Distributed predicate detection in a faulty environment

Vijay K. Garg* J. Roger Mitchell†
http://maple.ece.utexas.edu
Department of Electrical and Computer Engineering
The University of Texas at Austin,
Austin, TX 78712-1084

Abstract

There has been very little research in distributed predicate detection for faulty, asynchronous environments. In this paper we define a class of predicates called set decreasing *predicates which can be detected in such an environment. We introduce a set of failure detectors called infinitely often accurate detectors which are implementable in asynchronous systems. Based on these failure detectors we present an algorithm to detect conjunction of local predicates and send-monotonic channel predicates. Since perfect failure detection is impossible in an asynchronous system, we cannot guarantee that our detection algorithm will not have false detections. However, if the predicate ever holds then it is guaranteed to be detected.*

1 Introduction

Detecting a distributed predicate is a fundamental problem in distributed systems. However, very little research has been done for detecting these predicates in a faulty environment. Providing predicate detection amid failures can be useful for monitoring predicates that correspond to "bad states." Some examples of such predicates are: "no leader," "no token," or "no process or channel active." Consider an application which requires that a token be passed among a set of processes. The failure of a process which holds the token, or failure of a message containing the token means the token is no longer in the system. It is useful to detect this condition so that a new token can be generated.

A *perfect* predicate detection algorithm is expected to satisfy two properties - *completeness* and *accuracy*. A predicate detection algorithm is complete if it detects any occurrence of the predicate. A predicate detection algorithm is accurate if it it avoids false detection. Since failure detection is a special case of general predicate detection, and perfect failure detection is not possible in an asynchronous system [2, 7], it follows that perfect predicate detection is also impossible in a faulty, asynchronous system. In this paper we present a method for monitoring predicates in a faulty system in which we accept the limitations of failure detection. Our algorithm uses a class of failure detectors called infinitely often accurate (IO) detectors [11, 16]. These failure detectors guarantee that a failed process is always suspected; however, they may sometimes suspect an unfailed process. It is guaranteed that a false suspicion will not hold permanently at any process. The chief advantage of these failure detectors is that they can be implemented in an asynchronous environment.

Using IO failure detectors we describe an algorithm for detecting those global predicates which satisfy two properties. We require predicates to be *set decreasing* and *conjunctive*. A global predicate is *set decreasing* if whenever it holds for a set of processes and channels, it also holds for a subset of these processes and channels (with possibly fewer messages). For example, "no token" in the system is a set decreasing predicate. This property allows the algorithm to remove from consideration the suspected processes, channels or messages for the purpose of predicate evaluation. If the global predicate is false for the smaller set, then it must be false for the entire system even if the suspicions were incorrect. A set decreasing global predicate is *conjunctive* if it can be written as a conjunction of local predicates and *send-monotonic* channel predicates [8]. This property allows efficient evaluation of even unstable predicates.

We show that our predicate detection algorithm satisfies completeness, i.e. if the predicate holds it is detected by the algorithm. Since our failure detection mechanism is not perfect, it is clear that our predicate detection algorithm may sometimes give false detection. These false predicate detections arise from inaccurate failure detection. In fact if all failure detectors

*supported in part by the NSF Grants ECS-9414780, CCR-9520540, Texas Education Board Grant ARP-320, a General Motors Fellowship, and an IBM grant (garg@ece.utexas.edu)

†supported in part by a Virginia & Ernest Cockrell fellowship

are accurate for some computation, then our predicate detector is also accurate for that computation.

There is a large body of research for predicate detection in a non-faulty environment. Chandy and Lamport [3] present a method for detecting stable global predicates (predicates such as termination which, once true, remain true) using snapshots. Cooper and Marzullo [4] detect any stable or unstable predicate but require a search of the entire exponential state space. Garg and Waldecker [10] present an efficient on-line method of detection for conjunctive process predicates using control messages and a central checker process. In [8] and [9] Garg, et. al. present centralized and non-centralized on-line methods for detecting unstable conjunctions of process and channel predicates using control messages. In [12], Hurfin, et. al. give a distributed on-line method for detecting process predicates using only application messages.

For predicate detection in a faulty distributed environment, previous work has considered only specific global conditions. Li and McMillin, [14] give a method for deadlock detection and Venkatesan [18] gives a method for termination detection. In Meta [1], combinations of process conditions can be detected, but this approach requires underlying causal message ordering and broadcasts of control messages for detection. All of these algorithms assume perfect failure detection. Our approach is not based on perfect failure detection and can be implemented in an asynchronous system.

2 Model

The environment we consider is a system of N processes which communicate solely by passing messages via channels. The processes do not share a global clock or memory. We use P_i,P_j,P_k as identifiers for processes. The run of a process, P_i, is a sequence of states. We use s, t, and u to denote states. The relation $s \prec t$ means that s and t are states at the same process and s occurred before t. The relation $s \preceq t$ means that $s \prec t$ or s is equal to t. The relation, $\rightarrow$, is used to order states in the same manner as Lamport's happened before relation on events [13]. Therefore, $s \rightarrow t$ is the smallest transitive relation satisfying 1) $s \prec t$ implies $s \rightarrow t$, and 2) if s is the state immediately preceding the send of a message and t is the state immediately following the receive of that message then $s \rightarrow t$.

A global state, G, is defined as a set containing exactly one state from every process. The state of P_i in G is represented by $G[i]$. We also use the $\prec$ relation on global states. $H \prec G \iff \forall i : H[i] \preceq G[i] \wedge \exists j : H[j] \prec G[j]$. A consistent global state is a global state such that for all $s,t \in G : s \not\rightarrow t \wedge t \not\rightarrow s$. That is, all s,t in G are *concurrent*. For the remainder of the paper, we will only consider consistent global states.

In our model we allow processes to crash, channels to fail and messages to be lost. A process crashes by ceasing all its activity. We assume that once a process has failed (or crashed) it stays failed throughout the run. The predicate $failed(i)$ holds if the process i has failed in the given run. A process that has not failed is called a *correct* process. We assume that at least one of the processes in the system is correct. Let C denote the set of states on correct processes and C_j denote the set of states on any correct process j.

We define a channel failure as the condition when a channel stops delivering messages or when a process adjacent to the channel crashes. As for processes, once a channel fails it is assumed to stay failed. In a distributed system, a process failure is indistinguishable from failures of all channels adjacent to it. However, it is possible for a process to distinguish a channel failure from a process failure when a message is received from the suspected process along a different channel.

A message loss corresponds to a message that is sent by the sender but never received by the receiver. Note that a channel failure is a stronger condition than a message failure.

Finally, we assume that the run of a computation is infinite and every state of every process is eventually in the causal past of every process except where process crashes or channel crashes preclude this. This assumption is used for guaranteeing detection of predicates. If the computation is finite or some of the processes terminate, then it is sufficient to require that every process sends message to all processes before termination. This will ensure that if the predicate occurs in the computation then it will be detected even when the computation is finite.

2.1 Failure detection

A failure detector is responsible for maintaining the value of a predicate *suspects* at all correct processes. The predicate $suspects(s,i)$ holds if the process i is suspected in the state s (by the process which contains s). Note that this process may stop suspecting process i in some later state t.

We would like our failure detectors to satisfy certain completeness and accuracy properties. The completeness properties require suspicion of failed processes. The *weak completeness* property requires that every failed process is eventually permanently suspected by some correct process. Let the predicate $permsusp(s,i)$ be defined as

$$permsusp(s,i) \equiv \forall t : s \preceq t : suspects(t,i)$$

A detector is defined to be weak complete if for any run,

$$\forall i : \langle failed(i) \Rightarrow \exists s \in C : permsusp(s, i) \rangle$$

The strong completeness property requires the failed process to be eventually suspected by *all* correct processes. Formally,

$$\forall i, j : failed(i) \wedge \neg failed(j) \Rightarrow \exists s \in C_j : permsusp(s, i)$$

In [2], four accuracy properties have been presented. The weakest of these properties is *eventual weak accuracy*. A detector satisfies eventual weak accuracy if eventually some correct process is never suspected by any correct process, i.e., there exists a correct process i such that for all correct processes j

$$\exists s \in C_j, \forall t : s \preceq t : \neg suspects(t, i)$$

However, as shown in [2] a failure detector which satisfies weak completeness and eventual weak accuracy, called eventually weak detector ($\diamond\ \mathcal{W}$), can be used to solve the consensus problem in an asynchronous system[7]. This implies that all of the failure detectors in [2] are impossible to implement in asynchronous systems.

We now introduce a weaker accuracy property which we call infinitely often accuracy. A detector is infinitely often accurate if no correct process permanently suspects an unfailed process. Formally,

Definition 1 *A detector is* infinitely often accurate *if*

$$\forall i : \langle \neg failed(i) \Rightarrow \forall s \in C : \neg permsusp(s, i) \rangle$$

An IO detector is a failure detector which satisfies weak completeness and infinitely often accuracy. By combining the two properties, we get the following pleasant property of an IO detector.

$$\forall i : \langle failed(i) \equiv \exists s \in C : permsusp(s, i) \rangle$$

Intuitively, this says that a failure of a process is equivalent to permanent suspicion by some correct process.

A possible implementation of an IO detector is shown in Fig. 1. This implementation is similar to that proposed by [5]. The algorithm maintains a timeout period for each process. The variable $watch[i]$ is the timer for the process P_i. When the timer expires, the process is suspected. On the other hand when a message is received while a process is under a suspicion, the suspicion is removed and the timeout period for that process is increased. It is easy to verify that this

```
var
  suspects : set of processes initially {};
  timeout: array[1..N] of integer initially t;
  watch: array[1..N] 0f timer initially set to timeout;

After every t units;
send "alive" to all processes;

On receiving "alive" from P_i;
  if i ∈ suspects then
      suspects := suspects − {i};
      timeout[i]++;
  Set watch[i] timer for timeout;

On expiry of watch[i]
  suspects := suspects ∪ {i};
```

Figure 1: Implementation of an IO detector at P_j

detector satisfies strong completeness and infinitely often accuracy. From now on we will assume both of these properties for our failure detectors.

Observe that one needs to be careful with designing algorithms for failure detectors. For example, the approach of sending a query message and waiting for a reply for a certain timeout period does not satisfy the IO-property. The reply from a correct process to a query may always arrive after the timeout period causing a process to permanently suspect a correct process.

We have discussed suspicions for process failures. Simple mechanisms that satisfy IO-property can also be built for channel failures and message loss. An IO detector for channel failure can be implemented by the receiver of the channel as follows. Each process sends "alive" messages to all other processes infinitely often via different paths. If the receiver does not receive any message on a channel for some timeout interval then it starts suspecting the channel. Similarly, a message m is suspected if a message sent later than m is received but m is not. It is easy to verify that for channel failures as well as message loss we have completeness and IO-accuracy. In fact, all of them satisfy strong completeness and IO-accuracy.

2.2 Detectable predicates

We now define a class of predicates for which we can guarantee efficient detection. In a faulty environment we must exclude from consideration the entity (process, channel or message) which failed. We call a predicate *set decreasing* if it continues to hold upon removal of a process, a channel, or a message. The predicate "no token in system" is set decreasing.

Formally, let S denote the set of entities that should

be excluded from consideration. We say that $B(G)$ *holds with* S to denote that B holds in the global state G from which processes, channels and messages in S are removed from consideration. Now the property of set decreasing predicates can be defined as below.

Definition 2 *A predicate, B, is a set decreasing predicate if for any global state G and lists of entities to be excluded S and S':*
$(B(G)$ holds with $S) \wedge (S \subseteq S') \Rightarrow B(G)$ holds with S'

The set decreasing property gives us the following desirable assertion on the detection algorithm. If the global predicate B is false for the system from which entities that are suspected are removed, then it must be false for the entire system. This assertion allows the detection algorithm to be complete even when suspicions are inaccurate.

An example of a predicate which is not set decreasing is - "there are more than k tokens in the system". An inaccurate suspicion of a process may lead the predicate to evaluate to false on a subsytem even though it may be true on the entire system.

For efficiency, we also restrict the global predicate G to be a conjunction of *local* predicates and *send-monotonic* channel predicates. A *local* predicate is any boolean predicate which can be evaluated on the state of a single process. We use $LP_i(G)$ to mean that the *local* predicate is true on state $G[i]$. A channel predicate CP_{ij} on a global state G is a boolean valued function on the state of the channel from P_i to P_j in G where the state of a channel is defined as the set of messages sent before $G[i]$ but not received before $G[j]$. A channel predicate is *send-monotonic* if it satisfies the condition that if it is false for any global state, then it will remain false on sending more messages without any receives. The predicate "the channel is empty" is send-monotonic. If it is false, sending more messages will not make it true. Conjunctive predicates reduce the global state space to search from $O(M^N)$ to $O(MN)$ states, where M is the maximum number of messages sent by any process.

Thus, predicates we detect are of the form

$$B(G) \equiv \bigwedge_i LP_i(G) \wedge \bigwedge_{i,j} CP_{ij}(G) \qquad \text{(E1)}$$

For example, the predicate "no token in system," can be expressed within this equation by setting LP_i to "token not held" for all i, and CP_{ij} to "token not in channel" for all i,j. For the distributed predicate "no leader", $LP_i()$ would be "process not leader" and $CP_{ij}()$ would be TRUE.

Since our environment is faulty the algorithm excludes components that are suspected. Let $S(G)$ denote the entities suspected in the global state G. We use functions named $\mathsf{SP}(G,i)$, $\mathsf{SC}(G,(i,j))$, and $\mathsf{SM}(G,m)$ that return true if the process i, channel (i,j) or the message m is suspected in the global state G. Therefore, we modify E1 to:

$$B(G) \equiv \bigwedge_i [LP_i(G) \vee SP(G,i)] \wedge$$
$$\bigwedge_{i,j} [CP_{ij}(G - \{m \mid SM(G,m)\}) \vee SC(G,(i,j))] \quad \text{(E2)}$$

where $CP_{ij}(G-\{m \mid SM(G,m)\})$ corresponds to evaluation of the channel predicate on the state of the channel from which the suspected messages are removed.

It is easy to see that the predicate E1 is a set decreasing predicate. If a process (and all its adjacent channels), or a channel is excluded from evaluation of B due to suspicions, then the truthness of B continues to hold. Similarly, if some messages are removed due to suspicions, then the truthness of B still holds due to send-monotonicity of channel predicates. Thus, our algorithm uses E2 for evaluation of $B(G)$ instead of E1. We have the following Theorem.

Theorem 3 *Assume that B is set decreasing and that IO failure detectors are used for failure detection.*
1. E1 $\Rightarrow$ E2.
2. If all IO failure detectors are accurate on G, then E2 $\Rightarrow$ E1.

Proof: 1. Let B hold on G with failures in G. Since IO detectors are strong complete, the set of all the failed entities is a subset of $S(G)$. Since B is a set decreasing predicate it follows that B also holds on G with suspicions $S(G)$.

2. If all failure detectors are accurate on G, then $S(G)$ is a subset of all the failed entities in G. Therefore, if B holds on G with $S(G)$, then it also holds on G with failures. ■

3 Algorithm for detecting predicates

Our predicate detection algorithm is completely distributed so that it works in spite of failures. It is based on the algorithm for weak conjunctive predicate detection in [12]. In our algorithm every process performs predicate detection and no reliance is made on control messages for detection. It includes predicate information in application messages so that at any time a process can determine if a predicate holds in its causal past.

Failure detectors reside at every process and the suspicions they generate are input to the algorithm via suspicion sets. It is important to note that the algorithm cannot simply exclude processes, channels or

messages that are *currently* suspected. Processes need to maintain the information regarding when the suspicion for any entity was true. In our algorithm, we will simply assume that the failure detector provides values of the functions SP(G, i), SC$(G, (i, j))$ and SM(G, m) at all processes.

3.1 Data Structures

In our algorithm, three important data structures - VC, F, ML - are maintained at every process and sent in every application message. The first is the vector clock VC which is a slight variant of the one proposed in literature [12, 6, 15]. VC is an array with N components, one per process, each of which is a structure of two fields - val and LP. The definition of VC is based on the notion of a state interval. A *state interval* is a sequence of states between two *external* events (send of a message, receive of a message, beginning of the process, or termination of a process). Now VC for the process P_k can be defined as follows. The field VC[k].val indicates the current state interval number of P_k where the state interval number is equal to the number of messages sent or received by P_k. The entry VC[i].val at P_k for i different from k denotes the number of state intervals at P_i which causally precede the current interval at P_k. Thus, initially the value of VC[i].val is 0 for all i. For process P_k, the component VC[k].val is incremented for every send and receive. Further, the vector clock is included in all outgoing messages. On receiving a vector clock in a message, the process takes a component-wise maximum of its own vector clock with that received. It can be easily shown that the vector clock (as defined here) always corresponds to a consistent cut.

The second field LP indicates the value of the local predicate in the state indicated by the vector clock. This field is also updated when a message is received. The operation of taking maximum of two vector clocks X and Y will return a vector clock Z as follows. For any i,

$$\begin{aligned} &\text{Z}[i].\text{val} = \max(\text{X[i].val}, \text{Y[i].val}), \\ &\text{Z}[i].\text{LP} \\ &= \begin{cases} \text{X}[i].\text{LP} & \text{if X}[i].\text{val} > \text{Y}[i].\text{val} \\ \text{Y}[i].\text{LP} & \text{if X}[i].\text{val} < \text{Y}[i].\text{val} \\ \text{X}[i].\text{LP} \vee \text{Y}[i].\text{LP} & \text{if X}[i].\text{val} = \text{Y}[i].\text{val} \end{cases} \end{aligned}$$

The data structure first cut, F, represents the earliest global state for which the predicate detected could be true. It has the identical structure as VC. The variable F has tags indicating the current state (F[i].val), and the state of LP_i (F[i].LP) on the global state represented by F. The algorithm maintains the invariant that F always represents a consistent global state and that no consistent global state earlier than F has the conjunctive predicate true. This operation is similar to the algorithm of [12]. To update F a process also maintains a FIFO queue of states in its local past after F in which the local predicate is true. This queue, called SL (State List) is similar to that maintained by the monitor process in the distributed algorithm for weak conjunctive predicates [9].

The first cut F is initially set to VC and is passed with every message. F uses the same algorithm as vector clocks for combining the received entries of F with that of the local process (Note that F[k].val is not incremented after the send or receive event). After performing this combination when receiving a message, the local F is also combined with the earliest VC in SL (which is not earlier than F). If SL is empty, F is set to the current VC. By performing these actions, F always represents the earliest global state for which the predicate being detected could be true.

Finally, ML is the set of messages sent in the past which have not been received before the global state identified by F. This set is used to compute the state of the channel in the consistent cut F. Based on this state the channel predicate can be evaluated. Each message m is identified with four fields - sent, rcvd, source, and dest - corresponding to the state a message was sent in, received in, and the source and destination processes.

3.2 Actions of the Algorithm

The algorithm can be understood by considering what a process does when the local predicate becomes true, when it sends a message and when it receives a message. We will discuss the events at a process P_k. Our algorithm is symmetric; all processes execute the same program.

When the local predicate becomes true for the first time for the state VC[k], the process adds the vector clock for that state to SL. Further, if F[k].val is equal to VC[k].val then F[k].LP is set to true. Since the global predicate could now be true, we check for it by *eval_global* (described in Section 3.3). Formally,

```
Upon LP_k true first time for VC[k]
VC[k].LP ← true;
add VC to SL;
if (F[k].val = VC[k].val) then
    F[k].LP ← true;
eval_global();
```

To send a message m, the only action required by the algorithm is to send the vector clock VC , the first cut F and the message list ML with the algorithm. The structure corresponding to the message m is included

in the message list with the fields m.send, m.source and m.dest set appropriately. Formally,

When sending message m:
```
m.send ← VC[k].val;
ML ← ML ∪{m};
VC[k].val ← VC[k].val + 1;
send VC, F, and ML as part of the message;
```

On receiving a message m with the vector clock VC_m, the first cut F_m and the message list ML_m, P_k updates its own variables as follows. The vector clock is updated by taking the component-wise maximum as usual. It also updates its first cut by taking component-wise maximum with the received F. Any state in the state list SL which is before F is discarded. The message list is updated by taking the union of the message list with the list received in the message. All messages received before the global state F are discarded. At this point, P_k is ready to check if the global predicate is true. Formally,

On Receiving a message m with VC_m, F_m, and ML_m
```
m.rcvd ← VC[k].val;
VC ← max(VC, VC_m);
VC[k].val ← VC[k].val + 1;

F ← max(F, F_m);
SL← {VC' ∈ SL| (VC'[k].val > F[k].val) };
if (SL= NULL) then F ← VC;
else F ← max(F, first entry in SL);

ML← {m' ∈ (ML∪ ML_m) | F[m'.dest].val < m'.rcvd};
eval_global();
```

3.3 Evaluation of Global Predicate

To check whether the global predicate is true on F, we need to verify that all local predicates and all channel predicates are true. The flag global indicates whether the global predicate is true or not. It is initialized to true and set to false when it is determined that one of the local or the channel predicates is false. If it is found that the global predicate is false because of P_k then F needs to advance on P_k. This condition is captured by the flag advance. We already know that F is a consistent global state by our invariant. We next check that for each j, either F[j].LP is true or the process P_j is suspected in the global state F (i.e. SP(F, j)). In this manner we eliminate processes from consideration that are suspected to have failed. If this condition holds true then channel predicates are computed. Only those channels are considered which were not suspected in the cut F. First a subset of message list ML is constructed which includes only those messages which have been sent before F. Further, all messages that are suspected to be lost are removed from the message list. Now the channel predicate is evaluated. If a channel predicate evaluates to false, then by send-monotonicity of the channel predicate, we know that the state corresponding to the receiver is ineligible and we can advance the cut F by deleting the entry corresponding to the receiver. If all such channel predicates evaluate to true then the global predicate has been detected.

eval_global() at process P_k
```
var
  global: boolean initially true;
  advance: boolean initially false;

/* check for all local predicates */
for all j: ¬ SP(F,j) do
  if ¬ F[j].LP then
      global := false;
      if (j = k) then advance := true;

/* check for all channel predicates */
for all i, j : (i ≠ j) ∧ ¬ SC(F,(i, j)) do
  mm ← {m ∈ ML| m.source = i ∧ m.dest = j ∧
      m.send < F[i].val ∧¬ SM(F,m)};
    if channel predicate is false for mm then
        global := false;
        if (j = k) then advance := true;
        break;

if (global) then issue detected;
else if (advance) then
    if (SL= NULL) then F ← VC;
    else F ← max(F, first entry in SL);
```

We now show the correctness of our algorithm.

Lemma 4
1. F *for any process always represents a consistent global state.*
2. F *is eventually advanced if it does not satisfy the global predicate.*
3. Any global state which precedes F *does not satisfy the global predicate.*

Proof: 1. F is initialized to a consistent global state. Whenever it is modified, it is either set to VC, combined with the F from a message or the first entry in SL. Since VC always defines a consistent global state and combining two consistent global states produces a consistent global state[15], the result follows.

2. If a local predicate LP_k is false, then the process P_k will eventually advance its own F (see eval_global). All other processes will eventually receive a message with the new F and therefore advance their F as well. The similar reasoning applies when a channel predicate is false.

3. F at P_k is advanced only when the local predicate is false, the channel predicate for which P_k is a receiver is false or when combining with F received in a message. In the first two cases, by the definition of the local predicate and send-monotonicity, the lemma holds. In the last case it can be shown to hold by using induction. ■

Theorem 5 *The algorithm is* complete, *i.e. if the global predicate is true then the algorithm issues detected in spite of failures (so long as at least one process is alive.) Conversely, if the algorithm issues detected on a global state F, then the global predicate is true with the suspicions (i.e. if all suspicions are accurate then the global predicate is true on the cut F.)*

Proof : From Lemma 4 part 2 and part 3, F in any correct process will eventually be advanced to the global state in which the global predicate is true. Now due to the set decreasing property of the global predicate, it will evaluate to true even if there are incorrect suspicions of processes, channels or messages. Therefore, the algorithm is complete. The converse is obvious from the procedure eval_global. ■

4 Discussion

4.1 Overhead

The algorithm appends $O(N)$ integers for VC and F and $O(M)$ information for ML to every message where M is the number of messages sent by any process in the causal past which have not been received before F. In many cases, this overhead can be reduced. For example, when detecting the predicate "no token in system," only token messages need to be tracked. The computation overhead for maintaining VC and F is $O(N)$. The overhead to check whether channel predicates are true is $O(N^2)$ assuming that each channel predicate can be evaluated in $O(1)$ time. We have ignored overhead of suspicion list maintenance in this analysis.

In the algorithm presented above each process evaluates channel predicates for all the channels. This overhead can be reduced by requiring only the receiver of a channel to evaluate the predicate. Just as we have a tag with each component of F for the local predicate, we will then have a tag for the channel predicate as well. During the evaluation of the global predicate, a process would be required to check for all local predicates and channel predicates through these tags. This may, however, increase the latency of detection.

4.2 Action on Detection

Although our algorithm correctly detects all predicates which actually hold, it may detect a predicate which has not held in the past because of inaccurate suspicions. For some applications, a human can determine whether the predicate detection was true and take the corrective action. Sometimes we may want the corrective action to be taken automatically. For example, for the predicate "no token in system," we may wish to generate a new token. There are two problems with this approach. First, the detection of a lost token may be false. If a new token is generated, then there will be multiple tokens in the system. For some applications this may be okay if it is required that there be at least one token at all times and preferably only one for efficiency reasons. For example, if we require the process with the token to carry out some task (respond to a query) the computation will still be in good state after the false detection. Further, by keeping numbers with a token we can force processes to discard any old tokens.

The second problem is how to coordinate processes so that exactly one of them generates the new token. Unfortunately, it is impossible to guarantee this in an asynchronous environment in which a process can fail. Formally, define the problem of unique action as

> Devise an algorithm so that some process in the system performs an action and all others do not perform that action.

The next theorem demonstrates that implementing such an action is as difficult as implementing consensus.

Theorem 6 *There is no terminating protocol for the unique action problem.*

Proof: We show that the problem of the unique action is at least as difficult as the consensus problem in Fischer, Lynch, and Paterson [7]. Assuming that there exists a protocol for unique action, we solve the consensus problem as follows. The action we require from exactly one correct process is that it send its input value to all other processes in a special message called action message. By requiring all processes to output the value they receive in an action message, we meet all the requirements for consensus. ■

Thus, any protocol for unique action cannot both be safe and live. One solution is to use idempotent actions so that application of multiple actions is equivalent to a single action. For example in a process

control application the action may be to open some valve. If the valve is already open then performing the action again would not change anything.

4.3 Synchronous Environment

We define a synchronous environment as one which guarantees that all messages are received with a bounded delay. In a synchronous environment our failure detectors will eventually be accurate. This is because we increase our timeout period after any false suspicion and therefore the timeout period will eventually become greater than the bound on the delay. After this point of time all suspicions will be accurate. Since the loss of accuracy in our predicate detection is only due to inaccuracy of failure detection, it follows that eventually our predicate detection scheme will be both complete and accurate in a synchronous environment. Note that we only require that the message delays be bounded and not that the bound is known. No matter what the bound is, the algorithm will eventually make only accurate detections.

References

[1] K. P. Birman and R. V. Renesse. *Reliable Distributed Computing with the Isis Toolkit*, 1993.

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

[3] K. M. Chandy, and L. Lamport, "Distributed Snapshots: Determining global states of distributed systems," *ACM Transactions on Computer Systems*, no. 1, February 1985, pp. 63-75.

[4] R. Cooper and K. Marzullo, "Consistent Detection of Global Predicates," *Proc. of the ACM/ONR Workshop on Parallel and Distributed Debugging*, May 1991, pp. 163-173.

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

[6] C. J. Fidge. Partial orders for parallel debugging. *Proceedings of the ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging, published in ACM SIGPLAN Notices*, 24(1):183–194, January 1989.

[7] M. Fischer, N. Lynch, M. Paterson, "Impossibility of Distributed Consensus with One Faulty Process," *Journal of the ACM*, vol. 32, no.2, 1985, pp.374-382.

[8] V. K. Garg, C. M. Chase, R. Kilgore and J. R. Mitchell, "Efficient Detection of Channel Predicates in a Distributed System," *Journal of Parallel and Distributed Computing*, Vol. 45, No. 2, September 1997, pp. 134 – 147.

[9] V. K. Garg, C. Chase, "Distributed Algorithms for Detecting conjunctive Predicates," Proceedings of the Int'l Conference on Distributed Computing Systems, June 1995, pp. 423-430.

[10] V. K. Garg, and B. Waldecker, "Detection of Weak Unstable Predicates in Distributed Programs," *IEEE Trans. on Parallel and Distributed Systems*, Vol. 5, No. 3, Mar 1994, pp. 299-307.

[11] V.K. Garg, J. R. Mitchell, "Infinitely Often Accurate failure detectors in a faulty asynchronous environment," ECE dept, University of Texas, Tech. Report TR-PDS-1998-04.

[12] M. Hurfin, M. Mizuno, M. Raynal, M. Singhal, "Efficient Distributed Detection of Conjunctions of Local Predicates," IRISA technical report, PI-967, 1995.

[13] L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System," *Communications of the ACM*, vol. 21, no. 7, July 1978, pp. 558-565.

[14] P. Li, B. McMillin, "Fault-Tolerant Distributed Deadlock Detection/Resolution," *Proceedings of the 17th International COMPSAC*, November, 1993, pp. 224-230.

[15] F. Mattern, "Virtual Time and Global States of Distributed Systems," *Parallel and Distributed Algorithms*, 1989, pp. 215-226.

[16] J. R. Mitchell, "Algorithms for building fault tolerant distributed systems," Ph.D. dissertation, University of Texas at Austin, 1997.

[17] Y.-C. Tseng. Detecting termination by weight-throwing in a faulty distributed system. *Journal of Parallel and Distributed Computing*, 25:7–15, 1995.

[18] S. Venkatesan, "Reliable protocols for distributed termination detection," *IEEE Transactions on Reliability,* Vol. 38, No.1, April 1989, pp 103–110.

[19] M. Woods. Fault-tolerant management of distributed applications using the reactive system architecture. *Ph.D. dissertation*, January 1992.