if it is queried at time t. If $H_\Sigma(m,t)[i] = s$, then b suspects p_i at time t to be in s ($s \neq \perp$). A *failure detection sequencer* Σ maps a failure pattern F, a step function S_s and a process function S_p to a set of sequencer histories.

Given a time t, the most recent step of a process p_i can be determined by inspecting S_s and S_p. If p_i has not executed any step, then the most recent step is denoted by ϵ. Formally, the *most recent step (mrs) of p_i at t given S_s and S_p* is s iff

$$mrs(p_i, t, S_s, S_p) : \exists t' \leq t.(S_s(t') = s) \wedge (S_p(t') = p_i) \wedge (\forall t''.t' < t'' < t.S_p(t'') \neq p_i)$$

We require that the set of all possible sequencer histories H_Σ satisfies the following two properties:

- (Accuracy) No process is incorrectly suspected to be in state s. Formally:

$$\forall m.\forall p_i.\forall t.H_\Sigma(m,t)[i] = s \neq \perp \Rightarrow p_i \in F(t) \wedge (s = mrs(p_i, t, S_s, S_p))$$

– (Completeness) If p crashes, then eventually every monitor will permanently suspect p to be in some state. Formally:

$$\forall m.\forall p_i.\forall t.p \in F(t) \Rightarrow \exists t' \geq t.\forall t'' \geq t'.H_\Sigma(m,t'')[i] \neq \perp$$

Since the accuracy requirement has a conjunction in the consequent, it is possible to separate it into a step accuracy part and a crash accuracy part. Crash accuracy corresponds to *strong accuracy* of Chandra and Toueg [3] ("no process is suspected before it crashes"), while step accuracy would mean that a non-$\perp$ sequencer output for process p_i at time t always equals the state which p_i is in *at the same moment* (i.e., at time t). Clearly, this property has only trivial solutions (i.e., a solution which always outputs $\perp$) since asynchronous message passing does not allow instantaneous message delivery. However, the combination of step accuracy and crash accuracy makes sense again since crashes "freeze" the state of a process so that there is no danger of state change once the sequencer has suspected that process.

We have called the new device a "sequencer" because it allows to implement causal order on failure detection events. Indeed, using Σ it is possible to infer the state of a process at the moment it is suspected. This means that it is possible to know how many control messages are in transit. Hence, the "delivery" of the suspicion can be delayed until all causally preceding events have been delivered; Σ can be used to "sequence" crash notifications, as shown in the following section.

5.2 Equivalence to Predicate Detection

Now we investigate the power of failure detection sequencers and show that they are sufficient and necessary to solve predicate detection. First we consider sufficiency.

The idea of implementing predicate detection using Σ is to embed crash events consistently into the causal order $\rightarrow$ of events in a computation. For this purpose, the algorithm shown in Figure 1 uses *causal broadcast* [2] (using primitives *c-bcast* and *c-deliver*) to disseminate information about state changes to all monitors and to withhold issuing the crash occurrence when Σ suspects p_i after some state s until the state of p_i has indeed reached state s. This is done using a vector *def_crash*[i] (for "deferred crash").

The adaptiveness (i.e., the ability to "restart" predicate detection via *fork*) of predicate detection is implemented by using a variable *history*, which stores the sequence of global states. Whenever a new predicate ϕ is issued using the *fork* command, the entire history is checked whether or not ϕ held in the past.

Theorem 2. *Predicate detection can be solved using Σ.*

Proof. Proving the safety property of predicate detection requires to show that every state constructed by the algorithm in Figure 1 is a consistent global state over the extended state space of the application. Similarly, the liveness property can be proven by showing that once ϕ holds in the application, eventually every

monitor will construct a corresponding global state (this is where the completeness property of Σ is needed).

We now show that Σ is necessary to solve predicate detection. To do this we assume the existence of an abstract algorithm *PD* that solves predicate detection on a given computation. Then we give an algorithm that emulates the output vector of Σ using *PD*.

Similar to the predicate detection algorithm in Figure 1 we instruct application processes to send a control message to all monitors if a local event happens. These control messages are used to fork an increasing number of instances of *PD*. Initially, a single instance for the predicate "p_i crashed in initial state" is started for every process p_i. When the first control message (i, s) arrives, a new instance is *fork*ed for the predicate "p_i crashed in state s". This is done whenever a new control message arrives.

The *output* vector which simulates the output of Σ is initialized with $\perp$ values and only changed, if one of the instances of predicate detection terminates by issuing *detected*(ϕ). This indicates that a process crashed in some state. The algorithm reflects this by changing the corresponding entry in *output*. The change is permanent since the state in which a process crashes does not change anymore.

Theorem 3. *If predicate detection is solvable, then Σ can be implemented.*

Proof. The accuracy property of Σ follows directly from the safety property of the predicate detection algorithm. Exploiting the adaptiveness of predicate detection allows us to show the completeness property of Σ.

It is interesting to study the role of adaptiveness in the proof of Theorem 3. For example, consider a definition of predicate detection without adaptiveness, i.e., it is merely possible to start instances of *PD* at the beginning of the computation. Not knowing the way in which the computation will proceed, it is necessary to invoke an instance of predicate detection for *every* state a process may reach. Hence, non-adaptive predicate detection can be used to implement Σ as long as the state space of a process is finite. Adaptiveness allows to invoke instances of predicate detection "on demand". This means that — given infinite state space — while there is no bound on the number of calls to *fork*, the number of concurrent instances of predicate detection is always finite.

The following theorem is an immediate consequence of Theorems 2 and 3. It can be rephrased as showing that Σ is the "weakest failure detector" for solving predicate detection. The quotation marks are important, because from Theorem 1 we know that we should not call Σ a failure detector.

Theorem 4. *Solving predicate detection is equivalent to implementing Σ.*

6 Implementing Σ

The sequencer Σ is a rather strong device and its strength makes it a highly desirable tool in crash-affected systems. Hence, the question naturally arises on

```
1  On every application process p_i:
2    ⟨whenever a state change from s to s' happens⟩ do
3      c-bcast (i, s) to all monitors
4  On every monitor process b_j:
5    variables:
6      state[1..n] of ⟨local state information⟩ init ⟨initial states of processes⟩
7      crashed[1..n] of boolean init false
8      def_crashed[1..n] of {⊥} ∪ ⟨local state information⟩
9      history sequence of ⟨(state, crashed)⟩ init ⟨initial state⟩
10     S set of ⟨global predicates⟩ init ∅
11   algorithm:
12     do forever
13       case ⟨next event⟩ of            {* three cases possible *}
14       case 1: ⟨(i, s) is c-delivered⟩
15         state[i] := s
16         history := history · (state, crashed)
17         if ∃φ ∈ S.φ(state, crashed) then detected(φ) endif
18         if def_crash[i] = state[i] then
19           crashed[i] := true
20           history := history · (state, crashed)
21           if ∃φ ∈ S.φ(state, crashed) then detected(φ) endif
22         endif
23       case 2: ⟨Σ suspects p_i in s⟩
24         if state[i] = s then
25           crashed[i] := true
26           history := history · (state, crashed)
27           if ∃φ ∈ S.φ(state, crashed) then detected(φ) endif
28         else {* state[i] ≠ s *}
29           def_crash[i] := s
30         endif
31       case 3: ⟨fork(φ) is called⟩
32         S := S ∪ {φ}
33         if ∃s_k ∈ history.φ(s_k) then detected(φ) endif
34       end {* case *}
35     end {* do forever *}
```

Fig. 1. Solving predicate detection using Σ. The primitives *c-bcast* and *c-deliver* denote causal broadcast and causal message delivery, respectively. The operator $\cdot$ denotes concatenation of sequences. Furthermore, the choice of the case statement is supposed to happen in a fair manner (e.g., event handling is performed using first-come first-serve).

```
1  On every application process p_i:
2    ⟨whenever a state change (s, s') happens⟩ do
3      c-bcast (i, s) to all monitors
4  On every monitor process b_j:
5    variables:
6      output[1..n] of {⊥} ∪ ⟨process state information⟩ initially ⊥
7    algorithm:
8      for all i ∈ {1, ..., n} do begin
9        fork("p_i crashed in initial state") end
10     do forever
11       ⟨wait until (i, s) is c-delivered or detected(φ) is invoked⟩
12       if ⟨(i, s) was c-delivered⟩ then
13         fork("p_i crashed in state s")
14       elsif ⟨detected(φ) was called⟩ then
15         {* φ is "p_i crashed in s" *}
16         output[i] := s
17       endif
18     end {* do forever *}
```

Fig. 2. Emulating Σ using a predicate detection algorithm. State changes and sending control messages on application processes is assumed to happen atomically. Event handling in line 11 is again performed in a fair manner, e.g., using first-come first-serve.

how to implement Σ in "real" systems. First, consider *synchronous systems*, i.e., systems where bounds on message delivery delays and relative processing speeds exist. In synchronous systems, Σ can easily be implemented, for instance, by the algorithm in Figure 3. This algorithm is a variant of the algorithm for implementing a perfect failure detector in synchronous systems presented by Tel [15]. With every local step, process p_i decrements a special timer variable r, one for every remote process. Upon message reception from process p_j $(j \neq i)$, the timer is reset to the in-ital value δ, which is computed from the maximum message delivery delay and the difference in relative processing speed. If p_i fails to receive a message from p_j before the timer elapses, then p_j is suspected by p_i.

To see that the algorithm indeed implements Σ, we need to show that it satisfies the accuracy and completeness properties given in Section 5.1. The proof of the completeness property is the same as for perfect failure detectors. To see that the accuracy property is satisfied, consider the sequence of "alive" messages received by Σ. As these messages are sent and arrive in FIFO order[3], the failure detector also receives the correct sequence of state information. If p_j crashes, the final message received by p_i $(i \neq j)$ is also the final message which was sent by p_j. This implies that the state information given in that message is a true indication of the most recent step performed by p_j.

[3] FIFO broadcasts are implementable in synchronous systems, as they can even be implemented in asynchronous systems [12].

```
1  On every process p_j:
2    with ⟨every step⟩ FIFOsend "alive in state s" to all
3  On every process p_i:
4    variables:
5      D_i[1..n] init (⊥, ..., ⊥) {* sequencer output *}
6      r_i[1..n] init (δ, ..., δ) {* timers *}
7      S_i[1..n] init ⟨initial states of p_1, ..., p_n⟩
8    algorithm:
9      upon FIFOreceive "alive in state s" from p_j do
10       ⟨reset timer r_i[j] to δ⟩
11       S_i[j] := s
12     upon ⟨expiry of timer r_i[j]⟩ do
13       D_i[j] := S_i[j]
```

Fig. 3. Implementing Σ in synchronous systems. The value δ is a local timeout value computed from the global boundary on message delivery delay and relative processing speeds.

Theorem 5. *In a synchronous system the output of the algorithm in Figure 3 satisfies the accuracy and completeness conditions of Σ.*

Note that in general the entire state of the crashed process needs to be delivered by the failure detection sequencer. The efficiency of the implementation can be improved by taking into account the semantics of the predicate and the application (e.g., by delivering only references to certain predefined states).

Now consider a system without bounds on relative process speeds but bounded communication delays (i.e., asynchronous processes with *synchronous* communication). In such systems, Σ is implementable if any $\mathcal{D} \in \mathcal{P}$ is given. The algorithm is shown in Figure 4 and is similar to the one in Figure 3. Here the timing bound δ refers to the synchrony of the communication channels. Completeness is achieved through the completeness of $\mathcal{D}$ and the fact that the timer eventually runs out. Accuracy is satisfied because of the accuracy of $\mathcal{D}$, the FIFO property of messages (as above), and the fact that after expiry of the timer, no message can be in transit (bounded communication delays).

Theorem 6. *In a system with asynchronous processes, synchronous communication, and any $\mathcal{D} \in \mathcal{P}$, the output of the algorithm in Figure 4 satisfies the accuracy and completeness conditions of Σ.*

We discuss the relationship between perfect failure detectors and Σ in more detail in the following section.

7 Discussion

We have shown that predicate detection cannot be solved with a perfect failure detector. However, it is solvable using failure detection sequencer Σ. In a sense

```
8    algorithm:
9      upon FIFOreceive "alive in state s" from p_j do
10       S_i[j] := s
11     upon ⟨D suspects p_j⟩ do
12       ⟨reset timer r_i[j] to δ⟩
13     upon ⟨expiry of timer r_i[j]⟩ do
14       if⟨D suspects p_j⟩ then
15         D_i[j] := S_i[j]
16       endif
```

Fig. 4. Implementing Σ using $\mathcal{D} \in \mathcal{P}$ and synchronous communication (lines 1 to 7 are the same as in Figure 3). The value δ is a local timeout value computed from the global boundary on message delivery delay.

this means that Σ is "stronger" than a perfect failure detector. Since both abstractions can be implemented in synchronous systems, a perfect failure detector seems to "loose" some information that a sequencer retains at its interface. In this context, two questions arise which we now discuss: (1) How can this difference in information be characterized, and (2) how much information (if any) does a sequencer loose compared to a fully synchronous system?

Regarding question (1), it seems that the synchrony of communication is the aspect which Σ (in contrast to perfect failure detectors) encapsulates. Consider for example an additional oracle Δ which can be queried whether or not the communication channel to a process p_j is empty. Both oracles, Δ and any $\mathcal{D} \in \mathcal{P}$, are incomparable, since they cannot emulate each other in asynchronous crash-affected systems. However, using Δ instead of the timeout mechanism in the algorithm of Figure 4 yields Σ. Hence, knowledge about the synchrony of communication channels is all that is needed to strengthen a perfect failure detector to Σ. Conversely, this information can be regarded as being "lost" at the interface of a perfect failure detector.

Regarding question (2), we now argue that Σ retains the full information present in synchronous systems. Using Σ, it is possible to implement a *synchronizer* [1] for asynchronous crash-affected systems. A synchronizer is a distributed algorithm that allows asynchronous processes to proceed in *rounds*. For this, the synchronizer generates a sequence of clock-pulses [1] at each process which separate the rounds. With every pulse, a process is allowed to send at most one message to one of its neighbors. The synchronizer ensures that all messages sent at the beginning of round r are received within round r. It also ensures that every correct process (i.e., a process that does not fail) participates in infinitely many rounds.

Since the failure detection sequencer makes it possible to identify the final message from a crashed process, it is possible to implement such a synchronizer just like in the fault-free case [15, p. 408]: At the beginning of round r, every surviving process sends exactly one message m_r to every other process (using reliable broadcast [12]). The application message which the process might send

in round r is associated with this synchronizer message to form a single message. A process p_i waits until, for every other process p_j, either (a) m_r is received or (b) Σ suspects p_j. Note that in the latter case it is possible to distinguish the two cases where p_j crashed before or after sending the message m_r. (This distinction is not possible with a perfect failure detector.) Waiting for m_r is important in order to satisfy the specification of the synchronizer, as no other way exists to prevent application messages from round r to be received in round $r+1$ or later.

The pulses generated by the synchronizer resemble a form of global logical time. Such a time is present in synchronous systems and so the synchronizer transforms the asynchronous system into a synchronous system, with the exception of global real time. In other words, time-free applications [5] perceive an asynchronous system augmented with Σ as equally strong as a synchronous system. Hence, Σ can be regarded as a form of failure detector which offers applications full synchrony without referring to a global clock.

As Σ is equivalent to a perfect failure detector which suspects a crashed process only if there are no messages in transit from that process, one can argue that Σ could be specified based on these properties. However, we feel that such a specification makes assumptions about the implementation of failure detectors, although most implementations will indeed be based on message reception (or the lack thereof). Our specification, in contrast, is only based on the state of the process and does not refer to a particular implementation.

8 Future Work

Many open issues for future work remain: For instance, can other protocols (like those used for solving *consensus*) exploit the additional power of failure detection sequencers to improve efficiency (e.g., in terms of message complexity)? Another interesting issue is whether other (possibly weaker) classes of failure detection sequencers are meaningful in asynchronous systems and offer more information than failure detectors. An obvious candidate would be an "eventually accurate" failure detection sequencer $\diamond\Sigma$. However, we conjecture that $\diamond\Sigma$ is equivalent to $\mathcal{P}$ with respect to the problems it allows to solve.

Acknowledgments

We wish to thank Rachid Guerraoui for his comments on an earlier version of this paper and the anonymous reviewers for their suggestions. The first author wishes to thank Ted Herman for many helpful discussions on the topic of failure detection.

References

1. B. Awerbuch. Complexity of network synchronization. *Journal of the ACM*, 32(4):804–823, Oct. 1985.

2. K. Birman and T. Joseph. Reliable communication in the presence of failures. *ACM Transactions on Computer Systems*, 5(1):47–76, Feb. 1995.
3. T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, Mar. 1996.
4. B. Charron-Bost, C. Delporte-Gallet, and H. Fauconnier. Local and temporal predicates in distributed systems. *ACM Transactions on Programming Languages and Systems*, 17(1):157–179, Jan. 1995.
5. B. Charron-Bost, R. Guerraoui, and A. Schiper. Synchronous system and perfect failure detector: Solvability and efficiency issues. In *International Conference on Dependable Systems and Networks (IEEE Computer Society)*, 2000.
6. C. M. Chase and V. K. Garg. Detection of global predicates: Techniques and their limitations. *Distributed Computing*, 11(4):191–201, 1998.
7. R. Cooper and K. Marzullo. Consistent detection of global predicates. *ACM SIGPLAN Notices*, 26(12):167–174, Dec. 1991.
8. V. K. Garg and J. R. Mitchell. Distributed predicate detection in a faulty environment. In *Proceedings of the 18th IEEE International Conference on Distributed Computing Systems (ICDCS98)*, 1998.
9. F. C. Gärtner and S. 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, Oct. 2000. IEEE Computer Society Press.
10. F. C. Gärtner and S. Pleisch. (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, pages 98–113, Lisbon, Portugal, Oct. 2001. Springer-Verlag.
11. F. C. Gärtner and S. Pleisch. Failure detection sequencers: Necessary and sufficient information about failures to solve predicate detection. Research Report RZ 3438, IBM Research Laboratory, Zurich, 2002.
12. V. Hadzilacos and S. Toueg. A modular approach to fault-tolerant broadcasts and related problems. Technical Report TR94-1425, Cornell University, Computer Science Department, May 1994.
13. K. Marzullo and G. Neiger. Detection of global state predicates. In *Proceedings of the 5th International Workshop on Distributed Algorithms (WDAG91)*, pages 254–272, 1991.
14. F. Mattern. Virtual time and global states of distributed systems. In M. C. et al., editor, *Proceedings of the International Workshop on Parallel and Distributed Algorithms*, pages 215–226, Chateau de Bonas, France, 1989. Elsevier Science Publishers. Reprinted on pages 123–133 in [17].
15. G. Tel. *Introduction to Distributed Algorithms*. Cambridge University Press, second edition, 2000.
16. S. Venkatesan. Reliable protocols for distributed termination detection. *IEEE Transactions on Reliability*, 38(1):103–110, Apr. 1989.
17. Z. Yang and T. A. Marsland, editors. *Global States and Time in Distributed Systems*. IEEE Computer Society Press, 1994.
18. P. yu Li and B. McMillin. Fault-tolerant distributed deadlock detection/resolution. In *Proceedings of the 17th Annual International Computer Software and Applications Conference (COMPSAC'93)*, pages 224–230, Nov. 1993.