Transient Fault Detectors

(Extended Abstract)

Joffroy Beauquier[1] Sylvie Delaët[1] Shlomi Dolev[2*] Sébastien Tixeuil[1]

[1] Laboratoire de Recherche en Informatique, Bâtiment 490, Université de Paris Sud, F91405 Orsay Cedex, France. Email: {jb, delaet, tixeuil}@lri.fr.

[2] Department of Mathematics and Computer Science, Ben-Gurion University, Beer-Sheva, 84105, Israel. Email: dolev@cs.bgu.ac.il.

Abstract. In this paper we present failure detectors that detect transient failures, i.e. corruption of the system state without corrupting the program of the processors. We distinguish *task* which is the problem to solve, from *implementation* which is the algorithm that solves the problem. A task is specified as a desired output of the distributed system. The mechanism used to produce this output is not a concern of the task but a concern of the implementation.

In addition we are able to classify both the *distance locality* and the *history locality* property of tasks. The distance locality is related to the diameter of the system configuration that a failure detector has to maintain in order to detect a transient fault. The history locality is related to the number of consecutive system configurations that a failure detector has to maintain in order to detect a transient fault.

1 Introduction

In a system that may experience transient faults it is impossible for the processors to "know" that the system is currently in a consistent state: assume that every processor has a boolean variable that is true whenever the processor knows that the system is in a consistent state and is false otherwise. The value of this variable may not reflect the situation of the system since it is a subject for transient faults. This is the reason that the processors in self-stabilizing systems must continue the execution of the algorithm forever and never know for sure that the system is stabilized.

In this paper we propose a tool for identifying the inconsistency of a system, namely a transient failure detector. The requirement that every processor will know whether the system is in a consistent state is relaxed, instead we require that at least one processor identifies the occurrence of a fault when the system is in an inconsistent state. Moreover, the transient failure detector is unreliable since it can detect inconsistent state as a result of a transient fault that corrupt the state of the failure detector itself. The only guarantees we have is that when

* Part of this research was done while visiting the Laboratoire de Recherche en Informatique, Bâtiment 490, Université de Paris Sud. Partly supported by the Israeli ministry of science and arts grant #6756195.

the system is not in a consistent sate a failure is detected, and when both the system and the failure detectors are in a consistent state no failure is detected.

Our focus in this paper is in the implementation of failure detectors and not in the operations invoked as a result of detecting a fault; we just mention two such possible operations, namely: resetting (e.g., [4]) and repairing (e.g., [8, 13, 1, 16]).

In this paper we present failure detectors that detect transient failures, i.e. corruption of the system state without corrupting the program of the processors. We distinguish *task* which is the problem to solve, from *implementation* which is the algorithm that solves the problem. A task is specified as a desired output of the distributed system. The mechanism used to produce this output is not a concern of the task but a concern of the implementation. We study transient fault detectors for tasks and for implementations, separately. Designing failure detectors for tasks (and not for a specific implementation) gives the implementation designers the flexibility of changing the implementation without modifying the failure detector.

In addition we are able to classify both the *distance locality* and the *history locality* property of tasks. The distance locality is related to the diameter of the system configuration that a processor has to maintain in order to detect a transient fault. The history locality is related to the number of consecutive system configurations that a processor has to maintain in order to detect a transient fault. Both the distance and the history locality of a task may give the implementation designer hints concerning the techniques and resources that are required for implementing the task.

Then we turn to investigate failure detectors for a specific implementation — specific algorithm. Obviously, one may use a failure detector for the task of the algorithm without considering the data structure and techniques used by the algorithm. However, we are able to show that in many cases the amount of resources required is dramatically reduced when we use failure detector for a specific implementation and not a failure detector for the task.

Related work: The term failure detector was introduced in a different context in [5], where an abstract failure detector is used for coping with asynchrony and solving consensus. In the context of self-stabilizing systems checking the consistency of a distributed system was used in [15] where a snapshot of the system is repeatedly collected. Failure detectors, called observers, that are initialized correctly and are not subject to state corruption are used in [17]. Monitoring consistency *locally* for a restricted set of *algorithms* has been suggested in e.g., [3, 2]. A local monitoring scheme for every on-line and off-line algorithm has been presented in [1]. The local monitoring technique of [1] is a general technique that monitors the consistency of any *algorithm*. The method of [1] uses pyramids of snapshots and therefore the memory requirement of each processor is related to the size of the system. In this work we present a hierarchy of failure detectors for *tasks* and *algorithms* that is based on the amount of information used by the failure detector.

The rest of the paper is organized as follows: The system is described in Section 2, failure detectors for asynchronous silent tasks are considered in Section 3 and for synchronous non silent tasks in Section 4, respectively. Failure detectors for implementation (algorithms) are presented in Section 5. Implementation details of transient failure detectors appear in Section 6 and concluding remarks are presented in Section 7. Most of the proofs are omitted from this extended abstract.

2 The System

Distributed system: Our system settings is similar to the one presented in [10]. We consider an asynchronous system of n processors, each processor resides on a distinct node of the system's *communication graph* $G(V, E)$, where V is the set of *vertices* and E is the set of *edges*. Two processors that are connected by an edge of G are *neighbors*. Communication among neighboring processors is carried out by *communication registers*. In the sequel we use the term registers for communication registers. An edge (i, j) of G stands for two registers $r_{i,j}$ and $r_{j,i}$. P_i (P_j) can write in $r_{i,j}$ ($r_{j,i}$, respectively) and both processors can read both registers. The registers in which a processor P_i writes are the registers of P_i.

Configurations and runs: Each processor is a finite state machine whose program is composed of *steps*. Processors have unique identifiers. The *state* of a processor consists of the values of its internal variables. A *configuration*, $c \in (S_1 \times S_2 \times \cdots \times R_1 \times R_2 \times \cdots \times R_m)$ is a vector of the states of all the processors and the values of all the registers.

An *asynchronous run* of the system is a finite or infinite sequence of configurations $R = (c_1, c_2, \cdots)$ such that c_{i+1} is reached from c_i by a step of one processor. In such a step, a processor may execute internal computations followed by a read or write operation. This scheduling policy is known as the *read-write* atomicity model. We use the *round complexity* (See [11, 8, 7]) to measure time in asynchronous system. The first round of a run is the minimal prefix of the run in which each processors reads the registers of all its neighbors and writes to its registers[3]. The second round is the first round of the rest of the run, and so on.

A *synchronous run* of the system is a finite or infinite sequence of configurations $R = (c_1, c_2, \cdots)$ such that c_{i+1} is reached from c_i by the following steps of the processors: first every processor reads the register of its neighbors, once every processor finishes reading, the processors change state and write into their registers.

Specifications and failure detectors: An *abstract run* is a run in which only the values of a subset of the state variables, called the *output variables* are shown

[3] The round complexity is sometime defined by the execution of a single step of every processor, note that $O(\Delta)$ such rounds are equivalent to our definition, where Δ is the maximal degree of a node.

in each configuration. The *specification* P of a task T for a given system $\mathcal{S}$ is a (possibly infinite) set of abstract runs of $\mathcal{S}$. For example, the mutual execution task is defined by a set of abstract runs, such that in each run in this set at most one processor executes the critical section at a time and every processor executes the critical section infinitely often — it is assumed that an output boolean variable that indicates whether the processor is executing the critical section exists.

$\mathcal{S}$ is *self-stabilizing* (in fact *pseudo self-stabilizing* [14]) with relation to P if and only if each of its runs has a non empty suffix in P. The aim of a *failure detector* is to check if a particular run of the system $\mathcal{S}$ matches the specification P. More precisely, a failure detector is assimilated to a boolean variable that obtains the value *false* if the specification is not satisfied and *true* otherwise. We use the convention in which once a failure is detected i.e., a failure detector returns *false*, the output of the failure detector is not changed until an external process resets the operation of the failure detector.

We assume that each processor P_i maintains a variable $\mathcal{V}_i^d[1..s]$ for some s, called the *array of views* of P_i. $\mathcal{V}_i^d[j]$, is the view of P_i on every component of the system up to distance d from itself, the view $\mathcal{V}_i^d[j]$ is related to time j. In more details, the variable $\mathcal{V}_i^d[j]$, $1 \leq j \leq s$, contains the communication graph description up to distance d from P_i and a portion of the state of each processor in this graph description. Note that since we are interested in tasks, the portion of the state that we are interested in, is the portion that defines the output of the system. At this stage we do not concern ourselves with the way each processor obtains and maintains the value of the variables in $\mathcal{V}_i^d[1..s]$, later in Section 6 we present ways to implement failure detectors.

It turned out that the fault detection capabilities of a failure detector is related to the amount of information it stores. We distinguish two parameters that are related to the storage used by a failure detector:

- Distance — the distance d of $\mathcal{V}_i^d[1..s]$, where d is in between 0 and $r+1$, and r is the radius of the system.
- History — the number s of views in the array $\mathcal{V}_i^d[1..s]$.

A *failure detector at a processor* P_i is a process executed by P_i that we denote $\mathcal{FD}_d^s(P_i)$. $\mathcal{FD}_d^s(P_i)$ is responsible to repeatedly communicate to every neighbor P_j of P_i the portion of $\mathcal{V}_i^d[1..s]$ *that is shared* with $\mathcal{V}_j^d[1..s]$. $\mathcal{FD}_d^s(P_i)$ is also responsible to check whether the shared portions of $\mathcal{V}_i^d[1..s]$ and $\mathcal{V}_j^d[1..s]$ agree and satisfy a certain predicate.

A failure detector of the system is the combination of the failure detectors of all the processors in the system, denoted $\mathcal{FD}_d^s$.

A task is (d,s)-local if and only if the following conditions are satisfied:

1. For each run R of the system $\mathcal{S}$ and for all $1 \leq i \leq n$ $\mathcal{FD}_d^s(P_i)$ returns *true* if R is correct (i.e. matches the task specification), and there exists an index $1 \leq i \leq n$ such that $\mathcal{FD}_d^s(P_i)$ returns *false* if R is incorrect (i.e. does not match its specification).

2. There exists a run R that is correct (i.e. matches the task specification) and index i such that $\mathcal{FD}^s_{d-1}(P_i)$ returns *false* or there exists an incorrect run R' in which for every $1 \leq i \leq n$ $\mathcal{FD}^s_{d-1}(P_i)$ returns *true*.
3. There exists a run R that is correct (i.e. matches the task specification) and index i such that $\mathcal{FD}^{s-1}_d(P_i)$ returns *false* or there exists an incorrect run R' in which for every $1 \leq i \leq n$ $\mathcal{FD}^{s-1}_d(P_i)$ returns *true*.

Roughly speaking, if only the state of one process is needed, the specification is a *pure local* specification, since failure detectors need not check P_i's neighbors to detect a transient failure; if $d = 1$, the specification is *local*, and if $d = r+1$ the specification is *global*. In addition, one can identify task specification for which only a single view is needed as a *safety* specification, and the related task as a *silent* task. In contrast, *liveness* specification may require unbounded number of consecutive views.

3 Failure Detectors for Silent Tasks

In this section we study failure detectors for silent tasks ([9]) where the communication between the processors is fixed from some point of the run. In our model, registers are used for communication between processors. A system solving a silent task has the property that the contents of the communication registers is not changed. Since silent tasks are 1-history local, in the following, we assume that the locality property only refers to distance locality. Thus, there is no need for the variable that stores the views information to be an array; we use $\mathcal{V}^d_i$ to denote the view of P_i. Similarly, we use $\mathcal{FD}_d$ instead of $\mathcal{FD}^1_d$.

We now list several silent tasks, present their specification, and identify the minimal distance required for their failure detectors. We start with three tasks, maximum independent set, coloring and topology update that are 1-distance local.

3.1 Maximum Independent Set, Coloring and Topology Update

Maximum independent set, task specification: Each processor P_i maintains a local boolean variable $\mathcal{IS}_i$. No two neighbors may have their variable set to *true*. In addition every processor P_i with $\mathcal{IS}_i = \mathit{false}$ has at least one neighbor P_j with $\mathcal{IS}_j = \mathit{true}$.

Lemma 1. *The maximum independent set task is* 1*-distance local.*

Proof. The proof is by presenting a failure detector in the set of $\mathcal{FD}_1$ and proving impossibility result for the existence of a failure detector for this task in $\mathcal{FD}_0$.

By the definition of 1-distance local every processor P_i repeatedly sends the portion of its view $\mathcal{V}^1_i$ that is shared with the portion of the view $\mathcal{V}^1_j$ of its neighbor P_j. In other words each processor P_i repeatedly sends the value of $\mathcal{IS}_i$ to every of its neighbors P_j. Thus, a processor knows the value of $\mathcal{IS}_j$ of every of its neighbors. P_i can verify that: if $\mathcal{IS}_i = true$, then $\forall j \in Neighbors_i$, $\mathcal{IS}_i \neq \mathcal{IS}_j$, and if $\mathcal{IS}_i = false$, then $\exists j \in Neighbors_i$, $\mathcal{IS}_i \neq \mathcal{IS}_j$.

The failure detector at P_i will indicate the occurrence of a fault in case any of the above properties doesn't hold. The above test ensures that the value of all the $\mathcal{IS}$ variables constructs a maximum independent set.

By the definition of 0-distance local no information is communicated. Thus, no fault is detected in a configuration in which the $\mathcal{IS}_i$ variable of every processor P_i holds *true*. □

A similar proof holds for the coloring task that we now present.

Coloring, task specification: Each processor P_i maintains a variable $\mathcal{C}_i$ representing its color. In addition for every two neighboring processors P_i and P_j it holds that $\mathcal{C}_i \neq \mathcal{C}_j$.

Lemma 2. *The coloring task is* 1*-distance local.*

Topology update, task specification: Each processor P_i maintains a local variable $\mathcal{T}_i$, containing the representation of the communication graph, say by using neighboring matrix or the list of the communication graph edges.

Lemma 3. *The topology update task is* 1*-distance local.*

Proof. The proof is by presenting a failure detector in the set of $\mathcal{FD}_1$ and proving the (obvious) impossibility result for the existence of a failure detector for this task in $\mathcal{FD}_0$.

By the definition of 1-distance local every processor P_i repeatedly sends the portion of its view $\mathcal{V}_i^1$ that is shared with the portion of the view $\mathcal{V}_j^1$ of its neighbor P_j. In other words each processor P_i repeatedly sends the value of the output variable $\mathcal{T}_i$ to every of its neighbors P_j. Thus, a processor knows the value of the variables $\mathcal{T}_j$ of every of its neighbors. Therefore, P_i can verify that $\mathcal{T}_i = \mathcal{T}_j$ for every neighboring processor P_j. The failure detector at P_i will notify the occurrence of a fault in case there exists a neighbor for which the above equality does not hold. The above test ensures that if no processor detects a fault then the value of all the $\mathcal{T}$ variables is the same. In addition the failure detector at P_i checks whether the local topology of P_i appears correctly in $\mathcal{T}_i$. [4] This test ensures that the (common identical) value of $\mathcal{T}$ is correct, since every processor identified its local topology in $\mathcal{T}$.

By the definition of 0-distance local no communication take place. Thus, no fault is detected in a configuration in which the $\mathcal{T}_i$ variable of every processor P_i holds the local topology of P_i i.e. P_i and its neighbors without the rest (non empty portion) of the system. □

3.2 Rooted Tree Construction

In this subsection we present the rooted tree construction task that is $\lceil n/4 \rceil$-distance local.

[4] It is assumed that every processor knows the identity of its neighbors.

Rooted tree construction, task specification: Each processor P_i maintains two boolean variables $\mathcal{P}_{ij}$ and $\mathcal{C}_{ij}$ for each of its neighbors, P_j. The value of $\mathcal{P}_{ij}$ is true if P_i chooses P_j to be its parent in the tree. The value $\mathcal{C}_{ij}$ is true if P_i considers P_j to be one of its children in the tree. A single processor in the system, P_r which we call the *root*, has an hardwired false value in every $\mathcal{P}_{rj}$, while the value of the variables of the other processors define a tree rooted at P_r.

Lemma 4. *The rooted tree construction task is $\lceil n/4 \rceil$-distance local.*

Proof. The proof is by presenting a failure detector in the set of $\mathcal{FD}_{\lceil n/4 \rceil}$ and proving impossibility result for the existence of a failure detector for this task that is in $\mathcal{FD}_{\lceil n/4 \rceil - 1}$.

First we describe a failure detector that is in $\mathcal{FD}_{\lceil n/4 \rceil}$. The failure detector of every non root processor, P_i, checks that the value of exactly one $\mathcal{P}_{ij}$ is true. In addition every processor will verify that the value of each variable $\mathcal{C}_{ij}$ is equal to the value of $\mathcal{P}_{ji}$. The failure detector in every non-root processor will check whether a cycle exists or there is an evidence that the tree that is connected to the root does not include all the processors: The view includes all the processors in the above tree (i.e. each processor in the view does not have a child outside the tree) but there exists at least one additional processor in the view. The failure detector of the root will check whether there is an evidence that the subtree connected to it does not include all the processors.

Obviously, no failure is detected when the system encodes a tree rooted at P_r. If a processor which is not the root has no parent, this is detected immediately by the processor itself. Then only the case in which a partial tree is rooted at the root and where the other nodes belong to a cycle has to be considered. Consider a system in which no error is detected. In this system, the root does not belong to a cycle and each processor different from the root has a parent. If the output does not encode a rooted tree, then it is composed of a rooted subtree and of one or more cycles. But every cycle of $\lceil n/2 \rceil$ or less processors is detected by the processors in the cycle. Thus, if no cycle is detected it must be of length greater than $\lceil n/2 \rceil$ processors. Hence, the number of processor connected to the subtree of the root is less than $\lfloor n/2 \rfloor$. The diameter of this subtree is no more than the number of the processors in it. Thus, there exists a processor with a view on the entire subtree and at least one additional neighboring processor that does not belong to the tree, this processor identifies that there exists a processor that is not in the tree.

To prove that the task is not $(\lceil n/4 \rceil - 1) - local$ we assume that in a configuration c in which a chain of $\lceil n/2 \rceil$ processors are connected to the root and the rest of the processors are in a cycle, some processor indicates a fault and we prove that every such possible processor will indicate a fault in a configuration that encodes a tree.

Assume that the communication graph of the system in which a cycle exists includes in addition to the above chain and cycle a (non-tree) edge between the processor at the end of the chain and a processor in the cycle (thus it is possible to have a tree spanning the graph).

Assume that a processor P_i in the cycle identifies a fault in c then there exists at least one processors P_j in the cycle that is not included in $\mathcal{V}_i^d$ (where $d = (\lceil n/4 \rceil - 1)$. There exists a configuration c', that includes an additional edge from P_j to the root, P_j may have chosen this edge to be a tree edge and therefore no cycle exists in c', but P_i will still have the same view $\mathcal{V}_i^d$ in c' and therefore will indicate a fault in c'.

If the processor that indicates a fault is on the chain that is connected to the root, then there is at least one processor, P_j, on the chain and one processor P_k on the cycle that are not included in $\mathcal{V}_i^d$. Thus, P_i will indicate a fault when the system is in another configuration, one in which there are no cycles since P_j and P_k are connected by an edge and P_k chooses P_j as its parent.

□

3.3 $(x, \delta x)-$Graph Partition

$(x, \delta x)-$**Graph Partition, task specification:** Each processor P_i maintains a boolean variable $\mathcal{B}_{ij}$ for every of its attached links (i, j). If the edges with $\mathcal{B} = true$ which are called the *border* edges are disconnected then the graph is partition into connected component of at least x processors and no more than δx processors, where δ is the maximal degree of a node in the communication graph and x is a positive integer smaller than n (See [12] for such a partition). In the full paper we show that a graph partition task such as the $(x, \delta x)-$Graph Partition task is $O(\delta x)-$*distance local.*

3.4 Leader Election and Center Determination

In Sections 3.1, 3.2 and 3.3 we have presented silent task that requires failure detectors with larger and larger views in order to detect inconsistency. In this Section we present a class of tasks that requires that at least one processor will have a view on the entire system. We start with the leader election task.

Leader election, task specification: Each processor P_i maintains a local boolean variable $\mathcal{L}_i$ that is set to *true* if the node is elected and *false* otherwise. There is exactly one node P_l with local variable $\mathcal{L}_l$ set to *true*.

Lemma 5. *The leader election task is $(r+1)-$distance local.*

Proof. The proof is by proving impossibility result for the existence of a failure detector for this task in $\mathcal{FD}_r$ and presenting a failure detector in the set of $\mathcal{FD}_{r+1}$ for the task.

By the definition of $r-$distance local every processor P_i knows the portion of the configuration that includes itself and its neighbors at distance r.

Consider the system S_1 of radius r with $(4r-2)$ processors represented in Figure 3.4.

Consider a configuration of this system in which all variables $\mathcal{L}_i$ are set to false. Every processor knows the portion of the configuration including itself, its neighbors at distance r and all the outgoing channels (but not necessarily the

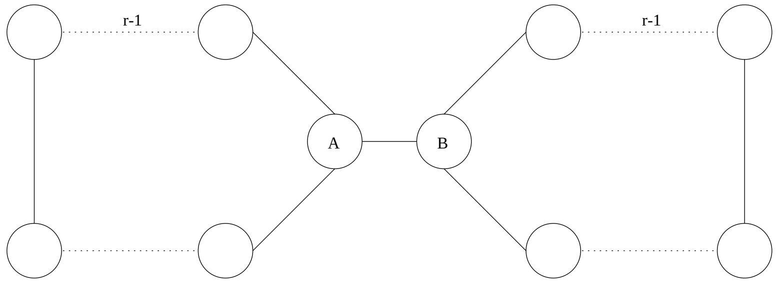

Fig.1. Leader election (1/2)

processors at their extremities). At least one processor must trigger an error and it is easy to see that, because all other processors have a partial view of the system, only A and B can possibly do that. Now consider a second system S_2, of $4r$ processors and with the same radius r, represented in Figure 3.4.

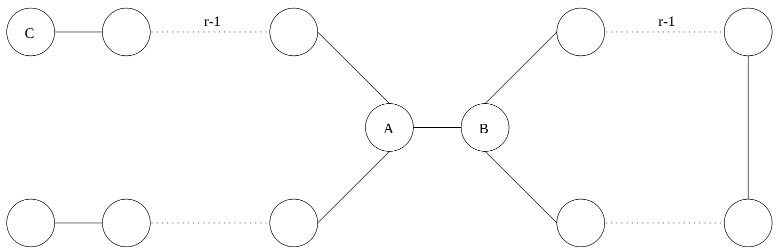

Fig.2. Leader election (2/2)

In S_2, all variables $\mathcal{L}_i$ are set to *false*, but the variable of C, which is set to *true* (so that a unique leader is elected). In both systems, the views at distance r of B are identical. Thus, in both systems, B must decide the same. Because in system S_2, B does not detect a failure, it does not detect a failure either in S_1. By considering a dual system, one can conclude that, in S_1, A cannot detect a failure as well.

The contradiction is complete since, in S_1, no processors triggers an error. Hence, there exists no failure detector in $\mathcal{FD}_r$ for the leader election task.

We now present a failure detector in the set $\mathcal{FD}_{r+1}$ for the leader election task. By the definition of $(r+1)-$distance local every processor P_i knows the part of configuration including itself and its neighbors at distance $(r+1)$. Thus P_i can check whether it knows the entire system, that is whether or not it is a center. If P_i is not a center, it never triggers an error. If P_i is a center, it checks in its $(r+1)$-view whether there is exactly one processor with its variable $\mathcal{L}_i$ set to true. It triggers an error if and only if this test is negative. $\Box$

Remark. Note that each time that the specification of a task is a predicate on the global system configuration, there is a failure detector in $\mathcal{FD}_{r+1}$ for this task. As above, the centers check whether the global predicate is true on their $(r+1)$-views and the other processors do nothing. The next, center determination task has a similar property.

Center determination, task specification: Each processor P_i maintains a local boolean variable $\mathcal{I}_i$ which is true if and only if the processor is a center of the system.

Lemma 6. *The center determination task is* $(r+1)-$*distance local.*

4 Failure Detectors for Non-silent Tasks

In this section we consider the set of non-silent tasks. Unlike the previous section that consider failure detectors for asynchronous (as well as synchronous) systems, we consider in this section failure detector for synchronous systems. We present tasks that are s-history local, with $s > 1$. Here, s defines the size of the array $\mathcal{V}_i^d[1..s]$ that is maintained by each processor P_i. The system is synchronous and each view in $\mathcal{V}_i^d[1..s]$ is correlated to a different time. This array is thereafter referred as the *local history* of processor P_i. Each $\mathcal{V}_i^d$ is a view on every component of the system up to distance d from P_i.

We now list several non-silent tasks, present their specification, and identify the minimal history required for their failure detectors. We start with the trivial bounded privilege task. In fact this task does not require any communication among the processors.

4.1 Bounded Privilege

Bounded privilege, task specification: Each processor P_i maintains a local boolean variable $\mathcal{P}riv_i$. For each processor P_i $\mathcal{P}riv_i$ is set to true exactly once (another variant is at least once) in every c synchronous steps ($c \geq 2$). It is easy to show that the bounded privilege task is 0-distance local, c-history local.

Lemma 7. *The bounded privilege task is* 0*-distance local,* c*-history local.*

Proof. A local history of $c-1$ views, such that in each view the output variable $\mathcal{P}riv_i$ is false does not give an indication on task violation. On the other hand it is clear that a local history of c views is sufficient for fault detection. □

4.2 Bounded Privilege Dining Philosophers

Bounded privilege dining philosophers, task specification: Each processor P_i maintains a local boolean variable $\mathcal{P}riv_i$. For each processor P_i $\mathcal{P}riv_i$ is set to true at least once in every c synchronous steps ($c \geq 2$). In addition for every two neighboring processors P_i and P_j if $\mathcal{P}riv_i = true$ then $\mathcal{P}riv_j = false$. It is easy to show that the bounded privilege dining philosophers task is 1-distance local, c-history local.

Lemma 8. *The bounded privilege dining philosophers task is* 1*-distance local, c-history local.*

4.3 Deterministic Non-interactive Tasks

In a synchronous run of a non-randomized, non-interactive, bounded space algorithm a configuration must be reached more than once, and thus the system must repeat its actions infinitely often, in every infinite run. Therefore, there is a bounded *repetition pattern* that can be identified, where the actions of the processors repeat themselves.

Each processor can have a local history that includes all the views of the repetition pattern in the order they should be repeated. The processor repeatedly send to their neighbors their local history and detect inconsistency if two views that are related to the same time do not agree, or the current value of the output variables are not correct. The distance of the views is a function of the task. Note that in fact, when the distance of views is equal to the diameter of the system, the above failure detectors may serve as an implementation of the task — there is no need for an algorithm.

5 Failure Detectors for Algorithms

Unlike the case of failure detectors for tasks the failure detectors for algorithms (implementation of tasks) may use the entire state of the processors (and not only the output that is defined by the task). For example, in an implementation of the spanning tree construction in which every processor has a variable with the distance from the root the failure detector may use the distance variable to detect inconsistency: if every processor has a distance that is one greater than its parent distance, and the root has no parent then the system is in a consistent state.

A monitoring technique that can be used as a failure detector is presented in [1]. The monitoring technique can detect inconsistency of every on-line or off-line *algorithm*. Since the monitoring technique is universal it is possible to design a more efficient (in terms of memory) failure detectors for specific sets of algorithms.

Hierarchy for failure detectors of algorithms can be defined analogously to the definition of the failure detectors for tasks. The choice of the algorithm that implement a task influences the fitting failure detector. For instance, one may suggest to use a topology update algorithm to implement the above silent tasks. A topology update algorithm provides each processor with the information concerning the entire system, thus every processor may perform *local* computations using this information to elect a leader, to elect an identifier, or to count the nodes in the system. Clearly, the above choice of implementation results in using much more memory than other possible implementations. On the other hand, it is possible to monitor the consistency of this particular implementation by a failure detector in $\mathcal{FD}_1$.

6 Implementing Transient Fault Detectors

In this section, we give a possible implementation for using and maintaining the local variables of the failure detectors, namely the local view (for silent algorithms) and the local history (for non-silent algorithms) variables.

As we have already stated, each processor P_i repeatedly communicates to every of its neighbors, P_j the portion of $\mathcal{V}_i^d$ that is shared with $\mathcal{V}_j^d$. In other words P_i does not communicate to P_j the view on the system components that are of distance $d+1$ from P_j (according to the communication graph portion in $\mathcal{V}_i^d$). When P_i receives the view $\mathcal{V}_j^d$ from its neighbor P_j, P_i checks whether the shared portions of $\mathcal{V}_i^d$ and $\mathcal{V}_j^d$ agree. P_i outputs a failure indication if these portions do not agree.

It is easy to see that the above test ensures that every processor has the right view on the components up to distance d from itself. Assume that the view of P_i is not correct concerning the variable of some processor P_k. Let $P_i, P_{j1}, P_{j2}, \cdots, P_k$ be the processors along a shortest path (of length not greater than d) from P_i to P_k. Let P_{jl} be the first processor in this path for which $\mathcal{V}_{jl}$ and $\mathcal{V}_i$ hold non equal values for a variable of P_k. Note that there must exists such a processor since P_i and P_k holds different values for the variables of P_k. Clearly, P_{jl} and $P_{j(l-1)}$ identify the inconsistency.

The updating policy of the array is the following: in each synchronous step the last view $\mathcal{V}_i^d[s]$ becomes the first view, each other view $\mathcal{V}_i^d[j]$ is copied into $\mathcal{V}_i^d[j+1]$, $1 \leq j < s$.

The last issue in implementing the failure detector is the action taken upon inconsistency detection. Although it is out of the scope of the paper, we mention the reset (e.g., [15], [4]) and the repair operations (e.g, [1], [16]), both should result in a consistent state of the system and the failure detector.

7 Concluding remarks

In this paper, we investigated the resources required for implementing transient fault detectors for tasks and algorithms. We have presented hierarchy of transient fault detectors that detect the occurrence of transient faults in a *single* asynchronous round (in an asynchronous system) or a single synchronous step (in a synchronous system). We propose a measure for identifying the *locality* of tasks and algorithms, which is the amount of information needed to implement a transient failure detector.

References

1. Y. Afek, and S. Dolev, "Local Stabilizer," *Proc. of the 5th Israeli Symposium on Theory of Computing and Systems*, pp. 74-84, 1997. Brief announcement in *Proc. of the 16th Annual ACM Symp. on Principles of Distributed Computing*, pp. 287, 1997.
2. Y. Afek, S. Kutten, and M. Yung, "Memory efficient self-stabilization on general networks", *Proc. 4th Workshop on Distributed Algorithms,* pp. 15-28, 1990.
3. B. Awerbuch, B. Patt-Shamir and G. Varghese, "Self-stabilization by local checking and correction," *Proc. 32nd IEEE Symp. on Foundations of Computer Science*, pp. 268-277, 1991.
4. B. Awerbuch, B. Patt-Shamir, G. Varghese, and S. Dolev, "Self-Stabilization by Local Checking and Global Reset," *Proc. of the 8th International Workshop on Distributed Algorithms*, pp. 226-239, 1994.
5. T. Chandra and S. Toueg, "Unreliable failure detectors for asynchronous systems," *Journal of the ACM*, 43(2):225-267, March 1996.
6. E. W. Dijkstra, "Self-Stabilizing Systems in Spite of Distributed Control", *Communications of the ACM 17,11* (1974), pp. 643-644.
7. S. Dolev, "Self-Stabilizing Routing and Related Protocols," *Journal of Parallel and Distributed Computing*, 42, 122-127, 1997.
8. S. Dolev, T. Herman, "SuperStabilizing protocols for dynamic distributed systems," *Proceedings of the Second Workshop on Self-Stabilizing Systems*, 1995.
9. S. Dolev, G. Gouda, and M. Schneider, "Memory Requirements for Silent Stabilization," *Proc. of the 15th Annual ACM Symp. on Principles of Distributed Computing*, pp. 27-34, 1996.
10. S. Dolev, A. Israeli and S. Moran, "Self Stabilization of Dynamic Systems Assuming Only Read/Write Atomicity", *Distributed Computing*, Vol. 7, pp. 3-16, 1993. *Proc. of the Ninth Annual ACM Symposium on Principles of Distributed Computation*, Montreal, August 1990, pp. 103-117. *Proc. of the first Workshop on Self-Stabilizing Systems*, 1989.
11. S. Dolev, A. Israeli, and S. Moran. "Analyzing expected time by scheduler-luck games", *IEEE Transactions on Software Engineering*, Vol. 21, pp. 429–439, 1995.
12. S. Dolev, E. Kranakis, D. Krizanc and D. Peleg. "Bubbles: Adaptive Routing Scheme for High-Speed Dynamic Networks", *Proc. of the 27th Annual ACM Symposium on the Theory of Computing*, pp. 528-536, 1995. To appear in *SIAM Journal on Computing.*
13. S. Ghosh, A. Gupta, T. Herman and S. V. Pemmaraju, "Fault-Containing Self-Stabilizing Algorithms", *Proc. of the Fifteenth Annual ACM Symposium on Principles of Distributed Computation*, Philadelphia, May 1996, pp. 45-54.
14. J. E Burns, M. G Gouda, and R. E Miller. "Stabilization and pseudo-stabilization", *Distributed Computing*, Vol. 7, pp. 35–42, 1993.
15. S. Katz and K. J. Perry, "Self-stabilizing extensions for message-passing systems", *Distributed Computing*, Vol. 7, pp. 17-26, 1993. *Proc. of the Ninth Annual ACM Symposium on Principles of Distributed Computation*, Montreal, August 1990, pp. 91-101. *Proc. of the first Workshop on Self-Stabilizing Systems*, 1989.
16. S. Kutten and B. P. Shamir, "Time-Adaptive Self Stabilization", *Proc. of the Sixteenth Annual ACM Symposium on Principles of Distributed Computing* pp. 149-158, 1997.
17. C. Lin and J. Simon, "Observing self-stabilization," *Proc. of the Eleventh Annual ACM Symposium on Principles of Distributed Computing* pp. 113-123, 1992.