Consistent Detection of Global Predicates in Asynchronous Distributed Systems with Crash Failures

Sven Kloppenburg
Systeam Engineering GmbH
and
Felix C. Gärtner
Darmstadt University of Technology

Darmstadt University of Technology
Department of Computer Science
Technical Report TUD-BS-2000-01

Feb. 29, 2000

Abstract

A fundamental problem in testing and analysis of distributed systems is detecting whether the global state of a system satisfies some predicate throughout a computation. There exist some well-developed methods for detecting predicates in fault-free systems (e.g., the predicate transformers *possibly* and *definitely* of Cooper, Marzullo and Neiger), but for systems running in faulty environments findings are rare.

We study the problem of detecting global predicates in distributed systems which may be opposed to process crashes. Process crashes introduce a new form of uncertainty—the problem of reliable failure detection. We first show that this turns the previously observer-invariant notions of *possibly* and *definitely* into observer dependent notions and thus renders these predicate transformers useless in faulty environments. Then we introduce a new pair of predicate transformers called *negotiably* and *discernibly* tailored to the new system model in the following sense: Building on the notion of *possibly*, we show that *negotiably* retains the same type of "pessimistic" semantics and thus can be used to detect the violation of special types of safety properties. In analogy to *definitely*, we show that our notion of *discernibly* has similar "optimistic" semantics and thus can be used to detect certain forms of liveness properties. We give several examples for predicates to which our new predicate transformers are useful and present algorithms to detect *negotiably* and *discernibly* in asynchronous distributed systems.

Categories and Subject Descriptors: D.2.5 [**Software Engineering**]: Testing and Debugging—*distributed debugging*

General Terms: algorithms, reliability, theory

Additional Key Words and Phrases: distributed predicate detection, predicate transformers, possibly, definitely, crash failures, failure detection, theory

---

Name: Sven Kloppenburg
Affiliation: Systeam Engineering GmbH
Address: Donnersbergring 15, D-64285 Darmstadt, Germany, Email: `sven@syseng.de`
Name: Felix C. Gärtner
Affiliation: Darmstadt University of Technology
Address: Department of Computer Science, D-64283 Darmstadt, Germany, Email: `fcg@acm.org`

1. INTRODUCTION

A distributed system is termed *asynchronous* if there are no assumed bounds on relative processing speeds or message delivery delays. For example, wide area networks like the Internet are usually assumed to be asynchronous distributed systems. While it is often possible to state timing bounds on communication parameters in other real-world networks, these bounds only hold with a probability of less than 1. In effect, this means that algorithms relying on these timing assumptions may become incorrect in some cases. Consequently, the asynchrony assumption is very attractive for the design of algorithms: If an algorithm works in an asynchronous system it will work in any other system too.

We focus on asynchronous distributed message-passing systems and study the problem of distributed predicate detection in such systems. Informally spoken, this is the problem to tell whether or not a global predicate holds throughout a distributed computation. Distributed predicate detection is an important task often encountered in software testing and analysis. For example, deadlock and termination detection of distributed computations are two instances of this problem. Other instances arise in the context of distributed debugging.

Distributed predicate detection is not an easy task. The difficulty stems from the complexity introduced by concurrency and timing uncertainty. Nevertheless, monitoring asynchronous distributed computations has been a research area for quite some time and is by now fairly well understood [Babaoğlu and Marzullo 1993; Schwarz and Mattern 1994]. For example, it can be shown that different observers monitoring the same computation can come to different conclusions about whether some predicate φ holds or not [Schwarz and Mattern 1994]. This result has lead to concepts like the predicate transformers *possibly*(φ) and *definitely*(φ) which give the question of φ holding or not a sensible, observer-invariant semantics in asynchronous distributed systems [Cooper and Marzullo 1991; Marzullo and Neiger 1991].

In almost all of the prior work the underlying system has been assumed to be fault-free. Naturally, introducing faults into the system introduces additional complexity which is reflected in an additional difficulty of answering questions related to predicate detection. In this paper, we study global predicate detection in asynchronous distributed systems in which processes may be subject to crash failures. It is well-known that it is impossible to detect a process crash in such systems [Chandy and Misra 1986]. We show that this uncertainty turns the notions of *possibly*(φ) and *definitely*(φ) into observer dependent notions again and thus makes them useless. We introduce a new pair of predicate transformers called *negotiably*(φ) and *discernibly*(φ) which have a precise and observer-invariant semantics in such faulty environments and can be understood in analogy to the prior pair of predicate transformers.

The only prior work that explicitly deals with distributed predicate detection in faulty asynchronous systems is a 1998 paper of Garg and Mitchell [1998]. In it, Garg and Mitchell allow for process and message channel crashes, but they only handle a restricted class of predicates called *set-decreasing*. Briefly spoken, 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. Restricting the class of predicates in this way circumvents many of the intricate questions related to the validity of predicates in faulty environments. This is because set-decreasing predicates may be detected by ignoring those processes which are suspected to have failed. In this paper we try to identify and answer these questions by not restricting the class of predicates we want to detect in the system.

We proceed as follows: After defining the asynchronous system model and our model of distributed computations in Section 2, we state the predicate detection problem precisely in Section 3. Then we briefly review the state of the art of distributed predicate detection in fault-free asynchronous systems in Section 4. In Section 5 we show that existing observation methodologies fail in faulty systems and identify basic mechanisms to deal with this problem. Section 6 summarizes our results and concludes the paper.

2. DISTRIBUTED COMPUTATIONS IN ASYNCHRONOUS SYSTEMS

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 eachother 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*. There are three types of events:

(1) A *send event* occurs if a process sends a message to another process via an intermediate channel.
(2) A *receive event* occurs if a process receives a message from another process.
(3) An *internal event* occurs if a process performs a state change without sending or receiving a message.

If a message is sent in event e_1 and is received in event e_2 we say that e_1 and e_2 *correspond*.

Informally, a *distributed computation* describes the execution of a set of local algorithms. A *local history of* p_i is a sequence $e_1, e_2, \ldots$ of events. A distributed computation can then formally be 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.

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. In such an asynchronous system, it is impossible to associate any total real-time order with the events within a computation. The best we can do is to associate a partial order (denoted $\rightarrow$) based on the notion of potential causality [Lamport 1978] according to the following rules:

(1) If e_i and e_j are two events on the same process p_k and $i < j$, then $e_i \rightarrow e_j$.
(2) If e_i is a send event and e_j the corresponding receive event, then $e_i \rightarrow e_j$.
(3) The relation $\rightarrow$ is transitive, i.e., if $e_i \rightarrow e_j$ and $e_j \rightarrow e_k$ then $e_i \rightarrow e_k$.

Two events which cannot be related by $\rightarrow$ are said to be *concurrent*.

A common way of visualizing distributed computations is to use *space-time diagrams*. Figure 1 illustrates such a diagram: The horizontal lines depict the local histories of the individual processes and arrows connect a send event with its corresponding receive event.

3. THE PREDICATE DETECTION PROBLEM

Given some global predicate φ over the state of a distributed system, we want to know whether this predicate holds at some point during the computation. For this we would like to have a distributed algorithm which answers the question by raising an exception "φ holds" whenever it detects φ. We formalize our goal by defining the *predicate detection problem* 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.

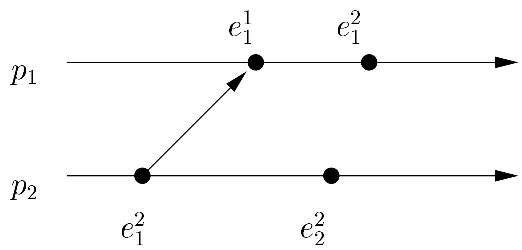

Fig. 1. A space-time diagram representing a distributed computation.

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 "φ not holds" if φ did not hold during the computation. We will concentrate on online detection in this paper. A detection algorithm for non-terminating computations can be easily transformed into one for terminating computations by using a distributed termination detection algorithm.

We wish that the question of φ holding "makes sense" in the current system model, meaning that it should never be the case that two concurrent invocations of the detection algorithm lead to different answers. As we will see in the following sections, φ takes on different forms according to different system environments. In usual non-distributed sequential computing φ will be a state predicate. In other (concurrent) environments, φ must be transformed into an adequate form by using the concept of a *predicate transformer*. What type of predicate transformer to use in different environments is sometimes as difficult to answer as it is to devise algorithms for predicate detection. Examples follow first for fault-free asynchronous systems and then for asynchronous systems where certain kinds of faults may happen.

4. OBSERVATION IN ASYNCHRONOUS SYSTEMS

Observation in distributed systems is usually based on the notion of a separate observer process called the *monitor*. Whenever an event occurs at some process p_k, a message is sent from p_k to the monitor containing information about this event. The messages sent to the monitor are called *control messages* while the usual computation messages are called *application messages*. It is assumed that control messages do not interfere with the execution of the distributed computation in any way.

4.1 Global States and Consistency

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. Formally, a global predicate φ is a function from global states to the values $\{\mathsf{true}, \mathsf{false}\}$. A global state $S = (s_1, \ldots, s_n)$ always corresponds to a *cut* through the space-time diagram of the computation, i.e., a "frontier" of events composed of all those events which have lead to state s_i on every process.

Based on the many possible combinations of local states, there are obviously many different global states. These states can be classified into consistent and inconsistent states. Informally, a state S is *consistent* if it adheres to the causality relation. Formally, 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. Figure 2 shows the computation from Figure 1 as well as cuts corresponding to consistent and inconsistent states.

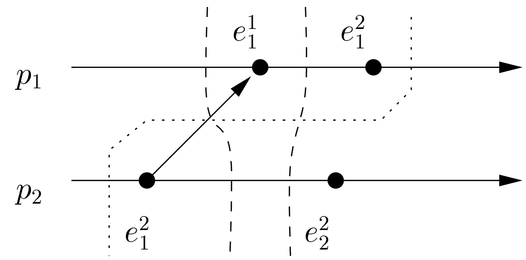

Fig. 2. Cuts representing consistent (dashed lines) and inconsistent (dotted line) states of a distributed computation.

4.2 Uncertainty in Making Observations

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 *leads to* S_{i+1}. Observing in such a way can be achieved, for example, by using vector clocks [Schwarz and Mattern 1994] to implement causal delivery of broadcasted control messages [Hadzilacos and Toueg 1994].[1] However, even when observing consistently, it may happen that two different monitors make different consistent observations when observing the same computation. For example, Figure 3 shows two monitors b_1 and b_2 consistently observing the computation of Figure 1 but in effect making different observations.

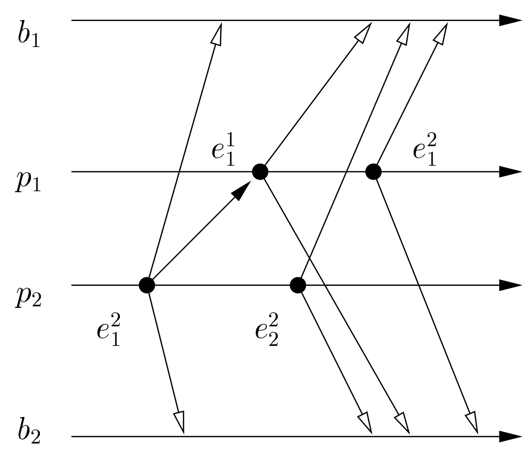

Fig. 3. Different consistent observations of the same computation.

As a consequence, two different monitors can come to different conclusions whether a global predicate holds within a computation or not.

[1]A vector clock is an array of n counters (one for every process) maintained by every process. Whenever a local event occurs at process p_i it increments the i-th entry of the vector. Sent messages are timestamped with the local clock setting. Upon receiving a message with timestamp t, the local vector clock is set to the component-wise maximum of the local clock and t. Thus, the vector clock setting is a representation of the "causal past" [Schwarz and Mattern 1994].

4.3 Predicate Transformers *possibly* and *definitely*

As shown in the previous section, answers to the question "Does predicate φ hold in a given computation?" depend on the monitor (or better: on a specific set of observations) and, hence, is a useless question in our context. It makes sense to transform the question into one which is *observer invariant*, i.e., is related to the entire set of possible observations of the computation.

It can be shown that the set of all consistent global states together with the transitive closure of the "leads to" relation forms a lattice [Babaoğlu and Marzullo 1993; Schwarz and Mattern 1994]. More precisely, the lattice is a sublattice of an n-dimensional hypercube where the top element of the lattice is the global starting state of the computation. An observation can then be viewed as a path through the lattice starting at the top element. Figure 4 shows the lattice corresponding to the computation from Figure 1.

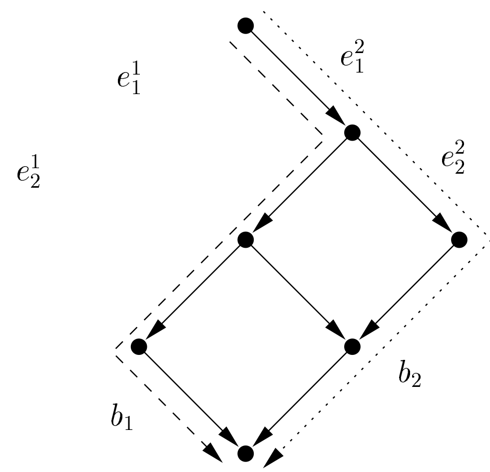

Fig. 4. The lattice of consistent global states.

Given some predicate φ as above and using this lattice, it is natural to define two notions which transform φ into a predicate over computations in the following way [Cooper and Marzullo 1991; Marzullo and Neiger 1991]:

—*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.

As mentioned in Section 3, constructs like *possibly* and *definitely* are called *predicate transformers*.

The two predicate transformers are of different use when monitoring distributed computations. For example, it is often possible to define an "exception" predicate β, i.e., a set of states which should never occur in a computation. Then for every computation it should never be the case that *possibly*(β) holds. On the other hand, *definitely*(γ) should hold for a set of states γ which are mandatory in a computation. As an example, consider a road intersection with a set of traffic lights. A predicate β in this context could state: "Both directions have a green signal." Detecting *possibly*(β) indicates a bug in the system. On the other hand, an instance of γ could state: "Traffic light x has a green signal." To be of use, a computation should satisfy *definitely*(γ) for both directions x.

The predicate transformers *possibly* and *definitely* have some affinity to the notions of *safety* and *liveness* [Lamport 1977]. Let φ be a global state predicate. If a safety property can be expressed as "always φ" then a computation should satisfy $\neg possibly(\neg\varphi)$. If a liveness property is expressible as "eventually φ", then a computation should satisfy $definitely(\varphi)$. [2]

4.4 Detection Algorithms

A number of algorithms for detecting *possibly* or *definitely* for general predicates have been published. They are based on the idea of incrementally constructing and traversing the state lattice [Cooper and Marzullo 1991; Babaoğlu and Marzullo 1993; Schwarz and Mattern 1994; Jégou et al. 1995]. The price for this generality is paid by exponential cost in terms of space [Cooper and Marzullo 1991] or time [Jégou et al. 1995]. The cost can be dramatically reduced by exploiting special properties of the predicates under consideration, restricting the generality of the detection method.

5. OBSERVATION OF ASYNCHRONOUS SYSTEMS WITH CRASH FAILURES

In the previous section we have introduced concepts related to observation in asynchronous systems that were fault-free. In this section we study the notion of observation in asynchronous systems which are opposed to faults.

5.1 The Crash Fault Assumption

A *fault assumption* defines the way in which certain components of a system might fail [Hadzilacos and Toueg 1994]. A fault assumption is a necessary starting point for any type of fault-tolerance consideration: an algorithm is not fault-tolerant "on its own", it is only fault-tolerant with respect to a certain fault assumption.

A simple and widely used fault assumption is termed *crash*. Under the *crash* fault assumption, at most $t < n$ 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. While being a rather simple fault assumption, it is arguably still a rather realistic one. Moreover, from a theoretical point of view, process crashes can have a dramatic effect on the solvability of important problems in asynchronous systems [Fischer et al. 1985]. Hence, assuming crash faults of processes is a good starting point for fault-tolerance considerations like ours.

5.2 Types and Truth Values of Predicates

In systems with crash faults, it is a natural desire to detect a class of predicates which make no sense in fault-free systems. Of course, all predicates which we wish to detect in the fault-free case are detection candidates for the faulty case, but now we 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 process in an array called *up*. 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

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

Returning to the traffic light system, we can state another example of a predicate γ as follows: "If the traffic light in one direction has crashed, then the light for the

[2] Note that $\neg possibly(\neg\varphi)$ is not equivalent to $definitely(\varphi)$ [Babaoğlu and Marzullo 1993, p. 88].

other direction must show a red signal." This can be formalized as follows:

$$\gamma \equiv \mathit{up}[\mathit{direction1}] = \mathsf{false} \Rightarrow \mathit{light}[\mathit{direction2}] = \mathit{red} \tag{2}$$

However, looking at the last example, we have to ask: If a process crashes, what happens with its local state? For example, consider a process p_1 which sets a local variable x to 1 and then crashes. Does the predicate $x = 1$ still hold after the crash or not? Interestingly, both answers make sense: For example, if the value $x = 1$ indicated the possession of a token (like in a token ring network), then we would be inclined to answer "No" since p_1 cannot give back the token. However, if $x = 1$ signaled a successful commit of a distributed database transaction, we should answer "Yes" since the opposite could lead to inconsistency issues.

Technically speaking, an answer to the question doesn't matter: For example, if you say $x = 1$ still holds after the crash, you can formulate the opposite semantics by specifying the predicate

$$(\mathit{up}[1] = \mathsf{true}) \wedge (x = 1)$$

So if p_1 crashes, the expression "$x = 1$" will not hold anymore. Hence, the types of predicates expressible is not restricted by such a decision.

5.3 Failure Detection

Defining the *up*-array as above imposes an additional burden on the monitor process, that of *failure detection*. Detection of crash faults is a task often needed in many areas of fault-tolerant distributed computing [Gärtner 1999]. To solve the problem, it is usually assumed that a software layer based on timeouts tries to keep the local *up* array at the monitor "as close to reality" as possible, meaning that, ideally, the mechanism should have $\mathit{up}[i] = \mathsf{true}$ iff p_i has actually crashed.

Failure detection looks like a simple task, but it is more difficult to implement than it seems in asynchronous environments. Intuitively, this is because it is very hard to distinguish a crashed process from one which is merely very slow. In fact, it can be shown that any form of reliable failure detection is impossible in asynchronous systems [Chandy and Misra 1986]. So failure detectors which are implementable in asynchronous systems will in general be *unreliable* [Chandra and Toueg 1996]. Unreliable means that they can make wrong suspicions or that they can fail to detect the crash of a process at all.

Following the terminology of Chandra and Toueg [1996], we will call the mechanism offering this service at a single monitor a *failure detector module*. The whole set of failure detector modules is called a *failure detector*. However, while we use the terminology of Chandra and Toueg [1996], we do not assume that the failure detectors we use have certain safety or liveness properties (which Chandra and Toueg call *accuracy* and *completeness*).

Throughout this section we assume a fixed set of m monitor processes $b_1, \ldots, b_m$, each equipped with a failure detector module. We will distinguish the individual *up* arrays by a superscript, i.e., $\mathit{up}^j[i]$ refers to the failure detector value of process p_i on monitor b_j. For the time being, we assume that the monitor processes are not affected by the crash fault assumption. We will return to this assumption in the conclusions.

5.4 Uncertainty in Observing Computations

When observing a distributed computation, the monitors regularly query their local failure detector module and construct observations in a similar way as in the fault-free case. The idea is to incorporate the information offered by the failure detector into the global state in a meaningful way. In analogy to a common way of modeling crash faults [Gärtner 1999] we have decided to treat $\mathit{up}[i]$ as being part of the state of p_i. This means a change of $\mathit{up}[i]$ will be treated as the notification of an event on process p_i.[3] Treating $up[i]$ in this way results in what we call the *extended state space*. A change in the given failure detector value $up^j[i]$ is treated as an event on p_i and thus as a state change on p_i. Using this interpretation of the failure detector value, b_j constructs its own computation lattice over the extended state space.

It turns out that the unreliability of the failure detector introduces a new type of uncertainty into observations. Consider for example the computation depicted in Figure 5 where there are two processes p_1 and p_2 being observed by two monitors. Due to inaccurate failure detection, b_1 suspects p_1 to have crashed at some point during the computation while b_2 does not suspect p_1. The moment b_1 receives the next control message from p_1 it exempts the suspicion, however, both monitors make different observations.

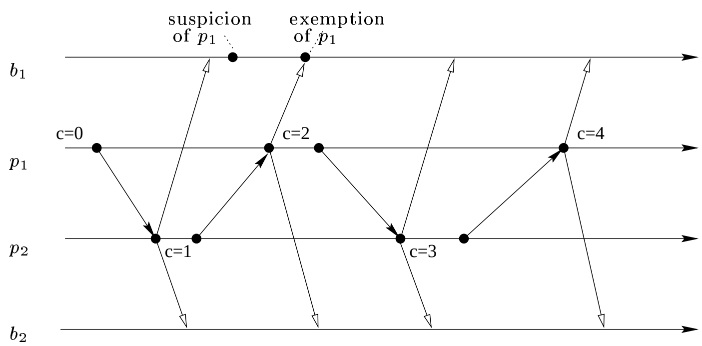

Fig. 5. Uncertainty in failure detection.

The interesting thing to note here is that constructing the computation lattice does *not* lead to an observer-invariant way of representing the computation. The lattices constructed by both b_1 and b_2 are depicted in Figure 6 (the observations they make are indicated by thick black lines). Considering for example the predicate

$$\varphi \equiv (\neg up[1]) \wedge (c = 1)$$

it is relatively easy to see that b_1 will detect $possibly(\varphi)$ while b_2 will not. Remember that in asynchronous systems there is no way to distinguish a correct from an incorrect suspicion, so both monitors make a sensible judgment about whether $possibly(\varphi)$ holds or not. This is what we mean by the uncertainty introduced by crash faults and unreliable failure detection.

5.5 Pessimistic and Optimistic Failure Detectors

Unreliable failure detectors still may be reliable sometimes. Measurements by Cristian and Fetzer [1999] have shown, that real networks alternate between short phases of asynchronous instability and long phases of near-to-synchronous stability. Thus, while it is impossible to know (in the precise sense of *knowledge* as defined for example by Halpern and Fagin [1989]) the truth about a failure suspicion, the information offered by a failure detection service can be taken as a good educated guess.

Since we want to construct observer-invariant concepts of observation in faulty asynchronous systems and the uncertainty was introduced by differing failure detector module information, the idea is to define a failure detector with a global

[3]Because we assume that a monitor process b does not interfere with the computation in any way, it is not useful to treat $up[i]$ as part of the state of b. Doing this would remove $up[i]$ from the global state. This would give rise to a lot of questions concerning the meaningfulness of predicates that refer to the functional state of processes, questions which we can avoid.

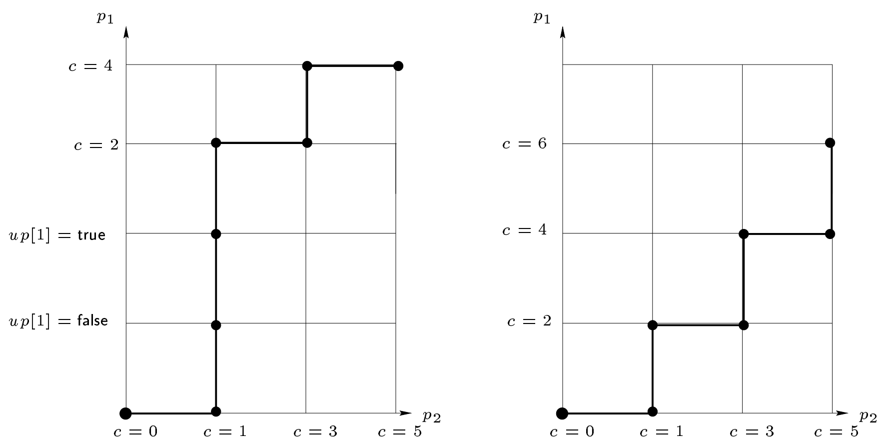

Fig. 6. The different lattices constructed by monitor b_1 (left) and b_2 (right). The top elements of the lattice appear at the bottom left corner.

semantics, i.e., a failure detector that gives the same information at every failure detector module.

Since we have to live with "imperfect" failure detectors, there are two pragmatic ways to treat the information they offer. The assumption underlying both is that the failure of a process is a "bad" thing and detecting a failure usually causes special routines like exception handlers to be invoked. Under this assumption, we state the two basic ways in which the information of a failure detector can be interpreted as follows:

(1) (*pessimistic*) The assumption underlying this approach is that it is better to assume the crash of a process as soon as possible, i.e., as soon as some failure detector module suspects the crash.
(2) (*optimistic*) Here it is better to assume the crash as late as possible, i.e., not until all failure detector modules unanimously suspect a process.

The pessimistic approach is always adequate if an immediate response to a fault occurrence is needed. For example, if considerable value is at stake if a fault is not handled early enough (like in manned spaceflight) the pessimistic approach should be used. On the other hand, if the cost of fault handling is rather high, the optimistic approach is adequate. For example, expensive reconfiguration work (like digging up and repairing a fibre optic cable) should only be done if it is absolutely sure that a failure has indeed happened.[4]

Both approaches can be formalized using the information offered by failure detector modules: A pessimistic observer would assume a "real" crash of process p_i as soon as a single failure detector module suspects p_i. An optimistic observer would wait for all failure detector modules to suspect p_i at the same time before actually assuming that it has really crashed. Combining the information of the failure detector modules, we can define two new types of failure detectors. Definition 1 constitutes a "pessimistic" failure detector because it suspects a process p_i to have crashed as soon as there is one monitor which thinks p_i has crashed.

Definition 1. A *pessimistic failure detector* is an array up_{pes} of n boolean values. The value of the array is defined as follows: For every point in time of the execution, $up_{pes}[i] = \mathsf{false}$ iff there exists a j for which $up^j[i] = \mathsf{false}$.

[4]Note that both approaches do not give real assurance in the fact that a fault has happened. This fact is merely very probable. Since failure detectors must be unreliable in asynchronous systems, these are circumstances we have to live with.

In the same way we can define an optimistic failure detector which waits until all individual failure detectors agree that p_i has in fact crashed. In this sense, optimistic includes a notion of conservativeness.

Definition 2. An *optimistic failure detector* is an array up_{opt} of n boolean values. The value of the array is defined as follows: For every point in time of the execution, $up_{opt}[i] = \mathsf{false}$ iff for all j holds $up^j[i] = \mathsf{false}$.

Definitions 1 and 2 define failure detectors which have a "global" semantics, i.e., the same behavior at every individual monitor. This indicates that these failure detectors are at the core of defining predicate transformers suitable for faulty asynchronous systems.

5.6 Predicate Transformers *negotiably* and *discernibly*

5.6.1 *Defining negotiably.* Now assume that with the given detection predicate φ at hand, monitor b_j observes a distributed computation and runs a predicate detection algorithm for *possibly*(φ) on the extended state space lattice. Now observe that the results of the detection attempts of two different monitors only vary if the failure detectors at the individual monitors give inconsistent evidence at some point during the execution. At this point we can use the failure detectors defined in the previous section to define the following new predicate transformer *negotiably* as follows:

Definition 3. Given a general predicate φ on the extended state space, the predicate *negotiably*(φ) holds iff a monitor can detect *possibly*(φ) using a pessimistic global failure detector.

In common English, "negotiable" means that something can be discussed and changed before being agreed upon. This already indicates that the predicate transformer *negotiably* takes a pessimistic view of failure detector information. Intuitively, *negotiably*(φ) holds iff φ possibly holds during the computation and a crash is assumed as soon as possible.

Interestingly, *negotiably* inherits the affinity of *possibly* towards safety properties which was mentioned in Section 4.3. Safety properties often identify a set of bad states which the system is supposed not to enter. If such a safety property is expressible in the form "always φ", then the computation should satisfy $\neg$*negotiably*($\neg\varphi$). This means that something bad has (probably) happened if *negotiably*($\neg\varphi$) can be detected. For example, the crash fault assumption postulates that at most $t < n$ processes are allowed to crash. Since this is only an assumption, more than t processes might crash in a real system. If we define a predicate γ as

$$\gamma \equiv \text{“for at most } t \text{ processes } p_i \text{ holds } up[i] = \mathsf{false}\text{”} \tag{3}$$

we can use a detector for *negotiably*($\neg\gamma$) to validate the fault assumption. A different example is the "traffic light" predicate γ defined in Equation 2 in Section 5.2. If $\neg$*negotiably*($\neg\gamma$) holds, there might be a bug in the algorithm.

5.6.2 *Defining discernibly.* It is relatively easy to see that in analogy to *negotiably* it is possible to define an "optimistic" counterpart. Using the concept of Definition 2 we can now define a new predicate transformer *discernibly* in the obvious way.

Definition 4. Given a general predicate φ on the extended state space, the predicate *discernibly*(φ) holds iff a monitor can detect *definitely*(φ) using an optimistic global failure detector.

Usually, "discernible" means that something is understood only after looking and thinking about it very carefully. So intuitively, *discernibly*(φ) holds only if φ definitely holds during the computation and a crash is assumed only if there is no way of denying it.

Again, *discernibly* has a similar affinity to liveness properties as *definitely*. Sometimes a liveness property can be expressed in the form "eventually φ", where φ is a global state predicate. In these cases, if a computation satisfies *discernibly*(φ) then (probably) the computation satisfies the liveness property in the presence of faults. Thus, *discernibly* can be used to check progress of a computation.

5.7 Building Optimistic and Pessimistic Failure Detectors

Obviously, we are interested in devising detection algorithms for *negotiably* and *discernibly*. The central problem to solve is building optimistic and pessimistic failure detectors. We now sketch how such a failure detector can be implemented. Because of lack of space, instead of a formal proof we give a detailed argument of why our proposed mechanism correctly implements an optimistic or pessimistic failure detector.

5.7.1 *Application timestamp of failure detection messages.* The basic idea is to diffuse the detection events among the monitors in the following way: Whenever a monitor b_j suspects process p_i it sends a message "$up^j[i] := \mathsf{false}$" to all monitors. This message is timestamped with the current value of b_j's local vector clock. This timestamp relates the failure detection event with the latest consistent global state that b_j has observed. This timestamp can be thought of as being b_j's approximation of the causal timestamp of a crash event on p_i. Hence, we call this the *application timestamp* of the detection event. Note that timestamping the detection event this way, the i-th entry of the timestamp indicates the event number k of the last event on p_i which b_j has reached knowledge of via control messages. Hence, when a monitor b_l receives such a message, it incorporates the event "$up^j[i] := \mathsf{false}$" into its own state lattice by augmenting the timeline of process p_i between event e_k and e_{k+1}. Commencing in this way, it is clear that eventually every monitor will incorporate the suspicion into its own state lattice at the same place. (The actions are similar if a failure detector refutes its suspicion, i.e., sets $up^j[i]$ to true again.)

The problem is that these additional control messages also take time to travel between the monitors. This means that it is impossible to inform all monitors in a timely "instantaneous" fashion about a failure suspicion. A monitor b_l might already have received events e_{k+1}, e_{k+2}, ... from p_i before it is notified about the suspicion of p_i by b_j. This would at least temporarily lead to differing state lattices on different monitors—a thing we must avoid.

5.7.2 *Causal delivery of failure detection messages.* The idea now is to introduce a software layer into the detection algorithm which holds back regular control message notifications until it is sure that no prior failure detector notifications will arrive. The problem here is similar to the problem of implementing causal delivery of control messages which was discussed in Section 4.2. Using vector clock timestamps on the failure suspicion notifications it is possible to determine the "latest" failure suspicion on every monitor. To see this, consider the following example.

Figure 7 shows a space-time diagram of a monitored distributed computation where only the timelines of the monitors and the local failure suspicions are shown. Every monitor maintains a vector clock of size m to timestamp failure detection control messages. Whenever the local failure detector module at monitor b_j suspects a process, it increments the j-th entry in the vector and sends the resulting value as a timestamp together with the notification itself to all other monitors (this is done by b_1 for example). The timestamp is called the *monitor timestamp* of a failure detection notification message.

Whenever a monitor receives such a message, it constructs the component-wise maximum of the incoming timestamp and its local vector clock. This value indicates the current "knowledge" of the causal past of failure suspicions. By keeping track of how many failure detection notifications have been received from which monitor,

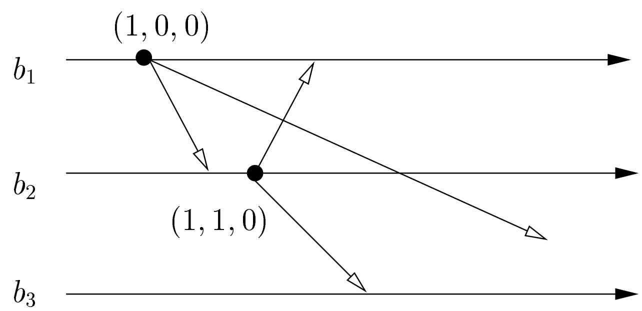

Fig. 7. Causal delivery of failure detection messages.

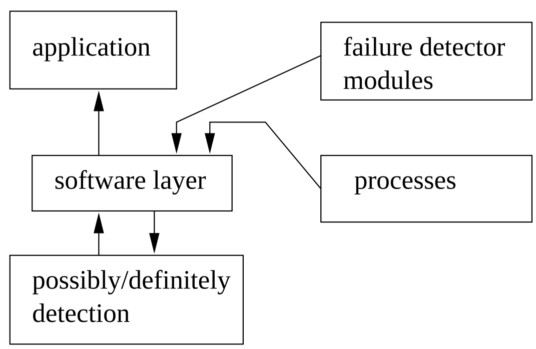

Fig. 8. Architecture of the filter to build an optimistic or pessimistic failure detector.

it is possible to derive whether or not there are still some notification messages in transit. For example, when b_2 receives the suspicion message from b_2 with vector timestamp $(1, 1, 0)$, it can conclude that there is still one message from b_1 in transit. When this message arrives, it knows that no other suspicion messages will arrive which causally precede the two detection events on monitors b_1 and b_2.

5.7.3 *Determining the common state lattice.* Using the *application timestamp* of these latest failure detector notifications, it is possible to determine those application events which will not be preceded by any failure detection events on every process.

For example, assume the monitors in Figure 7 are observing a computation between four processes. Let the application timestamp of b_1's detection message be $(10, 7, 4, 8)$ and the application timestamp of b_2's message be $(8, 7, 14, 9)$. After receiving both of these messages, b_3 takes the component-wise minimum of these timestamps and of its own vector clock: This gives the most recent consistent global state S_i which all monitors have seen and up to which all failure detection messages have been received. Thus, the sublattice containing all consistent global states that precede S_i in the computation lattice is common to all monitors. This sublattice can be used to extract optimistic and pessimistic failure detection information.

5.7.4 *Architecture of the software layer.* We propose the architecture for the software layer implementing optimistic and pessimistic failure detectors which is depicted in Figure 8. The filter is the same across all monitors observing the computation. We assume the following:

—The local failure detector module has access to information about the most recent consistent global state in order to equip detection messages with the application timestamp and diffuse them to other monitors.

variables:
most_recent_ats$[1..m, 1..n]$ **of counter init. 0**
events **set of events initially** $\emptyset$

algorithm:
$\langle$ upon receipt of (e, ts) from p_i $\rangle$ **do**
$events := events \cup (e, ts)$

$\langle$ upon receipt of (f, ats) from b_j $\rangle$ **do**
$events := events \cup (f, ats)$
$most\_recent\_ats[j] := ats$
$common := \min_{j=1..m}\{most\_recent\_ats[j]\}$
$\langle$release all elements from *events* prior
to *common*$\rangle$

Fig. 9. Data structures and algorithm of the software layer.

—The detection messages are diffused using a monitor timestamp so that they are delivered in a causally consistent way to the software layer (again, the techniques described by Hadzilacos and Toueg [1994] can be used).

—Control messages issued by the processes are already handed to the filter in causal order.

The pseudocode of the software layer is given in Figure 9. We maintain two data structures within the layer:

—*most_recent_ats* is an $m \times n$ array of counters, i.e., it holds the application timestamps of the latest failure detection messages from every monitor.

—*events* is a set of events used to buffer the events and release them in a controlled manner.

Whenever the software layer receives a regular control message about a state change on process p_i, it simply adds this event to the event set. Whenever it receives a failure detection notification from b_j it converts this event into a regular event by adding it to *events* and updates the array of most recent application timestamps. Using this information in can now compute the common consistent global state of all monitors and release all events from the event set which are prior to that state.[5] These events are the input to an algorithm to detect *possibly* or *definitely*, as we now explain.

5.7.5 *Optimistic and pessimistic information extraction.* Because of the way how failure notifications are timestamped, a sequence of suspicions and exemptions along the timeline of process p_i will always collapse into at most two events: a single suspicion followed by a single exemption (before the next regular event). In any case, we assume that a failure suspicion of a p_i at b_j is exempted whenever a control message from p_i arrives at b_j. It is now possible to attach information to the suspicion event by explicitly keeping track of which monitor suspected p_i during the suspicion-exemption interval. It is now relatively easy to extract the optimistic and pessimistic failure detection information from this: In a pessimistic failure detector, a suspicion event is released whenever at least one monitor suspects p_i. An optimistic failure detector will only need to release a failure detection event from the software layer if in fact all monitors have sent the same suspicion message for the same event interval.

[5]Obviously, the computation lattice 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.

5.8 Detection Algorithms

The detection algorithms for *negotiably* and *discernibly* both use the software layer explained above. For the former, the layer releases all regular events together with failure suspicions which resemble a pessimistic failure detector. For the latter, the layer emulates an optimistic failure detector. Receiving input in this way, it is possible to construct the computation lattice of the extended state space. For detecting *negotiably*(φ) one of the published algorithms for detecting *possibly*(φ) can be used. For detecting *discernibly*(φ) we can exploit an algorithm for detecting *definitely*(φ).

6. SUMMARY AND CONCLUSIONS

Distributed predicate detection is an important practical problem in many areas of program testing and analysis. We have concentrated on asynchronous distributed systems and in contrast to almost all of the prior work have studied the problem in a general form under the crash fault assumption. We have shown that crash faults introduce a new form of uncertainty into observations of a distributed computation, rendering the previously established methodologies useless. We have proposed a novel set of predicate transformers (called *negotiably* and *discernibly*) with semantics that explicitly take this new form of uncertainty into account and which are thus especially tailored to faulty asynchronous systems. The transformers can be understood easily in terms of their "fault-free counterparts" *possibly* and *definitely*. In fact, if no faults occur (and none are detected) and if the detection predicate does not reference the functional state of processes, then the semantics of *negotiably* and *discernibly* reverts back to those of *possibly* and *definitely*. We have also indicated how algorithms can be constructed in a modular way for the detection of these new predicates.

In their well-readable book chapter on consistent global states, Babaoğlu and Marzullo [1993] state that in cases where processes may fail, it may be "necessary to extend the causal delivery abstraction to include not only actual messages but also failure notifications" [Babaoğlu and Marzullo 1993, p. 92]. We think that our investigations are a good first starting point for further studies in this direction.

Since our study is only a starting point, many questions still remain. For example, the semantics of our predicate transformers are based on the assumption that monitor processes never fail. This may not be a realistic assumption, since often monitor and application processes reside on the same physical nodes and do not fail independently. In this case, monitors must be replicated, which makes it necessary to replicate the failure detector module they use too. What is the impact of such an undertaking on the structure and complexity of our detection algorithms?

Since the semantics of our predicate transformers are based on the information offered by the failure detectors, we have noted that it is even harder to associate any real evidence with a successful detection of a predicate than it was the case in ordinary distributed predicate detection. Therefore, an interesting field of investigation would also be to relate the correctness of our detection algorithms to different correctness conditions for failure detectors (as the ones described by Chandra and Toueg [1996]).

It is obvious that the predicate detection scheme we propose is very costly both in terms of local computation time and messages. This is not surprising since the detection of general predicates in fault-free systems has been shown to be NP-complete [Chase and Garg 1998]. In this sense, our scheme may be regarded as a theoretical study giving way to more efficient detection semantics and algorithms. Work by Garg and Mitchell [1998] has shown that the results obtained from classifying predicates in the fault-free case [Chase and Garg 1998] can sometimes be transferred to the faulty case rather elegantly, a result encouraging more work in this direction.

Acknowledgements

The second author greatly acknowledges support by the Deutsche Forschungsgemeinschaft (DFG) as part of the PhD program (Graduiertenkolleg) "Intelligente Systeme für die Informations- und Automatisierungstechnik" at Darmstadt University of Technology.

REFERENCES

BABAOĞLU, Ö. AND MARZULLO, K. 1993. Consistent global states of distributed systems: Fundamental concepts and mechanisms. In S. MULLENDER Ed., *Distributed Systems* (Second ed.)., Chapter 4, pp. 55–96. Addison-Wesley.

CHANDRA, T. D. AND TOUEG, S. 1996. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM 43*, 2 (March), 225–267.

CHANDY, K. M. AND MISRA, J. 1986. How processes learn. *Distributed Computing 1*, 40–52.

CHASE, C. M. AND GARG, V. K. 1998. Detection of global predicates: Techniques and their limitations. *Distributed Computing 11*, 4, 191–201.

COOPER, R. AND MARZULLO, K. 1991. Consistent detection of global predicates. *ACM SIGPLAN Notices 26*, 12 (Dec.), 167–174.

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

FISCHER, M. J., LYNCH, N. A., AND PATERSON, M. S. 1985. Impossibility of distributed consensus with one faulty process. *Journal of the ACM 32*, 2 (April), 374–382.

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

GÄRTNER, F. C. 1999. Fundamentals of fault-tolerant distributed computing in asynchronous environments. *ACM Computing Surveys 31*, 1 (March), 1–26.

HADZILACOS, V. AND TOUEG, S. 1994. A modular approach to fault-tolerant broadcasts and related problems. Technical Report TR94-1425 (May), Cornell University, Computer Science Department.

HALPERN, J. Y. AND FAGIN, R. 1989. Modellung knowledge and action in distributed systems. *Distributed Computing 3*, 159–177.

JÉGOU, R., MEDINA, R., AND NOURINE, L. 1995. Linear space algorithm for on-line detection of global predicates. In J. DESEL Ed., *Proceedings of the International Workshop on Structures in Concurrency Theory (STRICT)*, Workshops in Computing (1995), pp. 175–189. Springer-Verlag.

LAMPORT, L. 1977. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering 3*, 2 (March), 125–143.

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

MARZULLO, K. AND NEIGER, G. 1991. Detection of global state predicates. In *Proceedings of the 5th International Workshop on Distributed Algorithms (WDAG91)* (1991), pp. 254–272.

SCHWARZ, R. AND MATTERN, F. 1994. Detecting causal relationships in distributed computations: in search of the holy grail. *Distributed Computing 7*, 149–174.