Implementable Failure Detectors in Asynchronous Systems

Vijay K. Garg, J. Roger Mitchell

TR-PDS-1998-004 **May 1998**

Parallel & Distributed Systems group

Department of Electrical & Computer Engineering

University of Texas at Austin

Austin, Texas 78712

Implementable Failure Detectors in Asynchronous Systems

Vijay K. Garg[*] J. Roger Mitchell[†]

Parallel and Distributed Systems Laboratory
Electrical and Computer Engineering Department
The University of Texas at Austin,
Austin, TX 78712
email: garg@ece.utexas.edu
Fax: (512) 471-5907
http://maple.ece.utexas.edu

May 18, 1998

Abstract

Failure detection is one of the most fundamental modules of any fault-tolerant distributed system. The failure detectors discussed in the literature so far are either impossible to implement in an asynchronous system, or their exact guarantees have not been discussed. We introduce a failure detector called *infinitely often accurate* failure detector which can be implemented in an asynchronous system. We provide one such implementation and show its application to the fault-tolerant server maintenance problem. We also show that some natural timeout based failure detectors implemented on Unix are not sufficient to guarantee infinitely often accuracy.

1 Introduction

Failure detection is one of the most fundamental modules of any fault-tolerant distributed system. Typically, a fault-tolerant distributed system uses one of the two techniques - maintain replicas of a process (for example in [BE94]), or use primary-backup approach [BMST93]. Both of these approaches rely on detection of failures. While most commercial fault-tolerant systems use some sort of failure detection facility typically based on timeouts, we show that many of them are inadequate in providing accuracy guarantees in failure detection.

The first systematic study of failure detection in asynchronous systems was done by [CHT92, CT96]. The weakest failure detector they studied, called $\diamond \mathcal{W}$, can be used to solve the consensus problem in asynchronous systems. It follows from [FLP85] that failure detectors in [CT96] are not implementable in asynchronous systems. The failure detector introduced in this paper,

[*]supported in part by the NSF Grants ECS-9414780, CCR-9520540, a TRW faculty assistantship award, a General Motors Fellowship, and an IBM grant

[†]supported in part by a Virginia & Ernest Cockrell fellowship

called Infinitely Often Accurate detector (IO detector for short), can be implemented efficiently in asynchronous systems.

An IO detector is required to satisfy even weaker accuracy than eventually weak accuracy proposed by Chandra and Toueg [CT96]. Intuitively, eventually weak accuracy requires that for all runs the detector eventually never suspects at least one correct process. It is precisely this requirement which makes it possible to solve the consensus problem by using, for example, the rotating coordinator technique[CT96]. On the other hand, this is also the requirement which is impossible to implement in an asynchronous system. An IO detector only requires that a correct process is not permanently suspected. By assuming that a channel delivers the messages infinitely often, this requirement can be easily met. In fact, the algorithm is based on an earlier work by Dwork et al. [DLS88]. However, as mentioned earlier the algorithm does not guarantee properties of $\diamond \mathcal{W}$. Our contribution lies in formalizing the exact properties guaranteed by that algorithm and showing its usefulness in asynchronous systems. We also show that some other natural "timeout" implementation of failure detectors for Unix [HK95] and other systems[Bec91] do not satisfy the properties of an IO detector.

Although no algorithm can solve the consensus problem in an asynchronous system for all runs, it is desirable that the processes reach agreement in finite time for those runs which satisfy *partial synchrony* condition. Following [DLS88], a run is defined as partially synchronous if there exists a non-failed process such that eventually all messages sent by that process reach their destinations in bounded delay. We show that for these runs our failure detector will provide eventual weak accuracy and hence processes using the approach in [CT96] will reach agreement in finite time. Thus, our failure detector provides *conditional* eventual weak accuracy guarantee. In other words, whereas $\diamond \mathcal{W}$ insists on eventual weak accuracy for all runs, we require eventual weak accuracy only in partially synchronous runs and infinitely often accuracy for all runs.

While the IO detector cannot be used to solve the consensus problem, we show that it is useful for other applications. In particular, we give its application to a fault-tolerant server maintenance problem. This problem requires presence of at least one server during the computation. Our solution works in spite of up to $N-1$ failures in a system of N processes.

This paper is organized as follows. Section 2 provides our model of an asynchronous system and failures and describes the properties guaranteed by an IO detector. Section 3 compares IO detector with the $\diamond \mathcal{W}$ failure detector. Section 4 describes applications of IO detector. Finally Section 5 describes an IO detector which can be used when an upper bound on the number of processes that may fail is known.

2 IO Failure Detectors

We assume the usual model of an asynchronous system. The message delays are unbounded but finite. A *run* of any algorithm in an asynchronous system results in a set of *local* states denoted by S. We use s, t and u to denote the local states of processes and symbols i, j and k to denote process indices. The notation $s.v$ denotes the value of the variable v in the local state s. The relation $s < t$ means that s and t are states at the same process and s occurred before t. The relation $s \leq t$ means that $s < t$ 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 [Lam78]. Therefore, $s \rightarrow t$ is the smallest transitive relation on S satisfying 1) $s < t$ 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 a set of local states, one from each process, such that no two of them are related by the happened-before relation. We use symbols G and H to denote global states and $G[i]$ to denote the state of the process i in the global state G. We say that $G \preceq H$ (or $H \succeq G$) iff $\forall i : G[i] \leq H[i]$.

A processor crashes by ceasing all its activities. 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 denote the set of all processes and the set of correct processes by Π and Π_c respectively. The set $C \subseteq S$ denotes the set of states on the correct processes and the set C_j denotes the states for any correct process j.

The predicate $s.suspects[i]$ holds if the process i is suspected in the state s (by the process which contains s).

A failure detector is responsible for maintaining the value of the predicate $suspects[i]$ at all processes. The value of $suspects[i]$ is true at P_k, if P_k suspects that P_i has failed. We would like our failure detectors to satisfy certain completeness and accuracy properties of these suspicions. An IO failure detector (or IO detector, for short) is formally defined below.

Definition 1 *An IO failure detector is a failure detector that satisfies strong completeness, infinitely often accuracy, conditional eventual weak accuracy and perfect local suspicions.*

These properties are described next.

2.1 Strong Completeness Property

Let the predicate $permsusp(s, i)$ be defined as

$$permsusp(s,i) \equiv \forall t \geq s : t.suspects[i]$$

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

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

The *weak completeness* property requires that every failed process is eventually permanently suspected by some correct process. Thus, a detector is defined to be weak complete if for all runs,

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

Although the strong completeness property requires permanent suspicion by all correct processes and not just some correct process, it is easy to implement a detector that provides strong completeness given a detector that implements weak completeness while preserving accuracy. Informally, this can be achieved by requiring processes to periodically broadcast their suspicion lists. More details of this approach can be found in [CT96]. Since strong completeness is quite useful in design of distributed algorithms, and is easily achieved by simple timeout mechanisms, we require that IO detectors satisfy strong completeness.

2.2 Infinite Often Accuracy

In [CT96], four accuracy properties have been presented. The weakest of these properties is *eventual weak accuracy*. A detector satisfies eventual weak accuracy if for all runs eventually some correct process is never suspected by any correct process. Formally, for all runs

$$\exists i \in \Pi_c, \forall j \in \Pi_c, \exists s \in C_j, \forall t \geq s : \neg t.suspects[i]$$

However, as shown by [CT96] 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. This implies that $\diamond \mathcal{W}$ is impossible to implement in an asynchronous system. 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 for all runs*

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

Note that infinitely often accuracy is simply the converse of the weak completeness requirement. By combining the two properties, we get the following pleasant property of an IO detector.

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

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

2.3 Conditional Eventual Weak accuracy

We now introduce another useful property of failure detectors which is also implementable in an asynchronous environment. The property says that if we happen to be lucky in *some* asynchronous run in the sense that eventually all messages sent by at least one process reach under some bound then the failure detector will eventually be accurate with respect to that process *in that run*. We call this property *conditional eventual weak accuracy*. We first introduce the notion of a *partially synchronous* run. This notion is similar to that used in [DLP$^+$86].

Definition 2 *A run is partially synchronous if there exists a state s in a correct process P_i and a bound δ such that all messages sent by P_i after s take at most δ units of time.*

In the above definition we do not require the knowledge of P_i, s, or δ. This lets us define:

Definition 3 *A failure detector satisfies conditional eventual weak accuracy if for all partially synchronous runs it satisfies eventual weak accuracy.*

Observe that the conditional eventual weak accuracy is weaker than the eventual weak accuracy. The former requires that the failure detector satisfy eventual weak accuracy only when the run is partially synchronous, while the latter requires it for all runs.

2.4 Perfect local suspicion

We also require that a correct process should never suspect itself. This property is trivial to satisfy. A process never sets its own suspicion to true.

3 Comparison of IO detector with $\diamond \mathcal{W}$

How does IO detector compare to $\diamond \mathcal{W}$? First note that if a failure detector satisfies weak accuracy, then it does not necessarily satisfy infinitely often accuracy. The detector may be accurate with respect to one process but permanently suspect some other correct process. However, another way to compare failure detectors is by using the notion of reduction between detectors as introduced by [CT96]. We show that an IO detector can be implemented using a $\diamond \mathcal{W}$ detector.

Our implementation of an IO detector from a $\diamond \mathcal{W}$ detector is in two stages. In the first stage we build a $\diamond \mathcal{S}$ detector from $\diamond \mathcal{W}$. A $\diamond \mathcal{S}$ detector [CT96] is a failure detector that satisfies strong completeness and eventual weak accuracy. A $\diamond \mathcal{S}$ detector can be easily built using $\diamond \mathcal{W}$ detector as shown in [CT96]. So, we assume that a $\diamond \mathcal{S}$ detector is available to us. Let $IO.suspects$ and $ES.suspects$ be the set of processes suspected by the IO detector and $\diamond \mathcal{S}$ detector respectively. The algorithm to implement IO detector using $\diamond \mathcal{S}$ is given in Figure 1. The detector is based on two activities. Each process sends a message "alive" to all other processes infinitely often. On receiving such a message from P_i, a process P_j queries the $\diamond \mathcal{S}$ suspector and removes the suspicion of P_i and P_j.

```
$P_j$::
var
$IO.suspects$: set of processes initially $ES.suspects - \{j\}$;

(I1) send "alive" to all processes infinitely often;

(I2) On receiving "alive" from $P_i$;
        $IO.suspects := ES.suspects - \{i, j\}$;
```

Figure 1: Implementation of an IO detector using a $\diamond \mathcal{S}$ detector

We now have the following Lemma.

Lemma 2 *The algorithm in Fig. 1 implements an IO detector.*

Proof: We first show the strong completeness property. Consider any process P_i that has failed. This implies that eventually P_i will be in $ES.suspects$ for P_j, by the property of $\diamond \mathcal{S}$ detector. Further, eventually P_j will stop receiving "alive" message from P_i. This implies that P_i will be permanently in $IO.suspects$ of P_j.

We now show infinitely often accuracy property. Consider any process P_i that has not failed. This implies that any correct process will receive "alive" message from P_i infinitely often. Therefore, P_i will be not be in $IO.suspects$ infinitely often.

We show that the algorithm satisfies *eventual weak accuracy* and therefore *conditional eventual weak accuracy*. The detector only removes suspicion from the set $ES.suspects$. Since $\diamond \mathcal{S}$ satisfies eventual weak accuracy it follows that the IO detector built from $\diamond \mathcal{S}$ also satisfies this property.

Finally, it satisfies perfect local suspicion since j is never in the suspicion list of P_j.

■

We now consider the converse question. Is there an asynchronous algorithm that implements $\diamond \mathcal{W}$ using IO-detector? We answer this question in negative by giving an implementation of an IO-detector. A possible implementation is shown in Fig. 2. This implementation is similar to that proposed by [DLS88]. 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 suspected, the timeout period for that process is increased.

```
$P_j$::
var
    $IO.suspects$ : set of processes initially $\emptyset$;
    $timeout$: array[1..N] of integer initially t;
    $watch$: timer initially set to timeout;

(A1) send "alive" to all processes after every $t$ units;

(A2) On receiving "alive" from $P_i$;
    if $i \in IO.suspects$ then
        $IO.suspects := IO.suspects - \{i\}$;
        $timeout$[i]++;
    endif;
    Set $watch[i]$ timer for $timeout[i]$;

(A3) on expiry of watch[i]
    $IO.suspects := IO.suspects \cup \{i\}$;
```

Figure 2: Implementation of an IO detector

Theorem 1 *The algorithm in Fig.2 implements an IO detector.*

Proof: First we show the strong completeness property. If a process P_i has failed then it will stop sending messages. All the messages sent by it will eventually be received. After that point, none of the other processes will hear from this process and therefore will start suspecting P_i using rule (A3). Since they will never hear from it again, it will then be permanently in $IO.suspects$.

The property of infinitely often accuracy follows from the proof of Lemma 2.

The algorithm also satisfies conditional eventual weak accuracy. Consider any partially synchronous run. Let P_i be the correct process in that run for which messages obey the partial synchrony condition after some state s. Since messages sent after that state are received in less than δ units of time, there can only be a bounded number of false suspicions of P_i by any process (because the timeout period is increased by 1 after every false suspicion). Thus, eventually there is a time after which P_i is never suspected by any process.

Finally, a process never suspects itself and therefore perfect local suspicion is true.

■

Observe that one needs to be careful with designing algorithms for failure detectors. Some natural approaches do not satisfy the IO-property. For example, consider the following approach taken from [Bec91].

> This crash detection manager is responsible to multicast polling messages periodically to all other processes under surveillance. The other processes are expected to reply to these polling requests with "I am alive" messages immediately. If the answer is missing for three times consecutively, the crash manager assumes that this process has crashed.

```
$P_j$::
var
    watchd.suspects: set of processes initially $\emptyset$;

(B1) infinitely often broadcast a query message to all processes

(B2) After broadcast wait for timeout[i] time units (timeout period)
    watchd.suspects := { $P_i | P_i$ did not respond to the query in the timeout period }

(B3) On receiving a query from $P_i$
    send "I am alive" to $P_i$
```

Figure 3: A Detector that does not satisfy IO accuracy

The crash detection manager in [Bec91] does not satisfy infinitely often accuracy since it may permanently suspect a correct process. As another example, consider the algorithm in Fig. 3 which is similar to the *watchd* process implemented on Unix and reported in [HK95]. It is tempting to implement failure detectors using the algorithm in Fig. 3 since it only requires the failure detector to listen for incoming messages during the timeout interval. This algorithm, however, does not satisfy IO accuracy. It may suspect some process P_i at all times even when P_i is alive and just slow.

4 Applications of IO failure detectors

Since IO detectors are implementable, it is clear that they cannot be used for solving the consensus problem in an asynchronous system. What good are they then? We now discuss a practical problem that can be solved using IO detectors. Consider a service that is required in a distributed system consisting of N processes. We require that at least one process always act as the provider of that service. As a simple example, assume that the servers are stateless and any request from the client is broadcast to all servers. The problem requires that at least one (preferably exactly one) server respond to the request. We abstract this requirement using the concept of a token. Any process that has a token considers itself as the provider of the service. Since the process holding a token may fail, we clearly need a mechanism to regenerate a token to avoid interruption of the service. To avoid triviality, we only consider those runs in which at most $N-1$ out of N processes fail. The following predicates and functions are used for specifying the requirements.

- $G[i].token$: This predicate is true if process P_i has a token in the global state G.
- $hastoken(G)$: This predicate is true if the global state G has a token. It may have multiple tokens.
$$hastoken(G) \equiv \exists i : G[i].token$$
- $noduplicate(G)$: This predicate is true if the global state G has at most one token. It may have none.
$$noduplicate(G) \equiv \forall i,j : i \neq j : \neg G[i].token \vee \neg G[j].token$$
- $boundedafter(G,i)$: This predicate is true if there exists a bound δ such that all messages sent by P_i after G take at most δ units of time.

Ideally, we would like that all global states to have exactly one token. However, even the weaker requirement that eventually there exists exactly one token is impossible to implement in an asynchronous system. Therefore, we consider a weaker set of requirements given below.

1. *Availability:* There exists a global state such that all later global states have at least one token. Formally,
$$\exists G, \forall H \succeq G : hastoken(H)$$
2. *Efficiency:* For every global state G in which two different processes have tokens, there exists a later global state in which the token from at least one of the processes is removed. Formally,
$$\forall G,i,j : \langle (i \neq j) \wedge G[i].token \wedge G[j].token \Rightarrow \exists H \succeq G : \neg H[i].token \vee \neg H[j].token \rangle$$
3. *Eventually exactly one token under partial synchrony:* If after any global state G all messages sent by some correct process P_i arrive in less than a pre-determined bounded delay, then

there exists a later global state H such that all global states after H have exactly one token. Formally,

$$\forall G : \mathit{boundedafter}(G, i) \Rightarrow \exists H, \forall H' \succeq H : \mathit{hastoken}(H') \wedge \mathit{noduplicate}(H')$$

In practice, for most cases the partial synchrony condition would be true and therefore eventually we will have exactly one token. However, when the partial synchrony is not met we would still have properties of availability and efficiency. This illustrates the methodology that we propose for asynchronous algorithms. They provide useful guarantees even when the run is not well behaved and provide more desirable guarantees when the run is well behaved (i.e. partially synchronous).

We will present our solution in stages. The algorithm which satisfies availability and efficiency is given by the following rule. In a state s, process P_i is assumed to have a token if all processes with smaller indices than i are suspected and P_i is not suspected. Formally,

$$s.\mathit{token}(i) \equiv \forall j : j < i : s.\mathit{suspects}[j] \wedge \neg s.\mathit{suspects}[i].$$

To see that the algorithm satisfies availability, consider in any run the global state G after which there are no failures. By our assumption, at least one process is alive in the run. Let P_k be the smallest such process. By strong completeness of the IO failure detector, eventually P_k will suspect all smaller processes. In that global state P_k will generate a token.

To see that the algorithm satisfies efficiency, consider the global state G in which two processes P_i and P_j $(i < j)$ have tokens. For any continuation of run after G, in which P_i or P_j fails we have a global state H in which $\neg H[i].\mathit{token} \vee \neg H[j].\mathit{token}$. Otherwise, by the IO accuracy property of IO detector, P_j will remove the suspicion of P_i eventually. In that global state H, $\neg H[j].\mathit{token}$ holds.

The algorithm does not satisfy the property of exactly one token under partial synchrony. For example, in a system with two processes P_1 and P_2 in which partial synchrony is true only for process P_2, both processes will have a token infinitely often. Note that in this algorithm if the partial synchrony condition is satisfied with respect to the smallest correct process, then the system works in the desirable manner. There is exactly one token eventually. To see this consider any global state G such that $\mathit{boundedafter}(G, i)$ holds for the smallest correct process. This implies that IO detector for any process P_j that does not fail will eventually never suspect P_i. Therefore, P_j will never get the token.

We now present a solution which meets all our requirements. Intuitively, the idea behind the algorithm is as follows. In addition to the suspicion list maintained by the failure detector, each process maintains a timestamp for each process called *ticket time*. The ticket time of a process P_k is the logical time when it was suspected by some process P_i such that according to P_i, P_k had a token before the suspicion and P_i has the token after the suspicion. When P_i suspects a process P_k which it thinks has the token, it records the logical time of this event. This is the ticket time of process P_k. P_i will then send out a message to all processes informing them of the suspicion along with this ticket time.

Process P_i has a token if all processes that are currently not suspected by P_i have ticket times that are greater than that of P_i. If a process with a token is suspected its ticket time will become greater than all other ticket times. This way the token is moved from a slow process to the next process which is alive. The formal description of the algorithm is given in Fig. 4. The algorithm assumes that all channels are FIFO.

```
$P_i$:
var
    ticket:array[1..N] of (integer,integer) initially $\forall i : ticket[i] = (0, i)$

    suspected: array[1..N] of boolean; /* set by the failure detector */

    $token(k) \equiv (\forall j \neq k : suspected[j] \vee (ticket[j] > ticket[k])) \wedge \neg suspected[k]$

    (R1) Upon change from unsuspicion to suspicion of $P_k$ with $token(k)$
        if $token(i)$ then
            $ticket[k] := Lamport's\_logical\_clock$;
            send "slow", k, ticket[k] to all processes

    (R2) Upon receiving "slow",k,t
        $ticket[k] := max(ticket[k], t)$;
```

Figure 4: Algorithm for Alive Token Problem

Lemma 3
$\forall G, i : \exists j : G[i].token(j)$

Proof: Since a process never suspects itself, the set of unsuspected processes at $G[i]$ is non-empty. The unsuspected process with the smallest ticket number will then have the token according to P_i.

■

Note than in the context of implementing fault-tolerant server, a server P_i will respond to the client in the global state G if

$$G[i].token(i)$$

holds. We say that P_i has a token if this condition is true. Note that Lemma 3 only says that each process always thinks that there is at least one process which has the token. However, this condition does not imply that there exists a process which thinks that *it* has the token. In a faulty algorithm, it is quite possible that each process thinks that somebody else has the token.

Theorem 2 *The algorithm in Fig. 4 satisfies all three required properties.*

Proof:

- *Availability*: First consider the global state G after which no failures occur. We show that if there is a process with a token in G, then some process will always have a token. Since there is perfect local suspicion, a process with a token can lose it only if its own ticket time increases or somebody else's ticket time decreases. Since the ticket time for any component can only increase, it follows that a process can lose the token only if its own ticket time increases. However, this can happen only when a process with a token (after the suspicion) sends it a "slow" message. Thus, in this case some other process has a token.

  Now, consider the scenario when failures occur. If a process that does not have a token fails then no harm is done. Our obligation is to show that if a process with a token fails and there are no more tokens in the system then a token is regenerated. Of all the processes that are correct consider the process with the smallest ticket. By strong completeness, this process will eventually suspect all the failed processes. At that point, it will have a token.

- *Efficiency*: Consider the global state G in which two processes P_i and P_j have tokens. Since there is a total order on all tickets, and due to Lemma 3, this can only happen when the process with the smaller ticket number, say P_i is suspected by the other process, P_j. For any continuation of run after G, in which P_i or P_j fails we have a global state state H in which $\neg H[i].token \vee \neg H[j].token$. Otherwise, by the IO accuracy property of IO detector, P_j will eventually remove the suspicion of P_i. In that global state H, $\neg H[j].token$ holds.

- *Eventually exactly one token under partial synchrony*: Consider any global state G such that $boundedafter(G, i)$ holds. This implies that IO detector for any process P_j will eventually never suspect P_i. If there are multiple values of i such that $boundedafter(G, i)$ holds, then we choose the process with the smallest ticket number in G. This process is never suspected by anybody else after G and therefore no other process will ever have a token. Further, this process will never lose the token since no process which is unsuspected can have a smaller ticket number.

■

Remark: If suspicions are perfect, that is, a process is suspected only when it is failed, the algorithm ensures that there is at most one token. Thus, the algorithm can also be seen as a fault-tolerant mutual exclusion algorithm with perfect failure detectors.

5 Failure detectors with small suspicion list

So far we had assumed that up to $N-1$ out of N processes may fail. If we assume that at most f processes may fail, then we can build a failure detector which will never suspect more than f processes and yet provide strong completeness, infinitely often accuracy, and conditional eventual weak accuracy. An implementation of this detector, called f-IO detector, is shown in Fig. 5. The f-IO detector assumes an implementation of an IO detector. Note that when f equals $N-1$, it is equivalent to IO detector (because a process never suspects itself).

The algorithm for f-IO detector maintains a queue of slow processes in addition to the list of suspected processes. When a process is suspected it is added to the list only if the size of the list is less than f; otherwise, it is inserted in the queue of slow processes. When a process is unsuspected if the slow queue was non-empty, the process at the head is removed from the queue and inserted in the suspect list.

```
$P_j$::
var
    $f$-IO.suspects : set of processes initially $\emptyset$;
    slow : FIFO queue of processes initially empty;

(A1) On change of IO.suspects[k] from false to true and not inqueue(slow,k)
    if $|f$-IO.suspects $| < f$ then
        $f$-IO.suspects[k]:= true;
    else
        insert(slow,k);

(A2) On change of IO.suspects[k] from true to false
    if $f$-IO.suspects[k] then
        $f$-IO.suspects[k] := false;
        if not empty(slow)
            j := deletehead(slow);
            $f$-IO.suspects[j] := true;
    else
        delete(slow,k);
```

Figure 5: An IO Detector with small suspicion list

Theorem 4 *The algorithm provides all properties of IO detector assuming there are no more than f failures. Further, it does not suspect more than f processes at any time.*

Proof:
Strong completeness: If a process has failed it will eventually be permanently suspected by IO.suspects. If it is put in the suspicion list, it will never be removed. If it is kept in the slow queue, then we show that the number of processes ahead in the slow queue will decrease by 1. By our assumption that at most f processes fail, and there are f suspected processes, there is at least one process which is suspected incorrectly. This is because one of the failed process is in the slow queue. By IO accuracy the process that is incorrectly suspected will eventually be unsuspected. At that point the head of the slow queue will be removed.
IO-accuracy of f-IO detector: Follows from IO-accuracy of IO detector.
Conditional eventual weak accuracy: Again follows from conditional eventual weak accuracy of IO detector.

■

Remark: When f is 1, the f-IO detector gives us the following property which is stronger than infinitely often accuracy

(GIO) Infinitely often there exists a correct process which is not suspected by any correct process.

The property GIO can be shown as follows. Since each process is allowed to suspect at most one process, the only configuration in which there is no common unsuspected process is when suspicion of all processes are different. This configuration cannot hold permanently because of infinitely often accuracy.

References

[BE94] Kenneth Birman and Robbert Van Renesse (Editors). *Reliable Distributed Computing with the Isis Toolkit.* IEEE Computer Society Press, 1994.

[Bec91] Thomas Becker. Keeping processes under surveillance. In *Symposium on Reilable Distributed Systems*, pages 198 – 205. IEEE, 1991.

[BMST93] Navin Budhiraja, Keith Marzullo, Fred Scneider, and Sam Toueg. *The Primary-Backup Approach*, chapter 8. ACM Press, Frontier Series. (S.J. Mullender Ed.), 1993.

[CHT92] Tushar Deepak Chandra, Vassos Hadzilacos, and Sam Toueg. The weakest failure detector for solving consensus. In *Proc. of the 11th ACM Symposium on Principles of Distributed Computing*, pages 147–158, August 1992.

[CT96] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, March 1996.

[DLP$^+$86] Danny Dolev, Nancy A. Lynch, Shlomit S. Pinter, Eugene W. Stark, and William E. Weihl. Reaching approximate agreement in the presence of faults. *Journal of the ACM*, 33(3):499–516, July 1986.

[DLS88] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):288–323, April 1988.

[FLP85] M. J. Fischer, N. Lynch, and M. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2), April 1985.

[HK95] Y. Huang and C. Kintala. Software fault tolerance in the application layer. In Michael Lyu, editor, *Software Fault Tolerance*, pages 249–278. Wiley, Trends in Software, 1995.

[Lam78] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.