Consistent Detection of Global Predicates Under a Weak Fault Assumption

Felix C. Gärtner*
Technische Universität Darmstadt
Fachbereich Informatik
D-64283 Darmstadt, Germany
felix@informatik.tu-darmstadt.de

Sven Kloppenburg
Systeam Engineering GmbH
Donnersbergring 15
D-64285 Darmstadt, Germany
sven@syseng.de

Abstract

We study the problem of detecting general global predicates in distributed systems where all application processes and at most $t < m$ monitor processes may be subject to crash faults, where m is the total number of monitor processes in the system. We introduce two new observation modalities called negotiably and discernibly (which correspond to possibly and definitely in fault-free systems) and present detection algorithms for them which work under increasingly weak fault assumptions.

1. Introduction

There has been relatively little work on observation techniques in systems where faults are allowed to occur. Naturally, introducing faults into the system introduces additional complexity which is reflected in an additional difficulty of answering questions related to predicate detection. In previous work, Garg and Mitchell [8] allow for process and message channel crashes, but they tackle the additional difficulties of predicate detection by restricting the class of predicates. Briefly spoken, the detection algorithm can only detect predicates which are *set-decreasing*. Informally, a predicate is set-decreasing if whenever it holds for a set of processes, it also holds for a subset of these processes. For example, "no token" is a set-decreasing predicate. Set-decreasing predicates may be detected by ignoring those processes which are suspected to have failed.

However, there are many interesting predicates in faulty asynchronous systems which are not set-decreasing. Therefore we take a different approach and focus on methods where the set of detectable predicates is *not* restricted. In fact, the set of detectable predicates now also includes predicates which refer to the functional state of the processes. For example, the predicate "more than two processes have crashed" is useful when validating system assumptions, but it is not set-decreasing. The idea is to find suitable equivalents for the well-known observation modalities *possibly* and *definitely* [6, 12] in asynchronous systems where a limited number of processes are allowed to crash.

It is well known that crash failures are not easy to tolerate in asynchronous systems [7] since they cannot be detected with absolute certainty [3]. Mechanisms to detect these failures can at best be unreliable [2]. As a result, we show that the problem of unreliable failure detection introduces a new source of uncertainty into the observation problem. As a consequence, the previously observer-invariant notions of *possibly* and *definitely* become observer-dependent again, since failure detection may provide different information at different observers. As a solution we introduce a new pair of observation modalities called *negotiably* and *discernibly*. These are based on two possible "global" failure detector semantics in the following sense: (1) A process p_i may be globally suspected as soon as a single observer suspects p_i (this is called *pessimistic*), or (2) a process p_i may be globally suspected only if *all* observers suspect p_i (this is called *optimistic*). Informally spoken, *negotiably*(φ) holds whenever *possibly*(φ) holds under the pessimistic global failure detector semantics. In analogy, *discernibly*(φ) holds whenever *definitely*(φ) holds under optimistic global failure detector semantics.

We show that in analogy to *possibly*, *negotiably* retains the same type of "pessimistic" semantics and thus can be used to detect bad conditions. For example, "server 1 has crashed and the load on server 2 exceeds some limit" is such a predicate. Another example is "process 1 is waiting for a message from a crashed process". These and similar types of predicates can be detected using *negotiably* and can be used to trigger recovery procedures. On the other hand, we show that in analogy to *definitely* the notion of *discernibly* has similar "optimistic" semantics and thus can be used to verify the occurence of good predicates. An example in the context of distributed transaction systems is "every process either reaches a commit point or crashes before reaching it". The examples should indicate how useful a general predicate detection mechanism is in faulty environments.

We proceed as follows: After defining our model of distributed computations and the predicate detection problem in Section 2, we briefly review the observation modalities

*This author's work was supported by the Deutsche Forschungsgemeinschaft (DFG) as part of the PhD program (Graduiertenkolleg) "Intelligente Systeme für die Informations- und Automatisierungstechnik" at Darmstadt University of Technology.

which have been proposed for fault-free systems in Section 3 and then propose two new observation modalities *negotiably* and *discernibly* in Section 4. The following two sections develop detection algorithms for these new modalities under increasingly weak fault assumptions: While the algorithms in Section 5 are correct under the assumption that all application processes but no monitors may crash, Section 6 presents algorithms which work even if at most $t < m$ monitors may crash. In this last case we show that the original semantics of *discernibly* are impossible to detect in asynchronous systems under the weakened fault assumption. We subsequently weaken the definition of *discernibly* in a way which makes its detection feasible and retains as much of its original semantics as possible. However, we also show that implementing a detection algorithm for the weakened definition requires that less than half of the monitor processes are allowed to crash. We conclude in Section 7.

2. Preliminaries

Distributed computations. We model a distributed system as a finite set of n processes $p_1, p_2, \ldots, p_n$ which communicate by sending messages to each other over channels. Each process has a local state (defined by the values assigned to its local variables) and performs state transitions according to a local algorithm. A state transition is also called an *event*. A *local history of* p_i is a sequence $e_1, e_2, \ldots$ of events local to p_i. A *distributed computation* is defined as a set of local histories, one for every process. To distinguish between the events on different processes we will sometimes superscript the event with the process number, i.e., e_i^k denotes the i-th event on process p_k.

Asynchronous systems. We assume that the system is *asynchronous*, i.e., neither are there any real-time bounds on the relative processing speeds of the processes, nor are there any bounds on the message delivery delay. It is merely assumed that a process eventually takes steps and messages are eventually delivered at the destination. While no total real-time order between events exists in such systems, it is possible to define a partial order (denoted $\rightarrow$) based on the notion of potential causality as defined by Lamport [11].

The predicate detection problem. Given some global predicate φ over the state of a distributed system, we would like to have a distributed algorithm which answers the question of whether or not φ holds throughout the computation by raising an exception "φ holds" whenever it detects φ. We formalize our goal by defining the *predicate detection problem* [4] as follows: Devise an algorithm which satisfies the following two requirements:

- (safety) the algorithm does not raise the exception unless φ holds, and
- (liveness) whenever φ holds, the algorithm will eventually raise the exception.

The assumption behind this definition is that the computation being monitored does not terminate. This is sometimes called *online predicate detection*. If it does terminate, then we additionally require that the detection algorithm raises an exception of the form "$\neg\varphi$ holds" if φ did not hold during the computation. We will concentrate on online detection in this paper since it is a more general problem than offline detection.

3. Predicate Detection in Fault-Free Systems

We assume the existence of a set of m monitor processes $b_1, \ldots, b_m$. Whenever an event occurs at some process p_k, a *control message* about this event is sent from p_k to all the monitors (the usual computation messages are called *application messages*). We are interested in *observer-independent* solutions to the predicate detection problem, i.e., it should not be the case that two monitors come to different decisions whether or not the predicate held during the computation.

A *global state* $S = (s_1, s_2, \ldots, s_n)$ is an n-tuple of local states, one for each process. In general, the predicates we wish to detect are *global predicates* meaning that they are defined on global states. A global state $S = (s_1, \ldots, s_n)$ always corresponds to a *cut* through the space-time diagram of the computation [13]. A state S is *consistent* if the corresponding cut is left-closed according to $\rightarrow$, i.e., if e_2 is part of the cut and $e_1 \rightarrow e_2$ then e_1 is also part of the cut.

Throughout a computation, a monitor continuously collects the control messages sent to it by the other processes and constructs a sequence $\Sigma = S_1, S_2, \ldots$ of global states which is called an *observation*. It is natural to instruct a monitor to observe consistently, i.e., ensure that every S_i is a consistent global state and that for every state S_i in the observation the next state S_{i+1} is the result of a single event on one process. In this case we say that S_i *precedes* S_{i+1}.

It can be shown that the set of all consistent global states together with the transitive closure of the "precedes" relation forms a lattice [1, 13]. Every consistent observation corresponds to a path through the lattice starting at the top element, so observations are not observer-independent [13]. Given some predicate φ and by using this lattice, it is possible to define two observer-independent notions of observation in the following way [6, 12]:

- *possibly*(φ) holds iff (if and only if) there exists a consistent observation $\Sigma = S_1, S_2, \ldots$ such that φ holds for some S_i in Σ.
- *definitely*(φ) holds iff for all consistent observations $\Sigma = S_1, S_2, \ldots$ there exists a state S_i where φ holds.

4. Predicate Detection in Faulty Systems

A *fault assumption* defines the way in which certain components of a system might fail [10]. A simple and widely used fault assumption is termed *crash*. In our scenario this means that any number of application processes and at most $t < m$ monitor processes are allowed to fail by simply stopping to execute steps of their local algorithm. All other components (e.g., the channels) are absolutely reliable.

4.1. Types of Predicates and Their Truth Value

In systems with crash faults, it is a natural desire to detect a class of predicates which make no sense in fault-free systems because we now additionally want to detect predicates involving the functional state of processes. For example, we could wish to detect the fact that "exactly two processes have crashed."

To formalize this, we postulate that every monitor process keeps a record of the functional state of the observed processes in an array called *up*. (We will distinguish the individual *up* arrays at different monitors by a superscript, i.e., up^j refers to the array on monitor b_j.) The correspondence is as follows: Process p_i has crashed iff $up[i] = \mathsf{false}$. Assuming $n = 3$, the predicate γ stating that "exactly two processes have crashed" can be formalized as

$$\begin{aligned}\gamma \quad \equiv \quad & (\neg up[1] \wedge \neg up[2] \wedge up[3]) \vee \\ & (\neg up[1] \wedge up[2] \wedge \neg up[3]) \vee \\ & (up[1] \wedge \neg up[2] \wedge \neg up[3])\end{aligned}$$

A question related to this way of formalization is the following: Consider a process p_i which has a local variable x initially set to 1. The predicate φ which we want to detect is $x \neq 1$. As long as p_i is up and running, it is clear that φ does not hold. However, if p_i crashes, should the detection algorithm raise an exception that φ holds or not? Interestingly, both answers make sense: For example, if the value $x = 1$ indicated the possession of a token ($x \neq 1$ could be part of a "no token" predicate), then we would be inclined to answer that the predicate holds since p_1 cannot give back the token. However, if $x = 1$ signaled a successful commit of a distributed database transaction, we would not want the algorithm to raise an exception since the opposite could lead to inconsistency issues. However, when designing a concrete detection algorithm it is necessary to make a decision as to what the default behavior of the detection algorithm should be in this case. We have chosen that it should *not* detect φ, i.e., the state of a process is assumed to remain unchanged except for its *up* value. (This decision spares us from dealing with a lot of difficult questions, because if we would assume that variables lose their values, we would have to say which value they should aquire instead.) Note that our decision about the default behavior does not restrict the types of predicates which the algorithm can detect. To see this, consider a detection predicate φ which is not detected if our default behavior is assumed (take $\varphi \equiv x \neq 1$ from above as an example). If we wish to detect φ (at the point where p_1 crashes) using our algorithm, we simply formulate it in a different way as

$$\phi \equiv \neg up[1] \vee \varphi$$

So if p_1 crashes, $\neg up[1]$ will eventually hold. Consequently, the complex predicate ϕ will also hold and thus will be detected by the algorithm.

4.2. Crash Failures and New Uncertainty

Defining the *up*-array as above imposes an additional burden on the monitor process, that of *crash failure detection*. Unfortunately, this problem is impossible to solve in asynchronous systems [3]. So failure detectors which are implementable in asynchronous systems will in general be *unreliable* [2], meaning that they can make wrong suspicions or that they can fail to detect the crash of a process alltogether.

We will call the mechanism offering this service at a single monitor a *failure detector module* and the whole set of failure detector modules a *failure detector* [2]. A failure detector module asynchronously sets the entries in $up[i]$ to false whenever it suspects process p_i to have crashed. Given the causal delivery of control messages, it is possible to relate this event to the most recent event on p_i which has been delivered to the monitor. We say that monitor b_j *suspects* p_i *after* e^i_k if $up^j[i]$ is set to false and e^i_k is the last event notification from p_i which has been delivered to b_j. In our case, the failure detector module never sets $up[i]$ to true again by itself. This is done by the monitor process whenever it receives a control message from p_i. When observing a distributed computation we can then ensure that no message is received from a process which is currently suspected as follows: If b_j suspects p_i after e^i_k and later receives a message from p_i, then this is taken as a revocation of the suspicion, i.e., $up[i]$ is set to true before processing the message from p_i. In this case we say that b_j *rehabilitates* p_i *after* e^i_k.

It is relatively easy to see that the failure detector which we have described has the following properties:[1]

- (at-most-once suspect) For all j: b_j suspects p_i after e^i_k at most once.
- (at-most-once rehabilitate) For all j: b_j rehabilitates p_i after e^i_k at most once and only if it previously suspected p_i after e^i_k.

[1] In addition to the two mentioned properties it is reasonable to assume that our failure detectors are *infinitely-often accurate* [8], a class of failure detectors which is implementable in asynchronous systems and has proven useful elsewhere [8]. Infinitely often accurate failure detectors satisfy *strong completeness* [2] (when a process p_i crashes, then eventually $up^j[i] := \mathsf{false}$ for all j) and *infinitely often accuracy* [8] (no correct process is permanently suspected). The latter is a property ensured by our way of handling control messages.

In analogy to a common way of modeling crash faults [9] it makes sense to treat *up*$[i]$ as being part of the state of p_i, resulting in the *extended state space* of process p_i. A change of *up*$[i]$ will thus be treated as the notification of an event on process p_i. Using this interpretation, b_j constructs its own computation lattice over the extended state space. However, since failure detectors are unreliable, situations can occur in which monitor b_1 suspects a process p_i while b_2 does not. Consequently b_1 and b_2 construct differing computation lattices so that detection of *possibly* or *definitely* is not observer-independent anymore.

4.3. New Observation Modalities

It is possible to define new observation modalities for the faulty case based on "global" failure detector semantics in the following way: All monitors exchange their failure detector module information and run a distributed algorithm which generates the same suspicion events on all monitors. Consequently, if a suspicion/rehabilitation event is issued by the failure detector we say that p_i *is suspected/rehabilitated after* e^i_k respectively (note that this event does not refer to a monitor anymore). These events can depend on the original suspicions/rehabilitations in two possible ways which we call optimistic and pessimistic.

Definition 1 (pessimistic failure detector) *A* pessimistic failure detector *is a failure detector satisfying the following properties:*

- *(safety) p_i is suspected (is rehabilitated) after e^i_k only if there exists an r such that b_r suspects (rehabilitates) p_i after e^i_k.*
- *(liveness) If there exists an r such that b_r suspects (rehabilitates) p_i after e^i_k then eventually p_i is suspected (rehabilitated) after e^i_k.*

Definition 2 (optimistic failure detector) *An* optimistic failure detector *is a failure detector satisfying the following properties:*

- *(safety) p_i is suspected (is rehabilitated) after e^i_k only if for all r holds: b_r suspects (rehabilitates) p_i after e^i_k.*
- *(liveness) If it holds that for all r: b_r suspects (rehabilitates) p_i after e^i_k then eventually p_i is suspected (rehabilitated) after e^i_k.*

An optimistic failure detector waits until all monitors suspect p_i before actually suspecting p_i itself. This is termed optimistic in analogy to optimistic network protocols: these protocols do things without a priori coordination in the hope that no coordination is needed. In this sense, an optimistic failure detector ignores failure suspicions in the hope they will go away again. On the other hand, a pessimistic failure detector will suspect a process p_i to have crashed as soon as at least one failure detector module at a monitor suspects p_i. This is termed pessimistic since it allows actions to be taken if there is only the slightest indication of a crash.

Since multiple suspicions and rehabilitations in between two regular events on a process offer little additional information, it is reasonable to postulate the following additional properties of optimistic and pessimistic failure detectors:

- (at-most-once suspect) p_i is suspected after e^i_k at most once.
- (at-most-once rehabilitate) p_i is rehabilitated after e^i_k at most once and only if p_i was previously suspected after e^i_k.

The events issued by the failure detector are integrated into the causal ordering as follows: Let the events s^i_k and r^i_k denote the suspicion and rehabilitation of p_i after e^i_k. Then $e^i_k \rightarrow s^i_k \rightarrow r^i_k \rightarrow e^i_{k+1}$. Based on this definition it is possible to define observer independent state lattices in a pessimistic and optimistic variant. The *pessimistic state lattice* of a computation is the original state lattice of the computation where the events of a pessimistic failure detector have been taken as normal application events. The *optimistic state lattice* uses the events generated by an optimistic failure detector. Using these concepts (and the notions of *possibly* and *definitely*), we define two new observation modalities as follows:

Definition 3 (*negotiably*(φ)) *Given a general predicate φ on the extended state space, the predicate negotiably(φ) holds iff possibly(φ) holds in the pessimistic state lattice.*

Definition 4 (*discernibly*(φ)) *Given a general predicate φ on the extended state space, the predicate discernibly(φ) holds iff definitely(φ) holds in the optimistic state lattice.*

Interestingly, *negotiably* inherits the affinity of *possibly* towards certain forms of safety properties. If a computation should satisfy "always φ", then detecting *negotiably*$(\neg\varphi)$ indicates a violation of safety (in the introduction, $\neg\varphi$ was called a "bad" predicate). For example, if the fault assumption "less than two processes have crashed" is encoded as a predicate φ, then detecting *negotiably*$(\neg\varphi)$ tells whether there is *any* evidence of φ being violated (i.e., does there exist a state lattice L such that in L exists an observation in which there exists a state where $\neg\varphi$ holds). On the other hand, *discernibly* has a similar affinity to certain forms of liveness properties as *definitely*. If a computation is required to satisfy "eventually φ", then it must satisfy *discernibly*(φ) (we called these types of predicates "good" ones in the introduction). An example for φ is "if p_1 is down then p_2 is in a certain fault-handling state". If *discernibly*(φ) holds, then every monitor will observe φ in every possible observation,

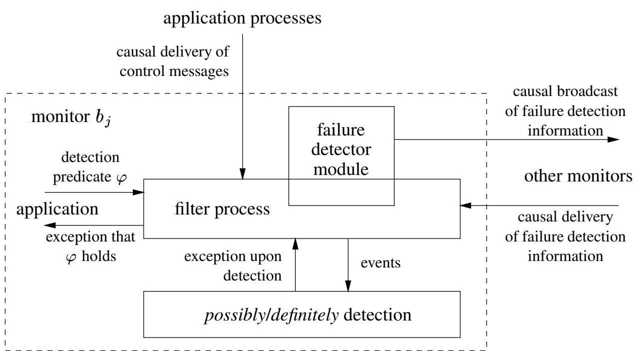

Figure 1. Architecture of the filter to build an optimistic or pessimistic failure detector.

giving the best assurance we can give about φ holding. Also note that in fault-free executions, *negotiably* and *discernibly* have the same semantics as *possibly* and *definitely*, respectively.

5. Detecting *negotiably* and *discernibly*

We now present detection algorithms for *negotiably* and *discernibly* based on existing algorithms for detecting *possibly* and *definitely* [1, 6, 13] under the fault assumption that no monitors crash. The core of the algorithm is a global failure detector (from which pessimistic and optimistic versions can be derived). The architecture is shown in Fig. 1. We now sketch the basic ideas of the algorithm.

The problem involved in building a global failure detector is the following: It must give all monitors the same failure detection information at virtually the same time. The first step is that the monitors broadcast their local failure detection information to all other monitors using a special type of control message. This message is timestamped in a special way which makes it possible to relate the event to the latest consistent global state which b_j has observed. For this, every monitor maintains two logical clocks, an *application clock* and a *monitor clock*. The application clock is the usual vector clock [13] to implement causal delivery of application control messages [10]. It is a vector of size n. Similarly, the monitor clock is a vector clock used to implement causal delivery of *failure detection control messages*. It is a vector of size m. So whenever monitor b_j suspects p_i, it broadcasts a message $((i, j, \mathit{ats}), \mathit{mts})$ to all monitors. Here, *mts* is called the *monitor timestamp* and is the current value of the monitor clock, and *ats* is called the *application timestamp* which is the current value of the application clock.

Eventually every monitor receives all such notifications. The monitor timestamp *mts* can be used to deliver the failure detection control messages in a causally consistent way. Delivering the message (i, j, ats) to the monitor means that b_j suspected p_i after $e^i_{\mathit{proj}_i(\mathit{ats})}$, where proj_i denotes the projection of a vector to its i-th component. Because of the causal delivery of failure detection notifications and the postulated failure detector properties we can show the following:

Lemma 1 *If two messages* $m_1 = (i, j, \mathit{ats})$ *and* $m_2 = (i, j, \mathit{ats}')$ *are delivered to a monitor and* $\mathit{proj}_i(\mathit{ats}) < \mathit{proj}_i(\mathit{ats}')$ *then* m_1 *is delivered before* m_2.

Hence, when a monitor receives such a notification, it knows that regarding b_j the process history prior to $e^i_{\mathit{proj}_i(\mathit{ats})}$ will not change anymore.

Now consider the following example where $m = 3$ monitors are observing a computation of $n = 2$ processes. If the failure detection event with monitor timestamp $(1, 1, 1)$ is delivered to monitor b_1 this means b_1 has now received one failure detection notification from every monitor and b_1 will not receive any other such notification that causally precedes those which it already received. Let e_1, e_2 and e_3 denote these three events from b_1, b_2 and b_3 respectively and let the application timestamps of e_1, e_2 and e_3 be $(3, 1)$, $(3, 2)$ and $(2, 3)$. Figure 2 shows the consistent global states corresponding to these timestamps within the computation lattices of the individual monitors.

By taking the componentwise minimum of the three application timestamps it is possible to derive a consistent global state which all monitors have observed already and which will not change anymore regarding *all* of the failure detector modules. This state is indicated by a small square in Fig. 2. Note that since the shaded areas of all the monitors' lattices will not change in the future, the common part of these areas can be viewed as a stable region of the lattice which is the same at all monitors. We will call this common state lattice a *settled region* and we can prove the following:

Lemma 2 *Consider one monitor* b_r, *let* $\mathit{most\_recent\_ats}[1..m, 1..n]$ *denote an array of the application timestamps contained in the most recent failure detection notifications from all monitors delivered to* b_r, *and define*

$$\mathit{common} := \min_{l=1,\ldots,m} \{\mathit{most\_recent\_ats}[l]\}$$

Then for all failure detection notifications (i, j, ats) *delivered to* b_r *holds that* $\mathit{ats} \geq \mathit{common}$.

A software layer implementing a global failure detector can buffer all incoming control messages, determine the value of *common* every time a new failure detection notification is received and then release all events in this settled region in a causally consistent way. In the same process, failure detection notifications are transformed into regular events and released according to the definition of the optimistic/pessimistic failure detector within the causal order.

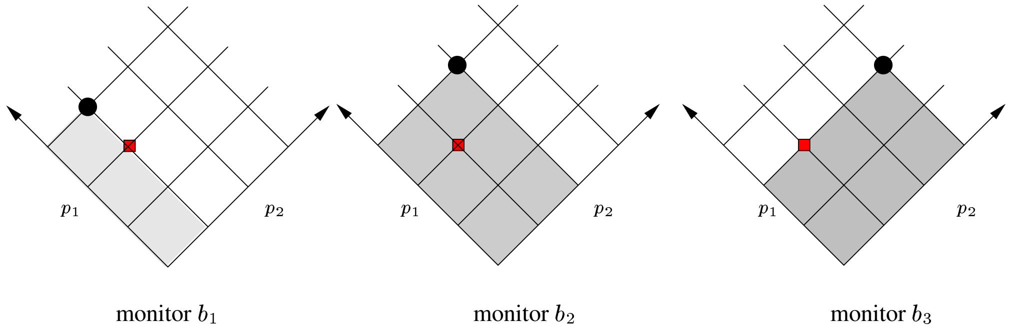

Figure 2. Consistent states corresponding to three application timestamps. The square indicates the states common to all monitors which are determined by the software layer.

A problem is that the settled area can only advance whenever failure detection notifications are received. Since this might be rare in practice, it may be necessary for failure detector modules to periodically send out empty dummy messages stating that the failure detector value has not changed at monitor b_j. So while the software layer adds "detection latency", it does not induce much additional computation overhead (apart from buffering). A disadvantage of this detection scheme is that it was constructed under the assumption that monitors themselves never crash which is a very strong assumption. We will address this point in the following section.

For completeness we have depicted pseudocode of the detection algorithm of *discernibly* in Fig. 3. The code of the detection algorithm for *negotiably* is similar. Building upon Lemma 2 we can show that the events released in line 40 in fact represent the optimistic state lattice. Upon these events we use one of the published algorithms to detect *definitely*, so overall we can show:

Theorem 1 *If no monitors crash then the algorithm in Fig. 3 solves the predicate detection problem for discernibly*(φ).

6. Detecting *negotiably* and *discernibly* When Monitors May Crash

In this section we reconsider the notions of *negotiably* and *discernibly* under a weaker fault assumption, i.e., we assume that monitor processes too may be subject to crash faults. This should make the notions of *negotiably* and *discernibly* more realistic since in reality monitor processes often reside on the same physical nodes as the application processes and so they do not fail independently. More precisely, we will assume that at most $0 < t < m$ monitor processes may fail, i.e., $m-t$ will remain always functional.

6.1. Detecting *negotiably* When Monitors May Crash

Detecting *negotiably* means to detect *possibly* and assume the crash of a process as soon as a single monitor suspects this process. Obviously the semantics still make sense as long as a single monitor stays alive (i.e., $t \leq m-1$). However, it is not possible to use the detection algorithm sketched in Sect. 4 anymore. To see this, consider again Fig. 2 and assume that b_3 crashes after it has issued event e_3. Since the settled region can only advance if b_1 and b_2 eventually receive information from b_3, the settled area cannot advance over the state in which b_3 has crashed. Since reliable failure detection is impossible in our system model, it is not possible for b_1 and b_2 to determine whether or not they will eventually receive something new from b_3. Hence, the algorithm may fail to detect predicate occurrences which happen at later global states.

Interestingly, it is still possible to detect *negotiably* using a different algorithm. The idea of the algorithm is as follows: Monitors broadcast their failure detection events as usual to all other monitors and timestamp them in the same way as described in Sect. 6.1. The software layer transforms a failure detection event (i, j, ats) into a regular event notification "$up[i] := \mathsf{false}$", however, it does not hold back any events but passes them through directly to the detection algorithm for *possibly*. This extinguishes all sources of blocking progress of the detection algorithm which can go about to incrementally construct the state lattice.

The problem is that failure detection messages can arrive "late", i.e., if b_j suspects p_i after e^i_k then this message may arrive at some other monitor b_r after the event e^i_{k+1} was delivered to b_r. This is because the causal relation between the suspicion and the event e^i_{k+1} is artifical and not based on notions which could be detected using mechanisms like vector clocks. So these late transitions must be integrated into the "computational past" of processes, i.e., we have to change the *internals* of the state lattice and not the edges. However, integrating belated suspicions generates a new set of global states in the lattice which of course must be taken into consideration when detecting *possibly*.

As an example, assume monitor b_1 has already received a notification about event number 3 on process p_1 and then learns that some other monitor b_2 suspected p_1 to have

```
On every monitor process b_j:

  variables:
    mc array [1..m] of ℕ init (0, ..., 0)                                   {* monitor clock *}
                                          {* used for causal broadcast of failure detector messages *}
    ac array [1..n] of ℕ init (0, ..., 0)                                   {* application clock *}
                                                             {* keeps current vector time of monitor *}
    up array [1..n] of {true, false} init (true, ..., true)
    most_recent_ats array [1..m, 1..n] of ℕ init ((0, ..., 0), ..., (0, ..., 0))
                           {* keeps track of application timestamps of latest messages from all monitors *}
    app_events set of "state change" × ℕ^n init ∅                         {* buffer of application events *}
                                                                       {* with application timestamps *}
    fd_events set of {1, ..., n} × {1, ..., m} × ℕ^n init ∅
                                  {* (i, j, ats) ∈ fd_events means b_j suspected p_i at b_j-time ats *}
    common[1..n] of ℕ init (0, ..., 0)                                        {* settled area *}

  process failure_detection_module:
    upon ⟨up[i] is set to true or delivery of a message from p_i⟩ do
      ⟨notify failure detection service⟩
    upon ⟨suspicion of process p_i by the failure detection service⟩ do
      up[i] := false;                           {* causally broadcast detection events among monitors *}
      mc[j] := mc[j] + 1;
      send ((i, j, ac), mc) to all monitors
    periodically do                                 {* send periodic updates even if nothing changed *}
      mc[j] := mc[j] + 1;
      send ((0, 0, ac), mc) to all monitors

  process filter:
    upon ⟨delivery of (e, ats) from p_i⟩ do                      {* a normal control message arrives *}
      app_events := app_events ∪ (e, ats);
      if ∃(up[i] := false, ats') ∈ app_events ∧ proj_i(ats') = proj_i(prev_i(ats)) then
                                                                     {* if p_i was previously suspected *}
        app_events := app_events ∪ (up[i] := true, prev_i(ats));                  {* rehabilitation *}
      end
    upon ⟨delivery of (i, j, ats) from b_j⟩ do                 {* failure detection notification arrives *}
      if (i, j) ≠ (0, 0) then
        fd_events := fd_events ∪ (i, j, ats);
        if ∀k ∈ {1, ..., m}.∃(i, k, ats) ∈ fd_events then                   {* extract optimistic data *}
          app_events := app_events ∪ (up[i] := false, ats);
      end
      most_recent_ats[j] := ats;
      common := min_{l=1,...,m}{most_recent_ats[l]}                           {* determine settled area *}
      ⟨release all events from app_events prior to common to det. mod. in causal order⟩
    upon ⟨exception "definitely(φ) holds" from detection module⟩ do
      raise exception "discernibly(φ) holds" at application
```

Figure 3. Pseudocode of the different modules of the detection algorithm for *discernibly*. The function $prev_i(x)$ takes an application timestamp x and returns the same timestamp in which the i-th value has been decremented by one. The function $proj_i(x)$ is simply the projection to the i-th value of vector x.

crashed after event number 2. If the detection predicate φ is "p_1 crashed after event 2" then neglecting the new failure detection event would lead to a failure to detect φ. Hence, it must be possible to "reset" or rollback the *possibly* detection algorithm so that it takes these new states into consideration. If this can be done, the algorithm will detect *possibly*(φ) and hence also *negotiably*(φ), so it satisfied the liveness requirement of the predicate detection problem. Fig. 4 shows pseudocode of the software layer.

However, during the computation the monitors may often perform *possibly* detection on incomplete information. If this could lead the algorithm into a premature detection of some predicate (which would not be detected on complete information) our detection schema would violate the safety condition of the predicate detection problem. Fortunately, this cannot be the case.

Lemma 3 *Let L be a non-empty computation lattice and let L' be the lattice which is constructed from L by inserting exactly one failure detection and a following rehabilitation event on one axis of the lattice. Then there exists no state predicate φ over the extended state space for which possibly(φ) holds in L and $\neg$possibly(φ) holds in L'.*

Proof. Because *possibly*(φ) holds in L there exists a consistent global state S in L for which φ holds. Since L' is constructed from L by merely adding states, S is also a state in L'. Hence, *possibly*(φ) holds in L'. □

Theorem 2 *If at most $0 < t < m$ monitors crash, the algorithm depicted in Fig. 4 solves the predicate detection problem for negotiably(φ).*

6.2. Detecting *discernibly* When Monitors May Crash

In contrast to the detection of *negotiably*, it is not possible to adapt the detection algorithm of *discernibly* so easily to the weaker fault model. This is because belated failure detection notifications can actually lead to false detections of predicates in the state lattice. The intuitive reason for this is that the detection of *discernibly*(φ) depends not only on the set of states but also on their placement within the lattice.

To see this consider the example depicted in Fig. 5. It consists of two processes: p_1 has a local variable x and process p_2 has a local variable y, both initially 0. Both processes keep on incrementing their variables successively. A monitor b_1 observing this computation will construct the computation lattice which is depicted on the left side of Fig. 5. In this computation we want to detect *discernibly*(φ) where φ is defined as

$$\varphi \equiv up[1] \wedge up[2] \wedge (x = 1 \vee y = 1)$$

We have marked all consistent global states where φ holds with a black dot. Since every computation path passes through such a state, *definitely*(φ) holds and, hence, *discernibly*(φ) also holds by definition and should be detected. However, suppose that there are still failure detection notifications in transit which arrive "late" and cause the introduction of additional events into the computation lattice as shown in the right side of Fig. 5. Now there is in fact a computation path (indicated by a dashed line) for which φ does not hold anywhere. Consequently, *definitely*(φ) does not hold for this lattice and, hence, *discernibly*(φ) does not hold either.

The example shows that a detection algorithm which does not wait for *all* undelivered failure detection notifications can cause false detections of *discernibly*. However, since $t < m$ monitors may crash, to know whether or not to wait for such messages from other monitors would require accurate failure detection, which again is impossible in our system model. We conclude that detecting *discernibly* is impossible under the weaker fault model. Instead, we propose the following definition which is similar and which allows again to define the notion of a settled area within a computation lattice.

Definition 5 (t-optimistic failure detector)
A t-optimistic failure detector *is a failure detector satisfying the following properties:*

- *(safety) p_i is suspected (is rehabilitated) after e^i_k only if for at least $m - t$ monitors b_r holds: b_r suspects (rehabilitates) p_i after e^i_k.*
- *(liveness) If for at least $m - t$ monitors b_r holds that b_r suspects (rehabilitates) p_i after e^i_k then eventually p_i is suspected (rehabilitated) after e^i_k.*

A t-optimistic failure detector can again be used to define a t-optimistic computation lattice. So the definition of *discernibly* changes to:

Definition 6 (t-*discernibly*(φ)) *Given a general predicate φ on the extended state space, the predicate t-discernibly(φ) holds iff definitely(φ) holds on a t-optimistic computation lattice.*

While the original definition of *discernibly* requires *all* monitors to suspect a process to globally declare it crashed, the definition of t-*discernibly* only requires at least $m - t$ monitors to suspect that process. Since our fault assumption ensures that at most t monitors crash, there are always at least $m - t$ monitors up and running. Hence, it is possible to define a settled area in the computation lattice as soon as the information of at least $m - t$ monitors has been gathered. So we can use a detection algorithm similar to the one for detecting *discernibly* presented in Sect. 4.

```
On every monitor process b_j:

  variables:
    mc array [1..m] of ℕ init (0, ..., 0)                                   {* monitor clock *}
                                {* used for causal broadcast of failure detector messages *}
    ac array [1..n] of ℕ init (0, ..., 0)                               {* application clock *}
                                                 {* keeps current vector time of monitor *}
    up array [1..n] of {true, false} init (true, ..., true)
    fd_events set of {1, ..., n} × {1, ..., m} × ℕ^n  init ∅
                            {* (i, j, ats) ∈ fd_events means b_j suspected p_i at b_j-time ats *}

  process failure_detection_module:
    upon ⟨up[i] is set to true or delivery of a message from p_i⟩ do
      ⟨notify failure detection service⟩
    upon ⟨failure detection service suspects process p_i⟩ do
      up[i] := false;                   {* causally broadcast detection events among monitors *}
      mc[j] := mc[j] + 1;
      send ((i, j, ac), mc) to all monitors

  process filter:
    upon ⟨delivery of (e, ats) from p_i⟩ do
      if ∃(i, k, ats') ∈ fd_events ∧ proj_i(prev_i(ats)) = proj_i(ats') then
                                                          {* if p_i is currently suspected *}
        ⟨release event (up[i] := true, prev_i(ats) to possibly detection module⟩
      end
      ⟨release event (e, ats) to possibly detection module⟩
    upon ⟨delivery of (i, j, ats) from b_j⟩ do
      ⟨release (up[i] := false, prev_i(ats)) to possibly detection module⟩
      if proj_i(ats) < proj_i(ac) then                               {* it's a "late" event *}
        ⟨release (up[i] := true, ats) to possibly detection module⟩
        ⟨rollback possibly detection⟩
      end
    upon ⟨exception "possibly(φ) holds" from detection module⟩ do
      raise exception "negotiably(φ) holds" at application
```

Figure 4. Pseudocode of the different modules of the detection algorithm for *negotiably*.

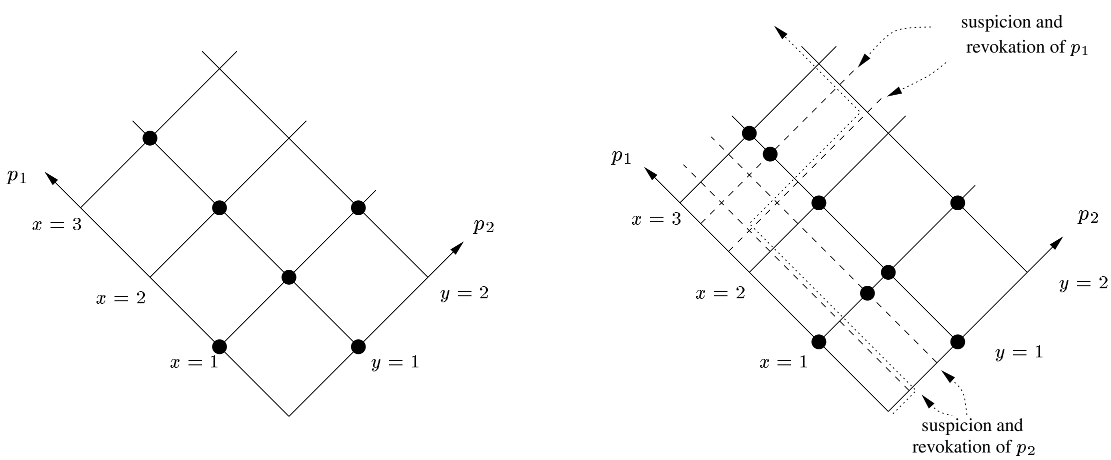

Figure 5. Example where $definitely(\varphi)$ is refuted when failure detection events arrive. The black dots mark all consistent global states where φ holds. Dashed lines indicate late detection events which have been integrated a posteriori into the lattice. The dotted line indicates an observation for which φ never holds.

To be able to declare a given area L in the computation lattice as settled, a monitor must decide on one of the following two cases for every process p_i:

- Either it has collected at least $m - t$ suspicions about a process p_i in L. In this case it may globally declare p_i as crashed and release the corresponding event to the *definitely* detection algorithm.
- Or it has collected information from at least $t + 1$ monitors that p_i is up in L. Since this implies that at most $m - (t + 1)$ monitors suspect p_i, p_i may globally be declared up and a corresponding event may be released to the *definitely* detection algorithm.

Note that a monitor *must* decide on one of these two cases for every process in order to eventually advance the settled area. However, this is sometimes not possible. For example, consider the scenario where $m = 10$ and $t = 8$. Thus, a monitor who collects two suspicions about p_i may globally declare p_i as down. Assume that the worst case has occurred and of all the monitors only b_1 and b_2 are alive. For a given computation lattice L, b_1 has suspected a process p_i and b_2 has not. Now neither monitor is able to collect either $m - t = 2$ suspicions or $t + 1 = 9$ non-suspicions, so the settled area cannot advance.

A solution to this problem is to require that one of the two cases must happen. However, this is only possible if $t + 1 \leq m - t$, i.e., it must be possible to collect $t + 1$ non-suspicions even in the worst case where $m - t$ processes crash. Simple arithmetic gives us that $t < m/2$. So to be able to detect t-*discernibly*, there must be a majority of correct monitors in the system.

7. Conclusions

Distributed predicate detection is an important practical problem in many areas of distributed computing. We have concentrated on asynchronous distributed systems and have studied the problem in a general form under the crash fault assumption. Since perfect predicate detection is impossible in these systems [3], solutions must either weaken the problem or strengthen the model. Since the asynchronous system model is attractive for many reasons [9], we have decided to accept imperfection in predicate detection and have sought ways to simplify the problem in a way that still makes sense in practice. In contrast to other work [8] we have chosen not to weaken the problem by restricting the set of detectable predicates. We have introduced two new modalities called *negotiably* and *discernibly* which can be understood in analogy to the well-known observation modalities *possibly* and *definitely* [6, 12] and which were tailored to the faulty case.

Our work is theoretical in nature but refers to very practical problems. Experience shows that distributed systems are not fault-free in practice and so observation methods for distributed computations which assume non-faulty operation fail miserably when faults actually do occur. We can only explain the lack of sufficient methodologies for the faulty case by the high additional complexity introduced even by simple fault assumptions (like the one we use). In effect, this complexity is added to a problem which is complex enough already [5]. With our work we try to show one possible way of dealing with the problem. We believe that our approach is useful since the semantics of our new predicate transformers can be understood in analogy to the existing notions of *possibly* and *definitely*. If used in fault-free environments, *negotiably* and *discernibly* have the same semantics as *possibly* and *definitely*. Also, since we still remain in the purely asynchronous system model, our methods have at least the potential of being implemented in real systems.

Acknowledgments: We wish to thank Andreas Zeidler for his comments on a previous version of this paper, and the anonynous referees for their challenging questions.

References

[1] Ö. Babaoğlu and K. Marzullo. Consistent global states of distributed systems: Fundamental concepts and mechanisms. In S. Mullender, editor, *Distributed Systems*, chapter 4, pages 55–96. Addison-Wesley, second edition, 1993.

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

[3] K. M. Chandy and J. Misra. How processes learn. *Distributed Computing*, 1:40–52, 1986.

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

[5] C. M. Chase and V. K. Garg. Detection of global predicates: Techniques and their limitations. *Distributed Computing*, 11(4):191–201, 1998.

[6] R. Cooper and K. Marzullo. Consistent detection of global predicates. *ACM SIGPLAN Notices*, 26(12):167–174, Dec. 1991.

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

[8] V. K. Garg and J. R. Mitchell. Distributed predicate detection in a faulty environment. In *Proc. of the 18th IEEE Int. Conf. on Distr. Comp. Systems (ICDCS'98)*, 1998.

[9] F. C. Gärtner. Fundamentals of fault-tolerant distributed computing in asynchronous environments. *ACM Computing Surveys*, 31(1):1–26, Mar. 1999.

[10] V. Hadzilacos and S. Toueg. A modular approach to fault-tolerant broadcasts and related problems. Tech. Report TR94-1425, Cornell Univ., Comp. Sci. Dept., May 1994.

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

[12] 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.

[13] R. Schwarz and F. Mattern. Detecting causal relationships in distributed computations: in search of the holy grail. *Distributed Computing*, 7:149–174, 1994.