Distributed Computing manuscript No.
(will be inserted by the editor)

Joffroy Beauquier · Sylvie Delaët · Shlomi Dolev · Sébastien Tixeuil

Transient Fault Detectors

Received: date / Accepted: date

Abstract We present fault detectors for transient faults, (*i.e.* corruptions of the memory of the processors, but not of the code of the processors). We distinguish fault detectors for *tasks* (*i.e.* the problem to solve) from failure detectors for *implementations* (*i.e.* the algorithm that solves the problem).

The aim of our fault detectors is to detect a memory corruption as soon as possible. We study the amount of memory needed by the fault detectors for some specific tasks, and give bounds for each task. The amount of memory is related to the size and the number of views that a processor has to maintain to ensure a quick detection. This work may give the implementation designer hints concerning the techniques and resources that are required for implementing a task.

Keywords Distributed Systems · Transient Faults · Fault Detectors · Self-stabilization

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 subject to transient faults. This is the reason why the processors in self-stabilizing systems must continue the execution of the algorithm forever and never know for sure that the system is stabilized.

An extended abstract of this paper was presented in the *12th International Symposium on DIStributed Computing (DISC'98).* Shlomi Dolev is partly supported by the Israeli ministry of science and arts grant #6756195. Part of this research was done while Shlomi Dolev was visiting the Laboratoire de Recherche en Informatique (LRI), Univ. Paris Sud.

Univ. Paris Sud, LRI-CNRS 8623, INRIA Grand Large, FR91405 Orsay, France, E-mail: jb@lri.fr · Univ. Paris Sud, LRI-CNRS 8623, FR91405 Orsay, France, E-mail: delaet@lri.fr · Department of Mathematics and Computer Science, Ben-Gurion University, Beer-Sheva, 84105, Israel, E-mail: dolec@cs.bgu.ac.il · Univ. Paris Sud, LRI-CNRS 8623, INRIA Grand Large, FR91405 Orsay, France, E-mail: tixeuil@lri.fr

In this paper we propose a tool for identifying the inconsistency of a system, namely a transient fault detector. Identification of a transient fault can be coupled with a self-stabilizing reset procedure to obtain a self-stabilizing algorithm from an existing non self-stabilizing algorithm (See *e.g.*, [9, 11]). 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 fault detector is unreliable since it can detect inconsistent state as a result of a transient fault that corrupts the state of the fault detector itself. The only guarantees we have is that when the system is not in a consistent state a fault is detected, and when both the system and the fault detector are in a consistent state no fault is detected.

Our focus in this paper is in the implementation of fault 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.*, [13, 19, 1]).

In this paper we present fault detectors that detect transient faults, *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 fault detectors for tasks (and not for a specific implementation) gives the implementation designers the flexibility of changing the implementation without modifying the fault 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 fault detectors for a specific implementation — specific algorithm. Obviously, one may use a fault detector for the task of the algorithm without considering the data structures 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 fault detector for a specific implementation and not a fault detector for the task.

Related work: The term "failure detector" was introduced in a different context in [6], 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 [20] where a snapshot of the system is repeatedly collected. Fault detectors, called observers, that are initialized correctly and are not subject to state corruption are used in [22]. Monitoring consistency *locally* for a restricted set of *algorithms* has been suggested in *e.g.*, [3,2,17,8,18,7]. 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 fault detectors for *tasks* and *algorithms* that is based on the amount of information used by the fault detector. It is interesting that different distributed *tasks* require different amounts of memory for detecting transient faults. We note that a transient fault detector that is designed for a particular *algorithm* may use the variables of the algorithm for fault detection.

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

2 The System

Distributed system Our system settings are similar to the one presented in [14]. 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.

Let u and v be two nodes of the communication graph, $Dist(u,v)$ denotes the number of edges on a shortest path from u to v (if such a path exists). Let $MaxDist(u)$ be the maximal value of $Dist(u,v)$ over all nodes v in the system. A node u is a *center* of G if there is no node v such that $MaxDist(u) > MaxDist(v)$. The *radius*, r, of G is the value of $MaxDist(u)$ for a center node u of G. The *diameter*, D, of G is the maximal value of $MaxDist(v)$ over all nodes v in the system. Let i be an integer, $Ball(u,i)$ denotes the set B of nodes $b \in B$ such that $Dist(u,b) \leq i$.

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 and its communication registers *i.e.*, the registers to which it writes. Let S_i be the set of states of the processor P_i. A *configuration*, $c \in (S_1 \times S_2 \times \cdots S_n)$ is a vector of the states of all the processors.

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 *cycle complexity* (See [15,13,10]) to measure time in asynchronous system. The first cycle of a run is the minimal prefix of the run in which each processor reads the registers of all its neighbors and writes to its registers. The second cycle is the first cycle of the rest of the run, and so on. In an asynchronous run, time i relates to the i^{th} cycle.

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. In a synchronous run, time i relates to the i^{th} configuration.

Specifications and fault 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 in each configuration. The *specification* P of a task T for a given system $\mathscr{S}$ is a (possibly infinite) set of abstract runs of $\mathscr{S}$. For example, the mutual exclusion 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.

$\mathscr{S}$ is *self-stabilizing* (in fact *pseudo self-stabilizing* [5]) with relation to P if and only if each of its runs has a suffix in P.

The goal of a *transient fault detector* is to check if a particular run of the system $\mathscr{S}$ matches the specification P. More precisely, a fault detector is assimilated to a boolean variable that obtains the value *true* if and only if the specification is satisfied. Our fault detectors use the concept of *views* and *histories*, that are defined for a specification P.

Definition 1 (View) The view $\mathscr{V}_i^d$ at distance d from a processor P_i contains: *(i)* the subgraph of the system communication graph G containing the nodes in $Ball(i,d)$, and *(ii)* the set of output variables that are associated with those nodes for a specification P.

Definition 2 (History) The history $\mathscr{V}_i^d[1\dots s]$ at distance d from a processor P_i is a sequence of s consecutive views at distance d from P_i.

The history element $\mathscr{V}_i^d[j]$, for a given index j, is the view at distance d from P_i at time j. View $\mathscr{V}_i^d[1]$ is associated to present time, while view $\mathscr{V}_i^d[s]$ is associated to $s-1$ time units in the past.

The fault detectors that we consider in this paper are *distributed* in the sense that they can be invoked by each of the system processors and give a different response to each of them. However, we assume that the response of a fault detector to a processor is uniquely determined by the history of the invoking processor. More precisely, the response of the oracle is *true* if and only if a predicate on the history of the calling process (induced by the specification of the task) is also true.

Definition 3 (Fault detector at a processor) The fault detector $\mathscr{F}\mathscr{D}_d^s(P_i)$ at processor P_i is an oracle that can be invoked by P_i and whose binary response is uniquely determined by the local history $\mathscr{V}_i^d[1\dots s]$ of P_i.

The result of $\mathscr{F}\mathscr{D}_d^s(P_i)$ can be used in any configuration c_j of a system run. The result of the oracle $\mathscr{F}\mathscr{D}_d^s(P_i)$ in configuration c_j is denoted by $\mathscr{F}\mathscr{D}_d^s(P_i,c_j)$.

Definition 4 (Fault detector in a configuration) The distributed result $\mathscr{F}\mathscr{D}_d^s(P_i,c_j)$ of a fault detector in a configuration c_j is the conjunction of the $\mathscr{F}\mathscr{D}_d^s(P_i,c_j)$ results for each processor P_i in the system.

Intuitively, if in a given system configuration, the task specification is satisfied, then every oracle invocation at every processor must return *true*. In the case of a system configuration where the task specification is not satisfied, at least one of the fault detectors at a processor must return *false* to the invoking processor.

Task classification A task is $(d,s)-$local if and only if there exists a fault detector that uses s consecutive views at distance d and is correct for this task, while there exist no correct fault detector for this task that use at most $s-1$ consecutive views or views at distance at most $d-1$. More formally, the following conditions are verified:

1. For each configuration c of any system $\mathscr{S}$ and for all $1 \le i \le n$ $\mathscr{F}\mathscr{D}_d^s(P_i,c)$ returns $true$ if c is correct (*i.e.* matches the task specification), and there exists an index $1 \le i \le n$ such that $\mathscr{F}\mathscr{D}_d^s(P_i,c)$ returns $false$ if c is incorrect (*i.e.* does not match its specification).
2. For any fault detector in $\mathscr{F}\mathscr{D}_{d-1}^s(P_i,c)$, there exists a configuration c of a particular system $\mathscr{S}$ that is correct (*i.e.* matches the task specification) and index k such that $\mathscr{F}\mathscr{D}_{d-1}^s(P_k,c)$ returns $false$ or there exists an incorrect configuration c' in which for every $1 \le k \le n$ $\mathscr{F}\mathscr{D}_{d-1}^s(P_k,c')$ returns $true$.
3. For any fault detector in $\mathscr{F}\mathscr{D}_d^{s-1}(P_i,c)$, there exists a configuration c of a particular system $\mathscr{S}$ that is correct (*i.e.* matches the task specification) and index k such that $\mathscr{F}\mathscr{D}_d^{s-1}(P_k,c)$ returns $false$ or there exists an incorrect configuration c' in which for every $1 \le k \le n$ $\mathscr{F}\mathscr{D}_d^{s-1}(P_k,c')$ returns $true$.

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

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

The two locality criteria that we consider for tasks refer to the ease that transient faults will be trigerred. The smaller the distance or time locality, the less information oracles use to give a correct response.

Roughly speaking, if the result of the oracle at a processor P_i uses a history where only P_i's output variables are present, the predicate that is evaluated by the oracle is *pure local*: it is independent from the neighborhood of P_i. In the particular case when the distance d of the history of each P_i equals 1, the oracle's predicate is *local*. If $d=r+1$, the predicate is *global*.

3 Fault Detectors for Silent Tasks

In this section we study fault detectors for silent tasks ([12]) where the output variables remain fixed from some point of the run. Since silent tasks are $0-$history local, in the following, we assume that the locality property only refers to distance locality. Therefore, we use $\mathscr{V}_i^d$ instead of $\mathscr{V}_i^d[1]$ to denote the history of P_i, and $\mathscr{F}\mathscr{D}_d$ instead of $\mathscr{F}\mathscr{D}_d^1$.

We now list several silent tasks, present their specification, and identify the minimal distance required for their fault detectors.

3.1 Leader Election, Center Determination and Number of Nodes

In this section we present a class of tasks that requires that at least one processor has a view of the entire system. In Section 2, we assumed that the $\mathscr{V}_i^d$ views contain the communication graph description up to distance d from P_i. There are two possible basic assumptions about the communication graph information that is stored in the $\mathscr{V}_i^d$ views:

Assumption 1 (Links included) *The edge identifiers of the network are stored in the $\mathscr{V}_i^d$ views. In other words $\mathscr{V}_i^d$ contains the information about the identity of the processors in distance $d+1$ that are connected to each processor in distance d from P_i.*

Assumption 2 (Links not included) *The edge identifiers of the network are* not *stored in the $\mathscr{V}_i^d$ views. In other words $\mathscr{V}_i^d$ does not contain the information about the identity of the processors in distance $d+1$.*

Let us see how these assumptions influence the information that can be deduced from the view of a processor (*not* knowing the actual radius of the communication graph):

1. If Assumption 1 holds, then the view $\mathscr{V}_i^r$ (where r is the radius of the communication graph of the system) of a node of the communication graph is sufficient to conclude that P_i *is* (or not) a center. Indeed, if each edge in $\mathscr{V}_i^r$ is connected to exactly two nodes, then P_i is a center, otherwise, it is not.
2. If Assumption 2 holds, then the view $\mathscr{V}_i^r$ is insufficient to conclude that P_i is (or not) a center, because it is impossible to distinguish the following two cases:
   (a) the radius of the network is r and $\mathscr{V}_i^r$ contains a complete knowledge of the network,
   (b) the radius of the network is $r+1$ and $\mathscr{V}_i^r$ misses information about at least one node.

   Note that if the view of P_i is $\mathscr{V}_i^{r+1}$, where r is the radius of the system, it is possible to conclude that P_i is a center by checking whether there are processors in distance $r+1$ from itself.

Now we are ready to consider the first task which is the leader election task.

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

Lemma 1 *If Assumption 1 holds, the leader election task is r distance local.*

Proof We first present an impossibility result for the existence of a fault detector for this task in $\mathscr{F}\mathscr{D}_{r-1}$ and then present a fault detector in the set of $\mathscr{F}\mathscr{D}_r$ for the task.

Let us consider a system $\mathscr{S}$ such that its communication graph $G=(V,E)$ verifies: $\exists x \in V$, $\exists y \in V$, $Dist(x,y)=2r-1$, where r is the radius of G. For example, a chain graph of $2r$ nodes verifies this property. Then, for any node v in this graph, either *(i)* $Ball(v,r-1)$ contains x but not y, or *(ii)* $Ball(v,r-1)$ does not contain x. Indeed, it is impossible that $Ball(v,r-1)$ contains both x and y, since $Dist(x,y)=2r-1$ by hypothesis, which is incompatible with the following derivation:

$$\begin{array}{rcl} Dist(x,y) & \leq & Dist(x,u)+Dist(y,u) \\ & \leq & r-1+r-1 \\ & \leq & 2r-2 \\ & < & 2r-1 \end{array}$$

Let us consider the following three configurations:
c_1 in which x is elected and all other nodes are not,
c_2 in which y is elected and all other nodes are not, and
c_3 in which no node is elected.

Suppose there exists a $\mathscr{F}\mathscr{D}_{r-1}$ for the leader election task. The result of $\mathscr{F}\mathscr{D}_{r-1}$ in c_1 and c_2, must be $true$ at every node, since the configuration satisfies the leader election specification; on the other hand, in c_3, at least one failure detector at a node node must respond $false$.

We now consider every node v in c_3. If $Ball(v,r-1)$ contains x but not y (case *(i)*), then $\mathscr{V}_v^{r-1}$ is the same in configurations c_1 and c_3, thus $\mathscr{F}\mathscr{D}_v^{r-1}(v,c_3)$ returns *true*. If $Ball(v,r-1)$ does not contain x (case *(i)*), then $\mathscr{V}_v^{r-1}$ is the same in configurations c_1 and c_3, thus $\mathscr{F}\mathscr{D}_v^{r-1}(v,c_3)$ returns *true*. Hence, $\mathscr{F}\mathscr{D}_{r-1}$ responds $true$ in c_3 in which there is no leader.

This completes the proof that the leader election task is not $r-1$ distance local.

We now present a fault detector in the set $\mathscr{F}\mathscr{D}_r$ for the leader election task. By the definition of $r-$distance local, every view $\mathscr{V}_i^r$ of processor P_i contains the part of configuration including P_i and its neighbors at distance r. According to Assumption 1, $\mathscr{F}\mathscr{D}_r(P_i)$ can check whether it knows the entire system, that is whether or not P_i is a center of the communication graph. If P_i is not a center, $\mathscr{F}\mathscr{D}_r(P_i)$ returns $true$ (does not detect a fault). If P_i is a center, $\mathscr{F}\mathscr{D}_r(P_i)$ checks in its $r-$view whether there is exactly one processor P_l with $\mathscr{L}_l = true$. $\mathscr{F}\mathscr{D}_r(P_i)$ detects a fault if and only if the above condition is false.

Next we present a similar proof for the case in which the view does not include the identity of the processors to which a link is connected, unless the link belongs to a path of length $d-1$ from a processor. Namely, using Assumption 2.

Lemma 2 *If Assumption 2 holds, the leader election task is $(r+1)-$distance local.*

Proof We first present an impossibility result for the existence of a fault detector for this task in $\mathscr{FD}_r$ and then present a fault detector in the set of $\mathscr{FD}_{r+1}$ for the task. By the definition of $r-$distance local, every view $\mathscr{V}_i^r$ of processor P_i contains the portion of the configuration that includes P_i and its neighbors at distance r.

Consider the system $\mathscr{S}_1$ of radius r with $(4r-2)$ processors represented in Figure 1. Consider a configuration of this system in which all variables $\mathscr{L}_i$ are set to false. At least one fault detector at a processor must detect a fault, it is easy to see that, because all other processors have a partial view of the system, only $\mathscr{FD}_r(A)$ and $\mathscr{FD}_r(B)$ can possibly do that.

Now consider a second system $\mathscr{S}_2$, of $4r$ processors (with the same radius value r), represented in Figure 1. In $\mathscr{S}_2$, all variables $\mathscr{L}_i$ 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, $\mathscr{FD}_r(B)$ must decide the same. Because in system $\mathscr{S}_2$, $\mathscr{FD}_r(B)$ does not detect a fault, it does not detect a fault either in $\mathscr{S}_1$. By considering a dual system, one can conclude that, in $\mathscr{S}_1$, $\mathscr{FD}_r(A)$ cannot detect a fault as well.

The contradiction is complete since, in $\mathscr{S}_1$, no processors detect a fault. Hence, there exists no fault detector in $\mathscr{FD}_r$ for the leader election task.

We now present a fault detector in the set $\mathscr{FD}_{r+1}$ for the leader election task. By the definition of $(r+1)-$distance local every view $\mathscr{V}_i^{r+1}$ of processor P_i contains the part of configuration including P_i and its neighbors at distance $(r+1)$. Thus $\mathscr{FD}_{r+1}(P_i)$ can check whether it knows the entire system, that is whether or not P_i is a center. If P_i is not a center, $\mathscr{FD}_{r+1}(P_i)$ never detects a fault. If P_i is a center, $\mathscr{FD}_{r+1}(P_i)$ checks in its $(r+1)-$view whether there is exactly one processor P_l with its variable $\mathscr{L}_l$ set to true. $\mathscr{FD}_{r+1}(P_i)$ detects a fault if and only if the above condition is false.

Observation 1 *For every task whose specification is a predicate on the global system configuration, there is a fault detector in $\mathscr{FD}_{r+1}$(resp. $\mathscr{FD}_r$) for this task if Assumption 2 (resp. Assumption 1) holds. Namely, the fault detector that uses the centers to check whether the global predicate is true in their $(r+1)-$views (resp. $r-$views) while the other processors return* true.

In the rest of the paper, we will assume that Assumption 2 holds.

Next, we consider the center determination task.

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

Lemma 3 *The center determination task is $r-$distance local.*

Proof The proof that there is no fault detector in $\mathscr{FD}_{r-1}$ for the center determination task is by presenting two systems $\mathscr{S}_1$ and $\mathscr{S}_2$ depicted in Figure 2. The configuration that we consider for both systems is that where only the $\mathscr{I}$ variable of C is $true$.

System $\mathscr{S}_1$ is in a correct configuration, where centers are correctly identified, so each transient fault detector responds $true$ in this configuration. System $\mathscr{S}_2$ is in an incorrect configuration, because E is incorrectly identified as a non-center.

Now every view of every node in the N branch is the same in systems $\mathscr{S}_1$ and $\mathscr{S}_2$, so each transient fault detector at these nodes responds $true$. Then every node in the W branch in $\mathscr{S}_2$ has the same view as its counterpart node in the N branch, so each transient fault detector at these nodes responds $true$. Finally every node in the E branch in $\mathscr{S}_2$ has the same view in systems $\mathscr{S}_1$ and $\mathscr{S}_2$, so each transient fault detector at these nodes responds $true$. Consequently, no fault is detected in $\mathscr{S}_2$.

Now we construct a fault detector in $\mathscr{FD}_r$ for the center determination task as follows:

1. if $\mathscr{I}_i = true$ (P_i claims it is a center) then the fault detector at P_i responds $false$ iff:
   (a) $\mathscr{V}_i^r$ contains a processor P_j that is a center in $\mathscr{V}_i^r$ and such that $\mathscr{I}_j = false$,
   (b) $\mathscr{V}_i^r$ contains a processor P_j that is not a center in $\mathscr{V}_i^r$ and such that $\mathscr{I}_j = true$.
2. if $\mathscr{I}_i = false$ (P_i claims it is not a center) then the fault detector at P_i responds $false$ iff:
   (a) $\mathscr{V}_i^r$ contains only processors P_k such that $\mathscr{I}_k = false$,
   (b) $\mathscr{V}_i^r$ contains two processors P_j and P_k such that $\mathscr{I}_j = true$ and $Dist(j,k) \geq r+1$ in $\mathscr{V}_i^r$.

If the system is correct, any center P_i has its $\mathscr{I}_i$ variable set to $true$ and any non-center P_j has its $\mathscr{I}_j$ variable set to $false$. Then none of the centers can detect an inconsistency in its view (that covers the whole network). Any non center is at distance at most r from a center (by definition of a center) in any view, so no fault detector returns $false$.

If the system is not correct, this means that one of the following two cases occurred: there exists a non center P_j such that $\mathscr{I}_j = true$, or there exists a center P_k such that $\mathscr{I}_k = false$. This leads to three kinds of configurations:

1. There exists a center P_k such that $\mathscr{I}_k = true$: then P_k can detect a fault because its view includes the whole network.

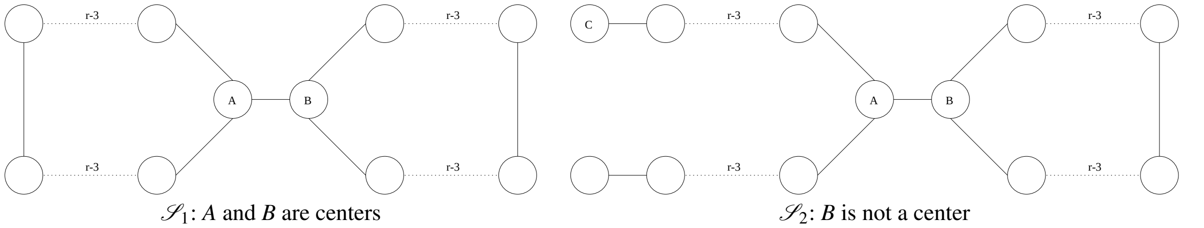


Fig. 1 Leader Election

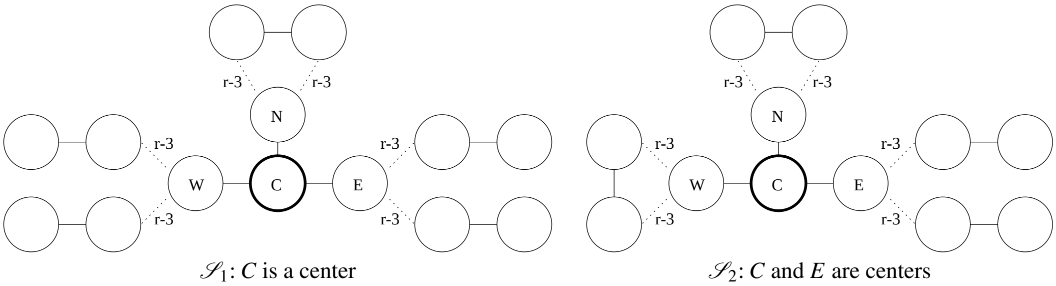


Fig. 2 Center Determination

2. There exists no node P_i such that $\mathscr{I}_i = true$, then all fault detectors respond $false$.
3. For any center P_k, $\mathscr{I}_k = false$, and there exists a non center P_j such that $\mathscr{I}_j = true$. Then P_k has the whole network in its view. By definition of a non center, there exists a processor P_i in the network such that $Dist(j,i) \geq r+1$. So P_k has in its view two nodes at distance more than r and one of them claims it is a center, thus it detects a fault.

It may be surprising that the center determination problem is r-local while the leader election problem is $(r+1)$-local under the same hypothesis, while the transient fault detector that we presented for leader election checks if the invoking processor is a center of the graph to build its decision.

The reason is the following: to check if the leader election problem is solved using only leader election output variables, a view at distance $r+1$ is needed; to check if the leader election problem is solved using both the leader election and center output variables, a view at distance r is sufficient. However, the resulting failure detector is not a failure detector for the leader election task, but a failure detector for the center determination *and* leader election task.

Number of Nodes Each processor P_i maintains a variable $Number_i$ containing the number of nodes in the system.

Lemma 4 *The number of nodes task is $(r+1)-$distance local.*

Proof According to Observation 1, it suffices to prove that there is no fault detector in $\mathscr{FD}_r$ for the number of nodes task. Suppose the contrary holds and consider the system $\mathscr{S}_1$ of $(4r-2)$ processors and radius r, placed in a configuration in which each processor P_i has its variable $Number_i$ set to $4r$, which is the number of processors in the system $\mathscr{S}_2$, see Figure 1 on page 6. Each processor P_i different from A and B (not being a center) has less than $(4r-4)$ processors in its view at distance r.

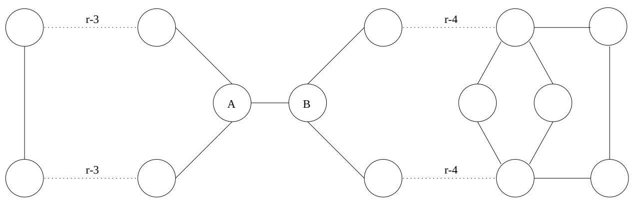


Fig. 3 Number of Nodes

It is easy to build a system $\mathscr{S}_1(i)$ with $4r$ processors for each such P_i, with a processor having the same $r-$view as P_i for any system configuration. For example, see Figure 3 for nodes at the left of A in $\mathscr{S}_1$. Thus $\mathscr{FD}_r(P_i)$ cannot detect a fault. Hence only $\mathscr{FD}_r(A)$ and $\mathscr{FD}_r(B)$ (the centers) can possibly detect a fault in system $\mathscr{S}_1$. But A has the same view at distance r in $\mathscr{S}_1$ and $\mathscr{S}_2$. Thus $\mathscr{FD}_r(A)$ does not detect a fault neither in $\mathscr{S}_1$ nor in $\mathscr{S}_2$. For similar reasons, $\mathscr{FD}_r(B)$ does not detect a fault either, a contradiction.

3.2 Rooted Tree, Hamiltonian Circuit, Eulerian Circuit and $x-$Tree Partition

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

Rooted tree construction, task specification Each processor P_i maintains a variable $\mathscr{P}_i$ with a pointer to one of its neighbors, chosen to be its parent in the tree. A single processor in the system, P_r which we call the *root*, has a hardwired *nil* value in $\mathscr{P}_r$, while the value of the variables of the other processors define a tree rooted at P_r.

Lemma 5 *The rooted tree construction task is $\lceil \frac{n}{4} \rceil-$distance local.*

Proof The proof is by presenting a fault detector in the set of $\mathscr{F}\mathscr{D}_{\lceil \frac{n}{4} \rceil}$ and proving impossibility result for the existence of a fault detector for this task that is in $\mathscr{F}\mathscr{D}_{\lceil \frac{n}{4} \rceil-1}$. The fault 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 some processor. The fault detector of the root will check whether the subtree connected to it does not include some processor. Obviously, no fault is detected when the system encodes a tree rooted at P_r. If a processor P_i which is not the root has no parent, this is detected immediately using $\mathscr{V}_i^1$. 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.

A fault detector that uses views of radius $\lceil \frac{n}{4} \rceil$ can detect cycles that are up to $2 \times \lceil \frac{n}{4} \rceil$ nodes long. Consequently, a cycle with $2 \times \lceil \frac{n}{4} \rceil + 1$ processors cannot be detected. Consider a graph that has such a cycle and a subtree rooted at the root node. This subtree contains at most $n - 2 \times \lceil \frac{n}{4} \rceil - 1$ nodes, so its diameter is at most $n - 2 \times \lceil \frac{n}{4} \rceil - 2$. Let B be the set of nodes that are in the subtree connected to the root or are direct neighbors of a node in this subtree. The diameter of the part of the communication graph that connects the nodes in B (the graph in which only the edges connecting nodes of B appear) is at most $n - 2 \times \lceil \frac{n}{4} \rceil$. If a center P_c of the B set has a view at distance $\lceil \frac{n-2\times\lceil \frac{n}{4} \rceil}{2} \rceil$, it can view all nodes in B. Since $\lceil \frac{n-2\times\lceil \frac{n}{4} \rceil}{2} \rceil \leq \lceil \frac{n}{4} \rceil$, all the nodes of B are included in the view of P_c.

Now, if *all* nodes that are direct neighbors of a processor in the subtree (but do not belong to the subtree) have not chosen their parent among the subtree nodes, then the transient fault detector at the center processor can detect a fault appropriately, since it detected processors that are not connected to the root.

To prove that the task is not $(\lceil \frac{n}{4} \rceil - 1) - \textit{local}$ we consider a system configuration c in which a chain of $\lceil n/2 \rceil$ processors are connected to the root and the rest of the processors form a cycle. In such a configuration, at least one transient fault detector at a processor must detect a fault, we prove that every such possible detector will also detect 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 a processor in the chain and a processor in the cycle (thus it is possible to have a tree spanning the graph).

Assume that a fault detector $\mathscr{F}\mathscr{D}_d(P_i)$ at processor P_i in the cycle identifies a fault in c (*i.e.* $\mathscr{F}\mathscr{D}_d(P_i,c)$ returns *false*). Then there exists at least one processor P_j in the cycle that is not included in $\mathscr{V}_i^d$ (where $d = (\lceil \frac{n}{4} \rceil - 1)$). 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 $\mathscr{V}_i^d$ in c' and therefore $\mathscr{F}\mathscr{D}_d(P_i)$ will detect a fault in c' (*i.e.* $\mathscr{F}\mathscr{D}_d(P_i,c)$ returns *false*).

If the fault detector that indicates a fault is at a processor P_i 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 $\mathscr{V}_i^d$. Thus, $\mathscr{F}\mathscr{D}_d(P_i)$ 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_j chooses P_k as its parent.

Hamiltonian circuit construction (in a Hamiltonian system) Each processor P_i has distinguished one incoming ($in(i)$) and one outgoing ($out(i)$) edge, that globally induce a Hamiltonian circuit.

Lemma 6 *The Hamiltonian circuit construction task is $\lceil n/4 \rceil$ $-$ distance local (n is the number of nodes).*

Proof Note first that a fault detector at a processor that does not have distinguished exactly one incoming and one outgoing edge can detect a fault. If the structure globally induced by the variables $in(i)$ and $out(i)$ is not a Hamiltonian circuit, it has necessarily several circuits covering the system (each processor is in one circuit). But one of the circuits has no more than $n/2$ processors and each processor in this circuit has the entire circuit in its view at distance $\lceil n/4 \rceil$. Thus a fault can be detected by any fault detector at these processors.

Now, for proving that the task has no fault detector in $\mathscr{F}\mathscr{D}_{\lceil n/4 \rceil - 1}$, consider the two systems $\mathscr{S}_1$ and $\mathscr{S}_2$ presented in Figure 4.

In both $\mathscr{S}_1$ and $\mathscr{S}_2$, processor A has the same $\mathscr{V}^{\lceil n/4 \rceil - 1}$, thus $\mathscr{F}\mathscr{D}_{\lceil n/4 \rceil - 1}(A)$ cannot detect a fault in $\mathscr{S}_1$. Since the system is symmetric, no fault detector can detect a fault in $\mathscr{S}_1$, where no Hamiltonian circuit is built.

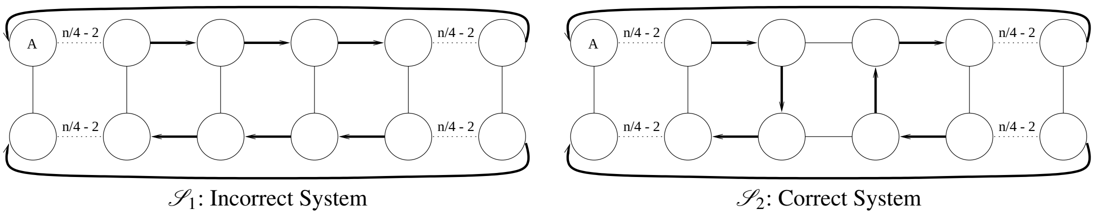


Fig. 4 Hamiltonian Circuit

Eulerian circuit construction (in an Eulerian system) Each processor P_i has $\delta_i/2$ variables (δ_i being the degree of node i in which P_i resides). Each variable l_j contains a pair of indices of edges attached to P_i, such that these edges appear one immediately after the other in the Eulerian circuit.

Lemma 7 *The Eulerian circuit construction task is $\lceil n/4 \rceil - $distance local (n is the number of nodes).*

Proof Note first that a fault detector at a processor P_i having an edge appearing in two different l_j can detect a fault.

Then to prove that there is exactly one circuit, we follow the same proof as in the Hamiltonian circuit construction proof, considering the two systems in Figures 5 and 6.

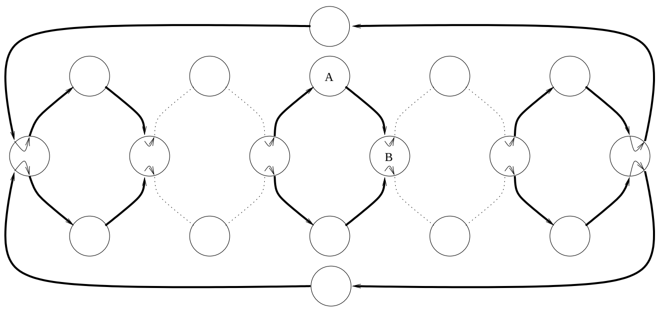


Fig. 5 No Eulerian Circuit

Processors A and B have the same $\mathscr{V}^{\lceil n/4 \rceil - 1}$ in both systems, thus neither of their fault detectors can detect a fault in the first one.

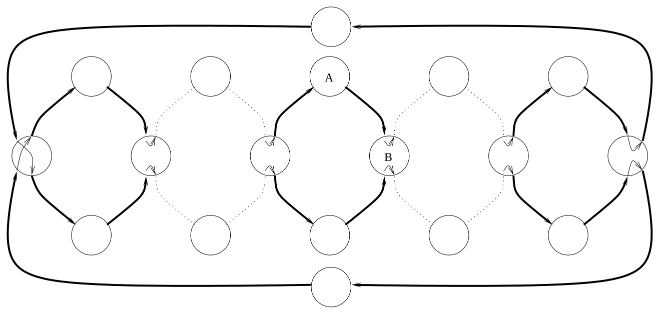


Fig. 6 Eulerian Circuit

Since the system is symmetric, no transient fault detector can detect a fault in the first system, where no Eulerian circuit is built.

Next we show that there is a class of tasks that require fault detector of radius i (where i is not related to n) for every integer i.

$x-$Tree partition, task specification Each processor P_i maintains a boolean variable $\mathscr{B}_{ij}$ for every of its attached links (i, j). For any two neighbor nodes i and j, $\mathscr{B}_{ij} = \mathscr{B}_{ji}$. The edges with $\mathscr{B}_{ij} = true$ are called the *border* edges. If the border edges are which are disconnected then the Tree is partitioned into connected components of diameter of no more than x, where x is a positive integer smaller than n (See [16]).

Lemma 8 *The $x-$Tree partition is $\lfloor \frac{x}{2} \rfloor$-distance local.*

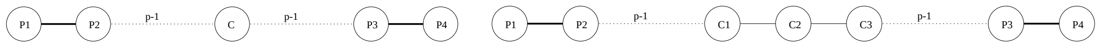
Proof The proof is by proving there exists no fault detector for the x-Tree partition task in the $\mathscr{F}\mathscr{D}_{\lfloor \frac{x}{2} \rfloor - 1}$ set and by showing the existence of a fault detector in the $\mathscr{F}\mathscr{D}_{\lfloor \frac{x}{2} \rfloor}$ set.

The impossibility result is by considering the two systems $\mathscr{S}_1$ (which is correct) and $\mathscr{S}_2$ (which is incorrect) presented in Figure 7 and by showing that if a fault detector responds *true* in the first system, it also responds *true* in the second system. In the two systems, thicker edges denote borders.

We consider $p = \lfloor \frac{x}{2} \rfloor$ (so that $x = 2 \times p$ if x is even and $x = 2 \times p + 1$ if x is odd). Suppose that in $\mathscr{S}_1$, the processor C has a view at distance $p - 1$. Then $\mathscr{F}\mathscr{D}_{p-1}(C)$ does not see processors P_2 and P_3. Since the first system is correct, no transient fault detector detects a fault. Now in $\mathscr{S}_2$, processors C_1, C_2 and C_3 have the same view as processor C in $\mathscr{S}_1$. All other nodes in $\mathscr{S}_2$ have the same view as in $\mathscr{S}_1$, so neither of the fault detectors at these processors can detect a fault. As a result, no detector detects a fault in $\mathscr{S}_2$.

Now we construct a transient fault detector for the $x-$Tree partition task that is in $\mathscr{F}\mathscr{D}_{\lfloor \frac{x}{2} \rfloor}$ at processor P_i. This fault detector responds *true* except in the following two cases:

1. $\mathscr{V}_i^p$ contains j and k such that $\mathscr{B}_{jk} \neq \mathscr{B}_{kj}$.
2. $\mathscr{V}_i^p$ contains x nodes $i_1, i_2, \ldots, i_x$ such that for any j in $[1..x]$, $\mathscr{B}_{i_j i_{j+1}} = false$.



$\mathscr{S}_1$: Correct System $\mathscr{S}_2$: Incorrect System

Fig. 7 *x*-Tree partition

If the configuration is correct, then for any two nodes j and k, we have $\mathscr{B}_{jk} = \mathscr{B}_{kj}$, so rule 1 does not apply. Also, since any chain such that any process P_i in the chain has one of its $\mathscr{B}_{i*}$ set to $false$ is of length less than the diameter of a connected component, it is also less than x, thus rule 2 does not apply either. As a consequence, in a correct configuration, the fault detector responds $true$.

If the configuration is not correct, it means that either two neighboring nodes j and k do not agree on a common border edge, or that there exists a connected component of diameter strictly greater that x. The first case is trivially solved by rule 1. The second case can be rewritten as there exists a chain of $x+1$ processors $P_{i_1}, P_{i_2}, \ldots, P_{i_{x+1}}$ such that for any j in $[1..x]$, $\mathscr{B}_{i_j i_{j+1}} = false$. By definition, a center node c of this chain has in its view $\mathscr{V}_c^p$ all processors $P_{i_1}, P_{i_2}, \ldots, P_{i_{x+1}}$, and by rule 2, $\mathscr{F}\mathscr{D}_p(c)$ can detect a fault.

3.3 Maximum Independent Set, Coloring and Topology Update

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

Lemma 9 *The maximum independent set task is* $1-$*distance local.*

Proof The proof is by presenting a fault detector in the set of $\mathscr{F}\mathscr{D}_1$ and proving impossibility result for the existence of a fault detector for this task in $\mathscr{F}\mathscr{D}_0$.

By the definition of $1-$distance local $\mathscr{F}\mathscr{D}_1(P_i)$ can verify that: if $\mathscr{I}\mathscr{S}_i = true$, then $\forall j \in Neighbors_i$, $\mathscr{I}\mathscr{S}_i \neq \mathscr{I}\mathscr{S}_j$, and if $\mathscr{I}\mathscr{S}_i = false$, then $\exists j \in Neighbors_i$, $\mathscr{I}\mathscr{S}_i \neq \mathscr{I}\mathscr{S}_j$. The fault 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 $\mathscr{I}\mathscr{S}$ variables constructs a maximum independent set.

By the definition of $0-$distance local, no fault is detected in a configuration in which the $\mathscr{I}\mathscr{S}_i$ 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 $\mathscr{C}_i$ representing its color. In addition for every two neighboring processors P_i and P_j it holds that $\mathscr{C}_i \neq \mathscr{C}_j$.

Lemma 10 *The coloring task is* $1-$*distance local.*

Proof The proof is by presenting a fault detector in the set of $\mathscr{F}\mathscr{D}_1$ and proving the impossibility result for the existence of a fault detector for this task in $\mathscr{F}\mathscr{D}_0$.

By the definition of $1-$distance local, $\mathscr{F}\mathscr{D}_1(P_i)$ can verify that:

$$\forall j \in Neighbors_i, \mathscr{C}_i \neq \mathscr{C}_j,$$

The fault detector at P_i notices the occurrence of a fault in case the above property doesn't hold.

By the definition of $0-$distance local, no fault is detected in a configuration in which the $\mathscr{C}_i$ variable of every processor P_i holds the same value $\mathscr{C}$.

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

Lemma 11 *The topology update task is* $1-$*distance local.*

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

By the definition of $1-$distance local, $\mathscr{F}\mathscr{D}_1(P_i)$ can verify that $\mathscr{T}_i = \mathscr{T}_j$ for every neighboring processor P_j. The fault detector at P_i notices the occurrence of a fault in case there exists a neighbor for which the above equality does not hold. The above test ensures that the value of all the $\mathscr{T}$ variables is the same. In addition the fault detector at P_i checks whether the local topology of P_i (that is included in $\mathscr{V}_i^1$) appears correctly in $\mathscr{T}_i$. This test ensures that the (common identical) value of $\mathscr{T}$ is correct, since every processor identified its local topology in $\mathscr{T}$.

By the definition of $0-$distance local, no fault is detected in a configuration in which the $\mathscr{T}_i$ 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.

4 Fault Detectors for Non-silent Tasks

In this section we consider the set of non-silent tasks. Unlike the previous section that consider fault detectors for asynchronous (as well as synchronous) systems, in this section we consider fault detectors for synchronous systems. We present tasks that are $s-$history local, with $s > 1$. Here, s defines the size of the history $\mathscr{V}_i^d[1..s]$ of each processor P_i. The system is synchronous and each view in $\mathscr{V}_i^d[1..s]$ is related to a different time. This array is thereafter referred as the *local history* of processor P_i. Each $\mathscr{V}_i^d$ 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 fault detectors. We start with a trivial bounded privilege task.

4.1 Bounded Privilege

Bounded privilege, task specification Each processor P_i maintains a local boolean variable $\mathscr{P}riv_i$. For each processor P_i $\mathscr{P}riv_i$ is set to true exactly once (another variant is at least once) in every c synchronous steps ($c \geq 2$).

Lemma 12 *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 $\mathscr{P}riv_i$ 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 $\mathscr{P}riv_i$. For each processor P_i $\mathscr{P}riv_i$ 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 $\mathscr{P}riv_i = true$ then

$\mathscr{P}riv_j = false$

Lemma 13 *The bounded privilege dining philosophers task is* $1-$*distance local,* $c-$*history local.*

Proof First we prove that there exists no 0-distance local fault detector for the bounded privilege dining philosophers task.

If there existed such a fault detector, it could not possibly detect that two neighbors P_i and P_j have $\mathscr{P}riv_i = \mathscr{P}riv_j = true$ simultaneously.

Then we prove that there exists no $(c-1)$-history local fault detector for the bounded privilege dining philosophers task.

If $c = 2$, a $(c-1)$-history local fault detector would use only the current view to detect a fault. Then consider two possible runs e_1 and e_2 of a system consisting in two processors P_1 and P_2, as depicted in Figure 8. In run e_1, $\mathscr{P}riv_1$ and $\mathscr{P}riv_2$ are alternatively set to $true$ every two time units, so this run is correct. Obviously, run e_2 is incorrect since $\mathscr{P}riv_2$ is never set to $true$. Now the 1-history in any configuration C_n of e_2 is the same as the 1-history in configuration $C_{2\times n}$ of e_1. Since the fault detector must respond $true$ in any configuration of e_1, it must respond $true$ in any configuration of e_2, which is an incorrect run, thus this fault detector is unable to detect a fault.

If $c \geq 3$, then consider a system containing two processors P_1 and P_2 and consider 2 runs of this system as depicted in Figure 9. Runs e_1 and e_2 consist in a repeated pattern of size c. In runs e_1 and e_2, processors P_1 and P_2 have their $\mathscr{P}riv$ variable set to $true$ every c configurations. In addition, in each configuration of these runs, we never have $\mathscr{P}riv_1 = \mathscr{P}riv_2 = true$. Consequently, runs e_1 and e_2 are correct, so the fault detectors at P_1 and P_2 respond $true$ in any of the configurations of these runs. Now consider run e_f of the same system as depicted in Figure 10. Run e_f consists in a repeated pattern of size $2 \times c$. In this pattern, the first c configurations are the same as in e_1, while the next c configurations are the same as in e_2. In run e_f, $\mathscr{P}riv_1$ is set to $true$ every c configurations, but $\mathscr{P}riv_2$ is alternatively set to true every $c+1$ configurations, and every $c-1$ configurations, so run e_f is incorrect.

In configurations C_1 to C_{c+1} of run e_f, the $(c-1)$-history of both P_1 and P_2 is the same as in configuration C_1 to C_{c+1} of run e_1, so both fault detectors respond $true$ in these configurations of e_f. In configurations C_{c+2} to $C_{2\times c+1}$ of run e_f, the $(c-1)$-history of both P_1 and P_2 is the same as in configuration C_{c+2} to $C_{2\times c+1}$ of run e_2, so both fault detectors respond $true$ in these configurations of e_f.

Using the same reasoning, the $(c-1)$-histories of P_1 and P_2 in configurations $C_{(k-1)\times c+2}$ to $C_{k\times c+1}$ of run e_f are the same as those in run e_1 if k is odd and those in run e_2 if k is even. Since the histories in run e_f could be those of correct runs, no fault is detected in any configuration of e_f, which is an incorrect run.

On the other hand it is clear that a local history of c views is sufficient for detection of the first predicate of the task specification. In addition to ensure that no two processors are privileged simultaneously a view of diameter 1 is sufficient.

4.3 Deterministic Non-interactive Tasks

In a synchronous run (assuming a synchronous scheduler that activates all enabled processors at each synchronous

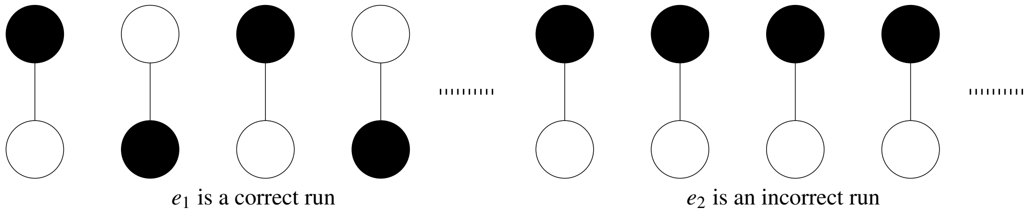


Fig. 8 P_1 and P_2 should be privileged every 2 configurations

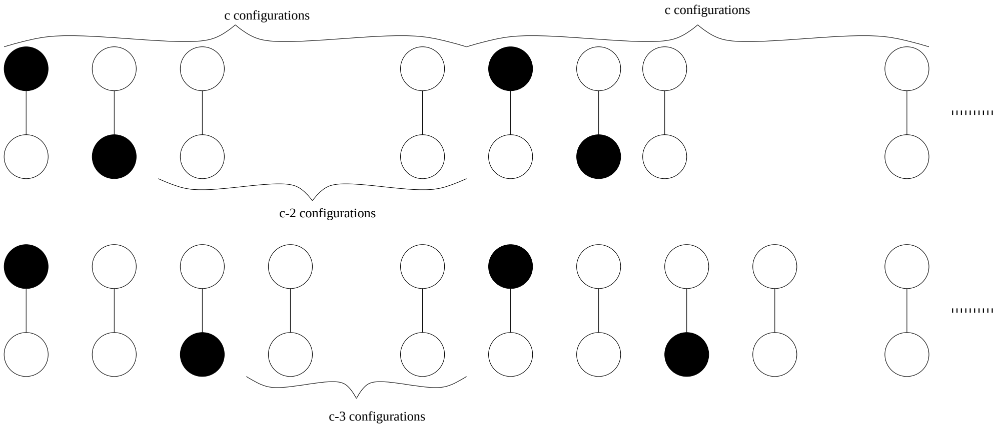


Fig. 9 Runs e_1 and e_2 are correct runs

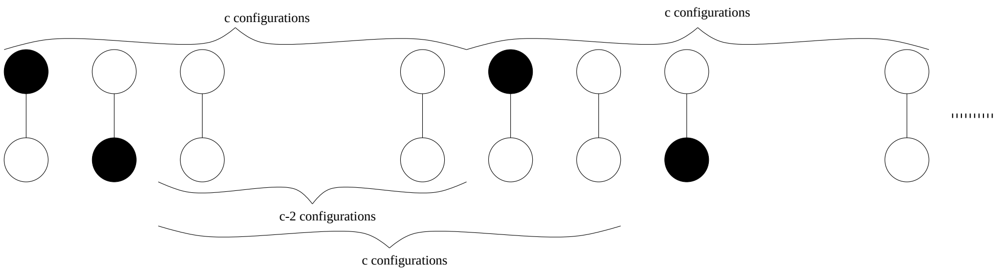


Fig. 10 Run e_f is incorrect

step) of a non-randomized, non-interactive, bounded space algorithm some configurations 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 are repeated.

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 fault detectors may serve as an implementation of the task.

4.4 Fair Privilege

Fair privilege, task specification Each processor P_i maintains a local boolean variable $\mathscr{P}riv_i$. For each processor P_i $\mathscr{P}riv_i$ is set to true infinitely often in every infinite synchronous run.

Lemma 14 *For any integers d and l, there exists no d-distance local, l-history local transient fault detector for the fair privilege task.*

Proof Assume that there exist two integers d and l such that there exists a d-distance local, $l-$history local transient fault detector for the fair privilege task.

Consider a system where in each synchronous run, processors get the same privilege status (for any processor P_i

and any configuration c, all $\mathscr{P}riv_i$ variables are equal). A run of this system is given by the successive values of $\mathscr{P}riv_i$, the variable of a single processor P_i. We consider two runs R_1 (which is correct) and R_2 (which is incorrect) and prove that if the $l-$history local transient fault detector responds *true* in R_1, it also responds *true* in R_2, and thus does not detect the faulty run.

$$R_1 = \underbrace{false,\ldots,false}_{l \text{ times}}, true, true, \ldots$$
$$R_2 = \underbrace{false,\ldots,false}_{l \text{ times}}, false, false, \ldots$$

In run R_1, the $\mathscr{P}riv_i$ variable is set to *true* infinitely often, so this run is correct. Therefore the transient fault detector at processor P_i must return *true* in any configuration.

In run R_2, the $\mathscr{P}riv_i$ variable is never set to *true*, so this run is incorrect. Therefore the transient fault detector must respond *false* in at least one configuration at node P_i.

Configurations c_0 to c_{l-1} are equal in R_1 and R_2, so the histories $\mathscr{V}_i^0[1..l]$ are equal in those configurations. Since the transient fault detector responded *true* in configurations $c_0 \ldots c_{l-1}$ in R_1 (which is correct), it also responds *true* in configurations $c_0 \ldots c_{l-1}$ in R_2. Note that in configuration c_{l-1} of either R_1 or R_2, the variable $\mathscr{V}_i^0[1..l]$ equals

$$\underbrace{false,\ldots,false}_{l \text{ times}}$$

Now consider configuration c_k of R_2, with $k \geq l$. In configuration c_k, $\mathscr{V}_i^0[1..l]$ equals $\underbrace{false,\ldots,false}_{l \text{ times}}$, thus the transient fault detector must respond *true* in configuration c_k.

Since for any configuration of R_2, the $l-$history local fault detector responds *true*, it does not detect any fault in R_2, which is an incorrect run.

5 Transient Fault Detectors for Algorithms

Unlike the case of fault detectors for tasks the fault 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 fault detector may use the distance variable to detect inconsistency: if every processor has a distance that is greater than one from 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 fault 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) fault detectors for specific sets of algorithms.

Hierarchy for fault detectors of algorithms can be defined analogously to the definition of the fault detectors for tasks. In fact, the predicates that were previously defined for the task can be replaced by new predicates for invariants that are preserved by the algorithm in every execution. The choice of the algorithm that implements a task influences the fitting fault 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 fault detector in $\mathscr{F}\mathscr{D}_1$.

6 Implementing Transient Fault Detectors

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

6.1 Updating the local views

The updating policy for the processor views is the following. Each processor P_i communicates (repeatedly in asynchronous systems, at each pulse in synchronous systems) to every of its neighbors, P_j the portion of $\mathscr{V}_i^d$ that is shared with $\mathscr{V}_j^d$. 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 $\mathscr{V}_i^d$). When P_i receives the view $\mathscr{V}_j^d$ from its neighbor P_j, P_i checks whether the shared portions of $\mathscr{V}_i^d$ and $\mathscr{V}_j^d$ agree. P_i outputs a fault 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 $\mathscr{V}_{jl}$ and $\mathscr{V}_i$ hold non equal values for a variable of P_k. Note that there must exists such processor since P_i and P_k holds different values for the variables of P_k. Clearly, P_{jl} and $P_{j(l-1)}$ identifies the inconsistency.

6.2 Updating the local histories

We define the updating policy of the local histories only for synchronous systems. 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 $\mathcal{V}_i^d[j+1]$, $1 \leq j < s$.

6.3 Towards self-stabilization

The last issue in implementing the fault detector is the action taken upon inconsistency detection. Although it is out of the scope of the paper, we mention the reset (*e.g.*, [20], [4]) and the repair operations (*e.g.*, [1], [21]), both should result in a consistent state of the system and the fault detector. The fault detector is not activated until the reset or the repair actions terminate. In particular, to collect a new view up to distance d the update algorithm [10, 13] can be used for d rounds (using a synchronizer, in the case of asynchronous system) and only then will the fault detector be activated.

7 Concluding remarks

In this paper, we investigated the amount of resources required for implementing transient fault detectors for tasks and algorithms. We presented a hierarchy of transient fault detectors that detect the occurrence of faults in a *single* asynchronous round (in an asynchronous system) or a single synchronous round (in a synchronous system). The benefits of our approach are twofold:

1. It gives the task implementer a formal framework to analyse the expected “cost” of the task, or even the implementation of the task (considered as a refined task), in terms of resources that are required for transient fault detection.
2. It provides a measure of the informal notion of locality, from a transient fault detection point of view (*i.e.* two tasks or two tasks implementations can now be compared based on the locality of their respective transient fault detectors). This is complementary to the usual notion of locality in distributed computing [23] which relates to the amount of resources needed to *solve* the task.

We suggest two interesting open problems:

1. While this paper concentrated on defining the notion of locality based on the effort needed to *detect* transient faults, an analogous definition based on the effort for *reparing* would be of great interest.
2. The transient fault detectors that we present in this paper could actually be used to detect arbitrary deviation from the specification, not just transient faults. However, the possible transient fault detector implementation that we give in this paper may only be subject to transient faults. Designing detectors that can cope with arbitrary (*i.e.* malicious) failures would turn our solution into a universal *predicate detector* mecanism.

Acknowledgements We are grateful to the anonymous reviewers that permitted to improve the contents and presentation of this paper.

References

1. Afek, Y., Dolev, S.: Local stabilizer. J. Parallel Distrib. Comput. **62**(5), 745–765 (2002)
2. Afek, Y., Kutten, S., Yung, M.: Memory-efficient self stabilizing protocols for general networks. In: J. van Leeuwen, N. Santoro (eds.) WDAG, *Lecture Notes in Computer Science*, vol. 486, pp. 15–28. Springer (1990)
3. Awerbuch, B., Patt-Shamir, B., Varghese, G.: Self-stabilization by local checking and correction (extended abstract). In: FOCS, pp. 268–277. IEEE (1991)
4. Awerbuch, B., Patt-Shamir, B., Varghese, G., Dolev, S.: Self-stabilization by local checking and global reset (extended abstract). In: G. Tel, P.M.B. Vitányi (eds.) WDAG, *Lecture Notes in Computer Science*, vol. 857, pp. 326–339. Springer (1994)
5. Burns, J.E., Gouda, M.G., Miller, R.E.: Stabilization and pseudo-stabilization. Distributed Computing **7**(1), 35–42 (1993)
6. Chandra, T.D., Toueg, S.: Unreliable failure detectors for reliable distributed systems. J. ACM **43**(2), 225–267 (1996)
7. Delaët, S., Ducourthial, B., Tixeuil, S.: Self-stabilization with r-operators revisited. Journal of Aerospace Computing, Information, and Communication (2006)
8. Delaët, S., Tixeuil, S.: Tolerating transient and intermittent failures. Journal of Parallel and Distributed Computing **62**(5), 961–981 (2002)
9. Dijkstra, E.W.: Self-stabilizing systems in spite of distributed control. Commun. ACM **17**(11), 643–644 (1974)
10. Dolev, S.: Self-stabilizing routing and related protocol. J. Parallel Distrib. Comput. **42**(2), 122–127 (1997)
11. Dolev, S.: Self-Stabilization. MIT Press (2000)
12. Dolev, S., Gouda, M.G., Schneider, M.: Memory requirements for silent stabilization. Acta Inf. **36**(6), 447–462 (1999)
13. Dolev, S., Herman, T.: Superstabilizing protocols for dynamic distributed systems. Chicago J. Theor. Comput. Sci. **1997** (1997)
14. Dolev, S., Israeli, A., Moran, S.: Self-stabilization of dynamic systems assuming only read/write atomicity. Distributed Computing **7**(1), 3–16 (1993)
15. Dolev, S., Israeli, A., Moran, S.: Analyzing expected time by scheduler-luck games. IEEE Trans. Software Eng. **21**(5), 429–439 (1995)
16. Dolev, S., Kranakis, E., Krizanc, D., Peleg, D.: Bubbles: adaptive routing scheme for high-speed dynamic networks (extended abstract). In: STOC, pp. 528–537. ACM (1995)
17. Ducourthial, B., Tixeuil, S.: Self-stabilization with r-operators. Distributed Computing **14**(3), 147–162 (2001)
18. Ducourthial, B., Tixeuil, S.: Self-stabilization with path algebra. Theoretical Computer Science **293**(1), 219–236 (2003). Extended abstract in Sirrocco 2000
19. Ghosh, S., Gupta, A., Herman, T., Pemmaraju, S.V.: Fault-containing self-stabilizing algorithms. In: PODC, pp. 45–54 (1996)
20. Katz, S., Perry, K.J.: Self-stabilizing extensions for message-passing systems. Distributed Computing **7**(1), 17–26 (1993)
21. Kutten, S., Patt-Shamir, B.: Time-adaptive self stabilization. In: PODC, pp. 149–158 (1997)
22. Lin, C., Simon, J.: Observing self-stabilization. In: PODC92 Proceedings of the Eleventh Annual ACM Symposium on Principles of Distributed Computing, pp. 113–123 (1992)
23. Peleg, D.: Distributed Computing: A Locality-Sensitive Approach. SIAM Monographs on Discrete Mathematics and Applications (2000)