Computing (2023) 105:2821–2845
https://doi.org/10.1007/s00607-023-01211-8

REGULAR PAPER

The missing piece: a distributed system-level diagnosis model for the implementation of unreliable failure detectors

Elias P. Duarte Jr.[1] · **Luiz A. Rodrigues**[2] · **Edson T. Camargo**[3] · **Rogério C. Turchetti**[4]

Received: 20 October 2022 / Accepted: 7 August 2023 / Published online: 18 August 2023

Abstract
Reliable systems require effective monitoring techniques for fault identification. System-level diagnosis was originally proposed in the 1960s as a test-based approach to monitor and identify faulty components of a general system. Over the last decades, several diagnosis models and strategies have been proposed, based on different fault models, and applied to the most diverse types of computer systems. In the 1990s, unreliable failure detectors emerged as an abstraction to enable consensus in asynchronous systems subject to crash faults. Since then, failure detectors have become the *de facto* standard for monitoring distributed systems. The purpose of the present work is to fill a conceptual gap by presenting a distributed diagnosis model that is consistent with unreliable failure detectors. Properties are proven for the number of tests/monitoring messages required, latency for event detection, as well as completeness and accuracy. Three different failure detectors compliant with the proposed model are presented, including vRing and vCube, which provide scalable alternatives to the traditional all-monitor-all strategy adopted by most existing failure detectors.

Keywords Distributed systems · Fault tolerance · System-level diagnosis · Failure detection · Fault management · Fault monitoring

Mathematics Subject Classification 68M14 · 68M15

1 Introduction

As computer systems have become part of the fabric of human organizations, failures can have serious consequences [1, 2]. It is essential to design systems that continue to work correctly even if some components become faulty. Fault tolerance emerged soon after the first digital computers were developed. In the 1950s, Von

Extended author information available on the last page of the article

Neumann himself investigated the construction of reliable computers based on unreliable components [3]. Today, dependability is a well-structured domain that encompasses the properties that reflect the degree of confidence that can be placed in a system [4].

There are several techniques for building fault-tolerant systems [5, 6]. Most of those techniques exploit redundancy, both explicitly and implicitly. In explicit redundancy, components are replicated to avoid single points of failure. For example, instead of a single secondary memory unit, multiple units are employed so that the failure of a single unit does not affect the correct operation of the system as a whole. Redundancy can also be implicit, meaning that it is part of the system, such as in fault-tolerant routing [7]. In the case of distributed systems, redundancy is implicit and intrinsic, since a distributed system by definition consists of a set of $n \geq 2$ processes that communicate and collaborate to accomplish a task [8].

Among the dependability properties, availability indicates the percentage of time a system is expected to operate correctly, taking into account failures and recoveries. An effective failure recovery mechanism is required to increase availability. The idea is to reduce the time the system remains unavailable as much as possible. Recovery usually begins with the determination that a failure has occurred. Although some dependable systems do not require explicit fault identification—for example, those that make decisions based on majority voting—most fault-tolerant systems follow the classical model of fault identification, fault isolation, and system reconfiguration [6].

Fault identification is also an old problem: the first model of diagnosable systems appeared in the 1960s: the PMC model, named after the authors' initials [9]. According to the PMC model, system units perform tests on each other. Based on the test results, you can determine which units are faulty. The PMC model assumes that a correct tester can accurately determine and report the actual state of each tested unit. Based on the PMC model, an extraordinary number of results have been obtained in the field, including various models applied to different types of systems. These results have made it possible to understand the limits of diagnosis and define a wide variety of strategies for identifying faults [10, 11].

In the context of distributed systems, the so-called FLP impossibility was published in 1985, also named after the authors' initials [12]. According to this key result, it is impossible to guarantee the correct execution of consensus in asynchronous distributed systems, where processes may crash. Asynchronous systems have no timing guarantees: There are no known bounds on the maximum time required to execute a task and transfer messages between processes. The root of FLP impossibility is precisely the difficulty in distinguishing a faulty process from a slow process. Given the importance of consensus—considered by many to be the central problem of distributed systems—this result has very important implications for the field as a whole.

In the 1990s, Chandra and Toueg [13] looked at the consensus problem from a different angle. The main idea was to investigate how the FLP impossibility would be affected if processes had information on failures. The authors defined a failure detector that acts as an oracle and provides information about process states. A process is reported as either correct or suspected to have failed. Failure detectors are inherently unreliable, which means that the detector may report a process state that does not correspond to reality. The authors have defined two

failure detection properties: completeness and accuracy. Informally, completeness reflects the ability of the detector to identify processes that have actually failed. Accuracy, on the other hand, is the ability of the detector to not suspect that correct processes have failed.

Most failure detectors use a monitoring strategy in which all monitored processes send heartbeat messages to every other process at regular intervals [14–16]. If the system runs on a single network segment, it is possible to implement this strategy efficiently, for example, using multicast. In other environments, the strategy does not scale well, because it requires periodic transmission of n^2 messages. Most efforts to develop scalable detectors have involved probabilistic broadcasting [17]. In contrast, going back to distributed diagnosis algorithms, they have been proposed precisely to reduce the number of messages required for system monitoring, as well as the latency for detecting new process state changes.

This work presents a model for unifying distributed diagnosis and failure detection. A new system-level diagnosis model is proposed that enables the specification of scalable failure detectors. The results are presented on bounds on the number of tests/monitoring messages required, latency for event detection, and completeness and accuracy. Three different failure detectors are presented that are consistent with the proposed model. The first is based on the traditional all-monitor-all strategy adopted by most existing failure detectors. Then, two classical diagnosis algorithms are specified in the new model: vRing and vCube, which provide scalable alternatives to the traditional all-monitor-all strategy used by most existing failure detectors. One of the major contributions of the paper is to provide several novel results for pull-based failure detectors that have not been published before.

The contributions of this work can thus be summarized as follows:

- To the best of our knowledge, this work is the first to unify distributed system-level diagnosis and unreliable failure detectors. Both approaches are extremely relevant and have been widely employed for fault monitoring.
- Multiple results are derived for the new model (which are valid for any algorithm developed under the model). The results are in terms of completeness, accuracy, number of tests required, amount of information transmitted, and latency. All of those results are valid for push-based failure detectors and represent a contribution in that context.
- Three algorithms are specified for the proposed model. Multiple properties are derived for each of them. Note that an algorithm specified for the new model is both a diagnosis algorithm and a failure detector.

The remainder of this paper is organized as follows. The next section provides an overview of the system-level diagnosis. Section 3 defines and gives an overview of failure detectors. Section 4 introduces the distributed diagnosis/ failure detector model and presents basic results, including the number of tests needed. Section 5 shows the results for completeness and accuracy of fault diagnosis/detection. Section 6 presents proofs of the best and worst detection latency for the model. Section 7 presents the three failure detectors for the proposed model. Finally, the conclusion can be found in Sect. 8.

2 System-level diagnosis: an overview

In 1967, Preparata et al. [9] published the first model for "diagnosable" systems, called the PMC, named after the author's initials. In the PMC model, the system consists of units that can test each other. A test is a procedure that is complete enough to determine whether the tested unit is *faulty* or *fault-free*. The set of all test results is called the *syndrome* of the system. A central entity external to the system collects and processes the syndrome to classify the units as faulty or fault-free.

Interestingly, the PMC model adopted a fault model that corresponds to Byzantine errors today. Although no threats or malicious components are assumed, a faulty unit produces an arbitrary output. More specifically, faulty units perform tests and report test results that can have arbitrary values. In contrast, the PMC model assumes that a fault-free unit runs the tests correctly and reports the correct test results. Thus, depending on the number of tests performed and the state of the testers, the syndrome may or may not allow the correct identification of faulty units. Figure 1 shows a classic PMC example. The system consists of $n = 5$ units connected by arcs that represent tests with labels indicating the results: 1 for faulty and 0 for fault-free. In the example, at most a single unit can be faulty in correctly determining the states of all units from processing the syndrome. If, on the other hand, there are two or more faulty units, the problem becomes impossible: there is no way to tell which unit is faulty. The concept of *diagnosability* was defined to reflect the ability of a system to diagnose f failures. An f-diagnosable system can correctly identify up to f faulty units. The system in Fig. 1 is 1-diagnosable.

The set of tests performed on the system was originally called *connection assignment* and later became called *test assignment*. In the early years of diagnosis, much of the research focused on determining testing assignments with a low number of tests and high *diagnosability*, which would allow an effective and efficient diagnosis. In 1974, Hakimi and Amim proved that for a system of n units to be t-diagnosable it is necessary that (i) $n \geq 2t + 1$ and that (ii) each process be tested for at least t other processes [18].

Ten years later, in 1984, two new diagnosis models were proposed that had a major impact on the field. The first model by Hakimi and Nakajima [19] allows the diagnosis to be adaptive, in the sense that the testing assignment is dynamic, i.e., the next tests to be executed are defined according to the results of the previous tests. The possibility of employing adaptive tests introduces a time dimension to diagnosis: at first, a set of tests is executed, then their results are evaluated, and finally new

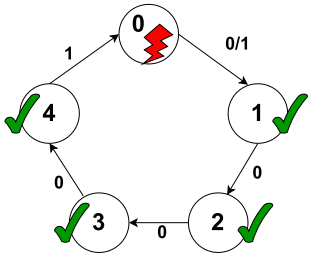


Fig. 1 The classic PMC model example

tests are defined for execution in the next round. The other model was proposed by Hosseini et al. [20], and eliminated the central entity, allowing for a fully distributed diagnosis. According to that model, the units not only execute tests, but also collect test results from each other to obtain the syndrome. Each fault-free unit can then process the syndrome and determine the set of faulty units in the system.

Only in the following decade, in 1992, Bianchini and Buskens [21] proposed the combination of adaptive and distributed diagnosis. The Adaptive Distributed System-Level Diagnosis algorithm (Adaptive-DSD) was proposed and used to monitor a large production network. The algorithm is based on a ring topology that presents the optimal number of tests: at most n, but the latency is up to n consecutive testing rounds. In 1998, Duarte and Nanya [22] proposed the Hierarchical Adaptive Distributed System-Level Diagnosis algorithm (Hi-ADSD). Hi-ADSD employs a hierarchical virtual topology to organize the system units. Latency is at most $log^2 n$ testing rounds, and all logarithms in this work are base 2.

The Hi-ADSD algorithm was originally proposed for the construction of efficient and fault-tolerant network fault management systems, and has been implemented using the Internet management protocol SNMP (*Simple Network Management Protocol*) [23]. Later, the algorithm was slightly modified to ensure that the number of tests executed does not exceed $nlogn$ per n testing rounds [24]. In the new version, the algorithm is called vCube, and has been used as a failure detector to build reliable distributed algorithms [25, 26].

The system-level diagnosis results presented above assume the PMC model, and a fully connected system, i.e., the topology is a complete graph. There are actually several other models, for instance, those that assume a system of arbitrary topology [27], which are partitionable by definition. In these models, the problem of determining which units are faulty becomes the problem of computing the reachability. Other works consider different diagnosis models. Camargo et al. [28] abandoned the assumption of the PMC model by assuming that correct units can make mistakes during tests. Comparison-based diagnostics [11, 29] do not assume crash faults. Faulty units can produce arbitrary outputs. The outputs of multiple units are compared to perform the diagnosis, such as distributed integrity checking [30]. Although most models assume that faults are permanent, some approaches have been proposed to diagnose intermittent faults [31, 32]. Finally, the probabilistic diagnosis model also presents an alternative approach, which takes into account the difficulty in defining deterministic tests [10].

3 An overview of unreliable failure detectors

The output produced by an unreliable fault detector is exactly the same as that produced by a system-level distributed diagnosis algorithm: the list of processes considered faulty. However, failure detectors were proposed in a completely different context [13, 33]. The original motivation for their development was the impossibility of consensus in asynchronous systems with crash faults. Chandra and Toueg investigated what would happen if a group of processes executing consensus had access to information about the process state. In a sense, they were investigating the

limits of the FLP impossibility. The question that arises here is whether this "extra" information would enable consensus.

Therefore, failure detectors were not proposed to be efficient strategies for process monitoring. The focus was on the properties that a failure detector should have to help solve consensus. Then two essential properties were then defined: *completeness* and *accuracy*. Informally, completeness reflects the ability of a failure detector to suspect failed processes. Accuracy, on the other hand, requires the failure detector not to suspect correct processes. With respect to completeness, two observations can be made. First, as crash faults are assumed, a process that has failed does not produce any response to any stimulus, so it is not difficult to guarantee that the failure detector raises suspicions about what has occurred. On the other hand, the monitored process may fail immediately after the failure detector has successfully exchanged the messages that were used to determine its correctness. Thus, it may take some time after the failure occurred for the suspicion to be raised.

Accuracy, on the other hand, can never be guaranteed in asynchronous systems. The problem is classic: It is difficult (impossible, in fact) to distinguish a process that has failed from a slow process. A correct process may be slow to execute tasks or communicate. This is exactly the cause of the FLP impossibility. However, if the failure detector does not suspect that correct processes have failed, what impact can it have on consensus? In [34], the authors show that even if eventually a single correct process is never suspected, then consensus can be solved in an asynchronous distributed system with *crash* faults. This is a far-reaching result that has consequences in both theory and practice. The unsuspected correct process can be elected leader and have multiple responsibilities in a distributed algorithm.

Completeness and accuracy can be classified as weak or strong [13], which comprises eight classes of failure detectors, as described below. Completeness is strong if all failed processes are eventually suspected by all correct processes. On the other hand, completeness is weak if, after a time interval, all failed processes are suspected by at least one correct process. It is not difficult to convert weak completeness into strong completeness. The correct process that detected the failure can broadcast the information to the other correct processes.

Accuracy can also be strong or weak. Strong accuracy means that the failure detector does not suspect that any correct processes have failed. Weak accuracy requires only that at least a single correct process never be suspected. These two properties are further extended as eventual weak/strong accuracy: no/one (respectively) correct process is eventually suspected of having failed. The weakest class of failure detectors (called $\diamond W$) presents weak completeness and eventual weak accuracy. Even that class of failure detectors guarantees the correct execution of consensus in asynchronous systems with crash faults, as mentioned above. Note that an asynchronous system enhanced with failure detection is no longer "purely" asynchronous, since it involves timing properties.

Classical implementations of failure detectors require the periodic transmission of heartbeat messages by each monitored process to all others [14, 16, 35]. This strategy is efficient when the distributed system runs on a single physical network based on broadcast, as it requires only a single message to send each heartbeat to all processes. On the contrary, for systems running on point-to-point

networks or on more than one network, the strategy does not scale because it requires regular transmission of n^2 messages. That is one of the motivations to define the system-level diagnosis model for building failure detectors based on tests presented in the next section.

4 A system-level diagnosis model for unreliable failure detection

In this section, we present the new distributed diagnosis model for the implementation of classical fault detectors. The model assumes a distributed system Π consisting of n processes that communicate by using message passing. Processes have sequential identifiers, $\Pi = \{0, 1, .., n-1\}$. The system is fully connected and is represented by a complete undirected graph $K_n = (\Pi, E)$, where E is the set of edges, $E = \{\{i,j\} \mid 0 \leq i,j < n \text{ and } i \neq j\}$. In this model, any two processes can communicate directly with each other without passing through an intermediary.

The system is asynchronous, i.e., the maximum time required to transmit a message or execute a task is unknown. The crash model is assumed, so a faulty process completely loses its internal state and produces no output for any input. A process can be in one of two states: *correct* or *failed*, also referred to as *fault-free/crashed*, respectively. The function $state(i)$ returns the state of the process i: $state(i) = \{ failed \mid correct\}$. An *event* is defined as the transition of the state of a process from *correct* to *failed*. Although it is not difficult to extend the model to include process recovery and also to allow a process to suffer multiple events over time, this paper assumes only permanent crash faults without recovery due to space limitations.

Processes test each other. The purpose of a test is to allow the tester to determine the state of the tested process. The outcome of a test can indicate either that the tested process is *correct* or *suspect* of having failed. A test consists of a set of stimuli sent by the tester to the tested process, for which proper replies are expected. The specific testing procedure adopted varies according to the technology of the system, and other factors such as specific functionalities can be checked to be correct. After the tester receives the proper response, it classifies the tested process as correct. Considering two processes $i, j \in \Pi$, a test executed by the tester j on the tested process i is defined as function $test_j(i) = \{correct \mid suspect\}$. A proper timeout mechanism has to be adopted to limit the time the tester will wait for a reply [15]. The testing procedure has to be strong enough to avoid mistakes as much as possible, thus, for instance, the decision to classify the tested process as suspect must be taken after more than a single timeout. Corollary 1 establishes a correlation between the states of a tested process and its classification after a test.

Corollary 1 *A process tested as correct is indeed in this state, thus* $\forall i, j \in V$, *if* $test_j(i) = correct$, *then* $state(i) = correct$. *On the other hand, a process classified as suspect can be either failed or correct due to slowness that caused a timeout when tested. Thus, if* $test_j(i) = suspect$, *state* $state(i) = \{correct \mid failed\}$.

The set of tests is called *testing assignment* and is represented by a directed graph $A = (\Pi, T)$. The set of arcs T represents the tests and an arc (i, j) indicates that process i tests process j (i.e., j is the tested process in this case). The tests are executed periodically, in testing intervals determined by each tester according to its local clock. The set of processes does not employ synchronized clocks or a global clock. Thus, the testing interval of a tester can be different from the testing interval of another tester. The only assumption is that local clocks move forward asymptotically. A *testing round* occurs after all processes have executed their assigned tests, i.e., all tests in T are performed. Testing rounds can be enumerated as $r_1, r_2, ...$ The first round corresponds to the first time tests are executed, and so on.

The purpose of diagnosis is to allow the correct processes to obtain, after a finite number of testing rounds, a classification of the state of *all* processes. Thus, $\forall i, j \in \Pi$, process j classifies process i with function $state_j(i) = \{correct \mid suspect\}$. Different processes can have different classifications for a given process, depending on the time instant in which the tests were executed, and the flow of test results among correct processes.

The latency L is defined as the number of testing rounds it takes for all correct processes to diagnose an event. Thus in $r_e, r_{e+1}, ..., r_{e+L}$ rounds, $\forall k, j \mid state(k) = correct$ and $state(j) = correct$, $state_k(i) = state_j(i)$. The latency is proportional to the diameter of A, the testing assignment graph.

We assume that if a given process fails, all correct testers will suspect the process in the next testing round after the failure. This assumption is trivial to guarantee. As a crashed process does not send any reply to any test, all testers will timeout the next time the crashed process is tested. Corollary 2 makes it clear that a test executed on a crashed process will always lead to suspicion.

Corollary 2 *If process i has crashed and thus state*$(i) =$ *failed, then for every correct process j that has tested i* $state_j(i) =$ *suspect.*

Corollary 3 presented next determines that each process must be tested periodically by a correct process. Note that if a process is not tested, it is not possible to detect an event that might occur with that process. In the present model, to ensure that all processes are properly monitored, each process must be tested by a correct process in each testing round. In each testing round r, $\forall i, \exists j$ such that $state(j)$=*correct* and $(j, i) \in A$.

Corollary 3 *Each process* $i \in \Pi$ *is tested by a correct process at each testing round if there is one. Thus, it is guaranteed that any event that might occur at i is detected.*

Note that instead of having every process tested in each testing round, an alternative would be to ensure that each process is tested once in m testing rounds. Although that alternative would reduce the total number of tests, it would have an impact in terms of the time it takes to detect an event.

A process obtains diagnostic information by either testing other nodes or obtaining the information from correctly tested nodes. However, if the correct process i

tested process j in the current testing round, it should not obtain diagnostic information about j from other tested processes. Furthermore, according to Corollary 4, it must be ensured that each correct process receives information about all system processes in each testing round. The test assignment A can be defined so that all processes can efficiently obtain all the necessary diagnostic information.

Corollary 4 *In each testing round, a correct process obtains diagnostic information about all other processes, either by testing those processes or by receiving information from other correctly tested processes.*

In an asynchronous system, a correct tester may suspect a correct tested process, which may be slow to respond to the test. In an extreme situation, all correct processes may suspect all other (correct) processes. In this case, all processes will test all other processes, as shown in Theorem 1.

Theorem 1 *Assuming that each correct process executes its assigned tests once per testing round, any diagnosis algorithm specified according to the proposed model executes $n^2 + n$ tests per round in the worst case.*

Proof If in a testing round, all n processes are correct, but each process suspects all other $n - 1$ processes, then each process tests all others. As all n processes do the same, $n * (n - 1) = n^2 - n$ tests are performed in total in that round. □

Actually, there is no assumption on the speed at which processes execute their tests. Thus, one process can be much faster than others and execute its assigned tests multiple times in a given testing round, whereas the slowest process executes its assigned tests only once. In this way, the assumption in Theorem 1 that processes execute their assigned tests once per testing round guarantees that the number of tests is not greater than $n^2 + n$.

In summary, the proposed model is $(n - 1)$-diagnosable, meaning that all but one process can fail and the diagnosis can still be completed. The remaining correct process tests all the others and completes the diagnosis of the system.

5 Diagnosis completeness and accuracy

In this section, we introduce the two classical properties of failure detection—completeness and accuracy—for the proposed model. Informally, completeness requires that faulty processes are suspected by correct processes. Accuracy requires that correct processes do not raise false suspicions, such that correct processes self-identify as correct. The properties are then formally defined and, following the original definition in [13], classified as *weak* and *strong*.

The diagnosis satisfies strong completeness if all correct processes classify every failed process as a suspect after a finite number of testing rounds since the corresponding failure event occurred. Consider a failure event in the process

$i \in V$, thus i's state changes from *correct* to *failed*. Strong completeness requires that after a finite number of rounds $\forall j \in V$ such that j is *correct*, j classifies i as *suspect*. Weak completeness, on the other hand, only requires that at least one *correct* process classifies every *failed* process as *suspect*. Thus, for $i \in V$, after a failure event at i, in a finite number of rounds $\exists j \in V \mid state(j)$=*correct*, and $state_j(i) = suspect$.

In the proposed system-level diagnosis model, it is impossible to convert weak completeness directly to strong completeness, as is done in Ref. [13]. The reason is that the correct tester j who suspected a failed process i can itself be *suspected* (incorrectly and indefinitely) by any other correct process in the system. Figure 2 shows an example: process 0, which is correct, tests the failed process 1 and classifies it as *suspect*. However, all other processes suspect 0, so it is not possible to use the strategy that the only process that correctly raised the suspicion communicates that information to the remaining correct processes, as is done in [13]. However, this does not prevent any algorithm designed according to the proposed model from satisfying strong completeness, as will be shown below.

In the proposed model, to guarantee strong completeness, it suffices to guarantee that a correct process, say k, executes a test on a failed process i or receives this information from another process tested as correct. By Theorem2, any algorithm specified according to the proposed model satisfies the strong completeness condition. After a finite number of testing rounds, each correct process will have tested the failed process or received this information from another correct process, which in turn will have tested the failed process or received this information from another correct process, and so on.

Theorem 2 *Any diagnosis algorithm specified according to the proposed model satisfies strong completeness.*

Proof The proof is based on the induction of the diameter of the testing assignment A. Consider a failed process i, such that $state(i)$=*failed*. Induction basis: if the diameter is one, then all processes test all other processes. According to Corollary 2, $\forall j \in V, j \neq i$ and $state(j)$=*correct*: $state_j(i) = suspect$. Induction hypothesis: If the diameter of the testing assignment $A = d$ the completeness is strong, i.e., all processes at a distance at most of d with respect to the failed process i effectively suspect i. Induction step: increment the diameter to $d + 1$, including the processes that test those processes at a distance d from i. These processes will obtain diagnostic information on the failure of i from the processes tested. □

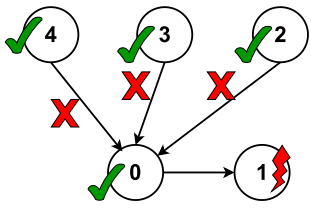


Fig. 2 Process 1 has failed and is tested as suspect by process 0. Eventually, correct processes 2, 3 and 4 will also suspect that process 1 has failed

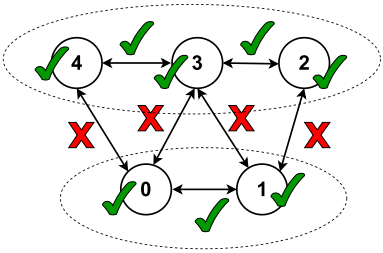


Fig. 3 Precision can only be satisfied if all correct processes are strongly connected in A

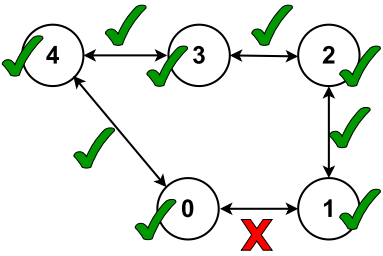


Fig. 4 In this example, although all correct processes are strongly connected among themselves, processes 0 and 1 only test and suspect each other, thus the accuracy is not strong

Next, Theorem 3 proves that to guarantee strong accuracy in the proposed model, in addition to being strongly connected among themselves, correct processes cannot raise any false suspicion about each other.

Theorem 3 *Consider any algorithm specified according to the proposed model. Strong accuracy is only guaranteed if all correct processes are strongly connected among themselves in the testing assignment A and no correct process suspects any other correct process.*

Proof The proof is similar to that of Theorem 2: if no correct process suspects any other correct process and all correct processes are strongly connected among themselves in A, then after at most d testing rounds, all correct processes obtain diagnostic information about each other. In other words, $\nexists k \in V$ such that $state(k) = correct$ and $state_k(i) = suspect$ indefinitely. □

It is easy to see that the correct processes must be strongly connected between themselves in A to guarantee strong accuracy. For example, in Fig. 3, processes 0 and 1 are correct, test each other, and do not suspect each other. However, all other correct processes suspect both and, thus, all correct processes are not strongly connected, and strong accuracy is not satisfied.

Theorem 3 shows that in addition to being strongly connected between themselves, no correct process can suspect any other correct process to ensure strong accuracy. For example, Fig. 4 shows contradictory tests for processes 0 and 1. Let i be a correct process. Both j and k test i, and they are the only processes that test i in A. Suppose that j determines that i is correct, while k suspects i. Some of the other correct processes will obtain diagnostic information on i from j and others from k, so the process i may remain suspect indefinitely, which breaks the accuracy.

6 Failure detection latency

Recall that a testing interval defines the frequency with which a process executes its assigned tests. It is a time interval defined by the local clock. A test round, on the other hand, occurs after all correct processes have executed their assigned tests. The length of a testing round depends on the slowest tester. The *failure detection latency* is defined as the number of testing rounds required to detect an event. Thus, *after* an event occurs on process j, the next round of testing counts as the first for latency and so on until $\forall i \in \Pi$ is such that i is correct, $state_i(j)$ *suspect*, and the local timestamp i for j is equal to 1.

Next, we prove theorems for latency and failure detection for best and worst cases for each algorithm specified according to the proposed model. We assume a single event, i.e., the next event cannot occur until the previous event has been fully diagnosed, i.e., the failure has been detected by all correct processes.

Theorem 4 *In the best case, the failure detection latency of any algorithm specified according to the proposed model is one test round.*

Proof After an event has occurred at process j, in the next testing round every correct process either tests j and discovers the event, or obtains information about j from another process tested as correct. The order in which tests are executed in the next rounds after the event occurs affects the latency. Suppose that the first correct processes that execute tests in that round are j's testers. After that, consider that all tests are executed by processes that do not test j directly, but test j's testers. If there are no false suspicions, these processes will obtain information about the event. Next, assume that all tests are executed on correct processes that already have information about the event. Thus in a single testing round, all processes obtain information about the event. □

Theorem 5 *In the worst case, the failure detection latency of any algorithm specified according to the proposed model is equal to the diameter of the testing assignment A.*

Proof Correct processes learn about some event by testing a failed process j or by getting the information from another process tested as correct. The proof is by induction on the length of the testing path in A through which the diagnostic information about the event propagates. Each tester i of a failed process j detects its failure in at most one round of testing. Now, consider k, the tester of j's tester i. Process k detects the event in 1 or 2 rounds of testing. If k tests i after i has tested j, k detects the failure in the same round of testing as i. However, if k tested i before i tested k and detected the event, k needs an additional round of testing to make the detection. Thus, if the testing path has a length of 2, the latency is at most 2 testing rounds.

Now assume that if the path has length l, it takes at most l testing rounds for all processes along the path to detect the event. Extend the testing path by one, say with

z. Process z will either detect the event in l testing rounds (a test executed after the tested process had made the detection) or $l + 1$ rounds, otherwise.

As a testing path has a length that is at most equal to the diameter of the testing assignment graph A, the latency in the worst case is equal to the diameter of A. □

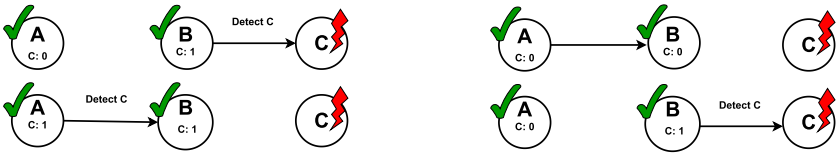
Figure 5a and b illustrate the two test situations that eventually lead to the best- and worst-case latencies. These tests are performed in a single testing round. In Fig. 5a, process B first tests process C and detects the event, updating the local diagnostic information accordingly. Then, process A tests process B as correct and receives diagnostic information about the event in process C. In a single testing round, both processes detect the event. On the other hand, in Fig. 5b, when process A tests process B, there is no information about any event, because process B will only test process C after that, detecting the event. Therefore, process A will not detect the event until the next testing round.

7 Three failure detectors

In this section, we present three failure detectors specified according to the proposed system-level diagnosis model. The first is the brute-force failure detector, where each process tests every other process in every testing interval. Next, the vRing failure detector is presented, which organizes tests on a virtual ring topology [21]. Finally, the vCube failure detector is described, which uses a hierarchical virtual topology as the testing assignment graph [24].

Figure 6 shows the proposed architecture for the deployment of the failure detection service based on the distributed diagnosis. The user is a distributed application that accesses the failure detection service through an API that allows both configuration (such as which processes are to be monitored or the algorithm to be employed) and obtaining the results (e.g. failure notifications). The specific algorithm to be employed defines a virtual testing topology, and monitoring tests are issued among the corresponding processes. Whenever a test detects a failure, the information flows back to the distributed application.

In comparison with traditional diagnosis models, the most important difference is that the new model allows tests to give incorrect results, i.e., a correct process can be suspected of having crashed. In traditional PMC-based diagnosis, the main



(a) If A tests B after B has tested C, both processes detect the event in a single round.

(b) If A tests B before B tests C, A will detect the event only in the next testing round.

Fig. 5 Two examples of tests executed in a single testing round

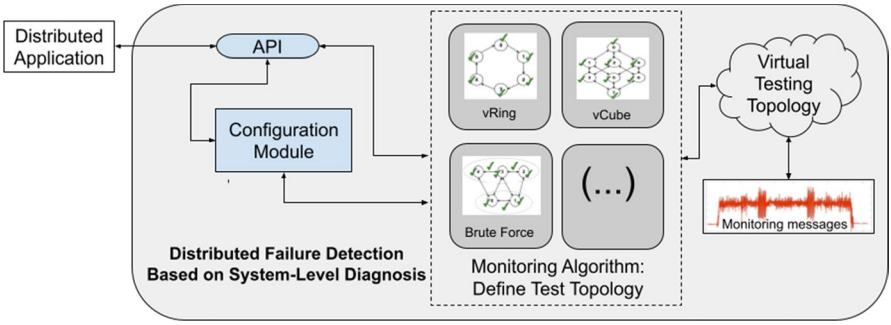


Fig. 6 Architecture: distributed failure detection based on diagnosis

assumption is that a correct tester can always accurately determine the state of the tested process. In addition, the classic properties of failure detectors—completeness and accuracy—are defined for the new model with several results being derived, completeness and accuracy have never before been considered in the context of diagnosis.

As proven in Sect. 5, any detector designed according to the proposed model satisfies the strong completeness condition, but can satisfy the strong accuracy condition only if no correct process suspects any other correct process, and all correct processes in the testing assignment graph A are strongly connected to each other. As proved in Sect. 3, any failure detector specified according to the proposed model requires n^2 tests in the worst case, but this is an extreme situation where every correct process incorrectly suspects every other correct process. All of these results are valid for the three failure detectors.

We describe the three detectors and show the trade-off in terms of the number of tests required, the amount of information transferred between testers, and the latency in worst-case failure detection. In Sect. 6, it was shown that, in the best case, each fault detector corresponding to the proposed model has the latency of a single testing round. The results in this section consider the worst-case latency, which is shown to be equal to the diameter of the testing assignment graph A. The results in this section assume that each correct process performs a single test per round. However, if some process is faster and performs more tests, this only affects the total number of tests executed.

As a process starts the execution of any of the three failure detection algorithms presented, all processes are considered to be in the *unknown* state, except the tester itself, which always assumes itself to be correct. The procedure + Test (i,j) + is executed by the tester process i on the tested process j. +Test(i,j)+ allows i to classify the tested process j either as *correct* or *suspect*. A test consists of a procedure that is tailored for the system technology and may involve a sequence

of message exchanges, including retransmissions, as well as the execution of multiple tasks by j. The +Test(i,j)+ procedure encapsulates timing assumptions, as it is necessary to give up waiting for a response eventually. As mentioned above, a correct process assumes itself to be correct, and there is no self-test, i.e., $\nexists$ +Test(i,i)+. A tester may obtain diagnostic information from a correctly tested process.

Diagnostic information consists of the process identifier and the corresponding state: -1 (*unknown*), 0 (*correct*), or 1 (*suspect*).

7.1 The Brute-force failure detector

The Brute-force failure detector requires every process to monitor directly all other processes. In the proposed model, this means that each process tests all other $n-1$ processes in every testing round. Thus, the procedure +Test(i,j)+ is executed by every correct process i at the beginning of each new testing interval on each process j, $\forall j \in \Pi \mid j \neq i$. The pseudocode of the Brute-Force failure detector is presented in Algorithm 1.

Algorithm 1 Brute-Force FD executed by each correct process i

```
1: upon the start of a new testing interval do
2:     for all process j ∈ Π, such that j ≠ i do
3:         state_j ← TEST(i, j)
4:     end for
5:     Sleep until the next testing interval
```

The Brute-force failure detector algorithm requires the execution of $n^2 - n$ tests per test interval when no process executes more than one test per round. On the other hand, the tester does not need to obtain any diagnostic information from the tested processes. The latency for detecting an event is at most 1 test round after the event has occurred.

7.2 The vRing failure detector

The vRing (Virtual Ring) Failure Detector is inspired by the Adaptive Distributed System-level Diagnosis (Adatptive-DSD) algorithm [21]. The pseudocode is presented in Algorithm 2. According to vRing, each process i performs a test on the process $j = (i+1) \mod n$. If this process tests correctly, then process i receives diagnostic information about all processes except itself and the processes it tested in the current interval. Otherwise, if process j is suspected, process i tests the next process in the ring until a correct process is found or all processes are suspected. After testing a correct process and receiving diagnostic information, the tester is done for the test interval.

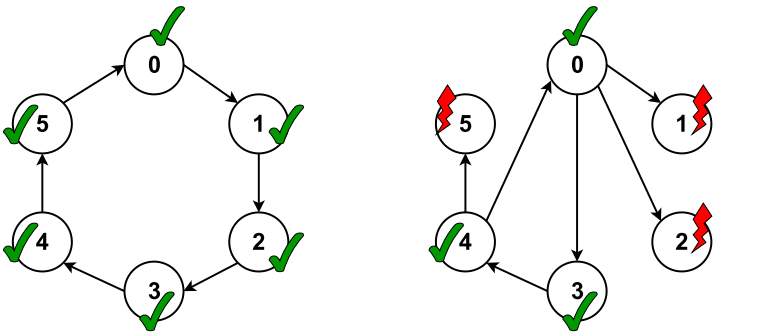
Algorithm 2 vRing FD executed by each correct process i

```
1: upon the start of a new testing interval do
2:     j ← i
3:     repeat
4:         j = (j + 1) mod n
5:         state_j ← TEST(i, j)
6:     until j is tested correct or every process is suspected
7:     if j is correct then
8:         Obtain new diagnostic information from j about all processes except i and
       those tested in the current interval
9:     end if
10:    Sleep until the next testing interval
```

Figure 7a shows a vRing with $n = 6$ processes, none of which have crashed and no test has raised a false suspicion. Figure 7b shows the same example system, but after processes 1, 2, and 5 have crashed. Therefore, process 0 tests 1 as suspected, 2 as suspected, and finally stops after testing process 3 as correct, from which it receives information about processes 4 and 5. Process 4 tests process 5 as suspected, tests process 0 as correct and receives information about processes 1, 2 and 3.

In the proposed model, in the best case, each process running vRing tests a single process per round of testing. If some processes crash, more tests are performed. On the other hand, if there are no false suspicions, even in the worst case, each process (whether correct or crashed) is tested once per round. As shown in Corollary 5, vRing uses the optimal (minimum) number of tests required by the proposed model, namely n tests per testing interval. However, the worst case remains the one proved in Theorem 1, and every correct process i tests every process j, $\forall j \in \Pi \mid j \neq i$. If all processes raise false suspicions about all other processes, $n^2 - n$ tests are executed.

Corollary 5 *According to Corollary 3, each process must be tested once per testing round by a correct tester, if there is one. Thus, if there are n processes, the minimum number of tests that can be executed per testing round is n, if fewer than n tests are*



(a) All processes are correct (b) Process 1, 2 and 5 are crashed

Fig. 7 A vRing with $n = 6$ process

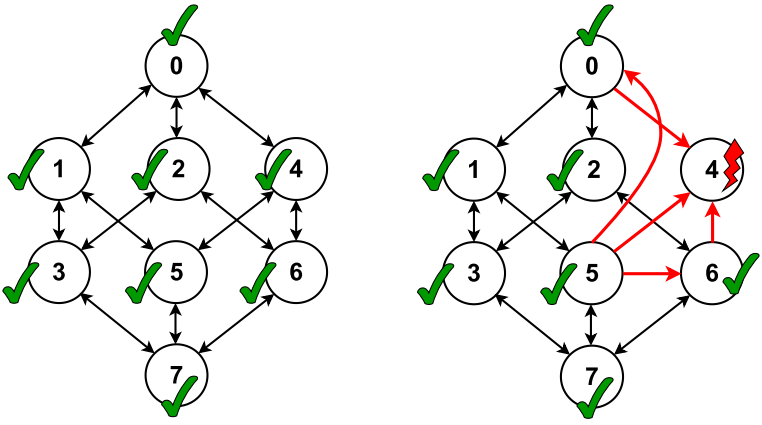
executed, one or more processes are left untested. If there is a single correct process and all others have crashed, then $n-1$ *tests are executed.*

If there are at least two correct processes and no false suspicions, the vRing failure detector requires the execution of n tests per testing interval. On the other hand, if all processes are correct, each tester needs to obtain diagnostic information about $n-2$ other processes from each tested process. Thus, at most n^2-2n items of diagnostic information could theoretically be transferred between processes in each testing round. This amount of information can be easily reduced by having each tested process record locally which information it had previously sent to its tester so that the next time it is tested only new information is transferred. Furthermore, if the corresponding timestamp is equal to -1 (*unknown*), the tester does not obtain the item. This information is transferred as processes learn the states of each other.

In terms of latency, according to Theorem 1, the best case requires a single testing round, and the worst case latency is the diameter of the test assignment graph A, which in this case is $n-1$ testing rounds.

7.3 The vCube failure detector

The vCube (virtual Cube) Failure Detector is inspired by the VCube distributed diagnosis algorithm [24]. Similar to the vRing failure detector, each process has a sequential identifier that is used as a parameter to define the tests to be executed. However, instead of a ring, vCube builds a hierarchical virtual topology that is a hypercube when all processes are correct. Figure 8a shows a vCube with $n=8$ correct processes. Bidirectional edges indicate that both processes test each other. The virtual topology corresponds to the graph $A=(V,T)$. In each testing round, each correct process i performs its assigned tests, i.e., process i tests process j if $(i,j)\in T$. If all processes are correct and there are no suspicions, the tests correspond to the



(a) All processes are correct (b) Process 4 is crashed

Fig. 8 Testing graph in a vCube with $n=8$ processes

hypercube edges. Thus, each process performs $\log_2 n$ tests per round, for a total of $n \log_2 n$ tests.

After a process crash, vCube autonomously reconfigures itself, maintaining several logarithmic properties, thus being scalable by definition. vCube organizes processes in increasingly larger clusters for testing. Each correct process i executes tests on $\log_2 n$ clusters, which are ordered lists of processes returned by the function $c(i, s) = (i \oplus 2^{s-1}, c(i \oplus 2^{s-1}, 1), .., c(i \oplus 2^{s-1}, s-1))$. Table 1 shows all clusters for a vCube with 8 processes.

The procedure for determining the tests to be performed is as follows. For each node i, there is an edge (j, i), if j is the first correct process in $c(i, s)$, $s = 1, .., \log_2 n$. Thus, each process is tested by exactly $\log_2 n$ testers, unless all processes in a cluster have crashed. After detecting a new event, the set of tests is recalculated. For example, in the example shown in Fig. 8b, process 4 has crashed. The tests that are different from those that are executed when all processes are correct are highlighted. The three tests that were executed by process 4 on 0, 5, and 6 have disappeared, but they still execute process 4. Two new tests have also been added: Process 5 tests processes 0 and 6 because it is the first correct process in $c(0, 3)$ and $c(6, 2)$.

When a process tests another process as correct, it receives all the information that the tested process has. The diagnostic information is timestamped to identify the most recent events, i.e. whenever the timestamp received for a particular process is greater than the local timestamp of the same process, it is updated with the largest value. The next time the same test is run, the tester only receives information about new events. Algorithm 3 presents the vCube failure detector in pseudocode as executed by any process i.

Algorithm 3 vCube FD executed by each correct process i

```
1: upon the start of a new testing interval do
2:     for all assigned test (i, j) in T do
3:         state_j ← TEST(i, j)
4:     end for
5:     if j is correct then
6:         Obtain new diagnostic information from j about all processes except i and
   those tested in the current interval
7:     end if
8:     if any newevent was detected then
9:         Recompute the set of assigned tests in T
10:    end if
11:    Sleep until the next testing interval
```

Table 1 Function $c(i, s)$ returns the clusters for $n = 8$ processes

s	c(0,s)	c(1,s)	c(2,s)	c(3,s)	c(4,s)	c(5,s)	c(6,s)	c(7,s)
1	1	0	3	2	5	4	7	6
2	2,3	3,2	0,1	1,0	6,7	7,6	4,5	5,4
3	4,5,6,7	5,4,7,6	6,7,4,5	7,6,5,4	0,1,2,3	1,0,3,2	2,3,0,1	3,2,1,0

Note that this algorithm is different from the original hierarchical distributed diagnosis algorithms [22, 24], since in this version each process tests all clusters in each test interval. In distributed diagnosis algorithms, each process tests a single cluster per interval. According to Corollary 6, the maximum number of tests performed is $n \log_2 n$ when there are no false suspicions, since each node is tested by the first correct process of each of its $\log_2 n$ clusters. The number of tests will be less than $n \log_2 n$ only if all processes in a given cluster have crashed.

Corollary 6 *Each process i running vCube checks $\forall j \in V$ in which clusters $c(j, s), s = 1...\log_2 n$ it is the first correct process. As each process j has $\log_2 n$ clusters, if there are no false suspicions, there are at most $\log_2 n$ testers. Thus, considering all n processes, at most $n \log_2 n$ tests are executed per testing round.*

Regarding latency, the best case is one testing round, as proven in Theorem 1. Theorem 6 proves that in the worst case, there are at most $\log_2 n$ testing rounds, even if there are false suspicions.

Theorem 6 *In the worst case, the failure detection latency of vCube is of $\log_2 n$ testing rounds.*

Proof If there are no false suspicions, the diameter of the testing assignment graph created by vCube is equal to $\log_2 n$ [24]. In case there are false suspicions, the diameter can only reduce as more edges are added to T. □

Finally, the amount of diagnostic information transferred between a tester and the tested process is at most $n - 2$ items. The worst case corresponds to a situation in which the tester obtains information about every other process (except the tester and the tested processes) in a given testing round. As a tester only gets diagnostic information that is novel, and if the timestamp is not equal to -1 (*unknown*), the average amount of information transferred should be much less than that.

7.4 A comparison

In this section, we present a brief comparison of the three failure detectors listed in Table 2. The detectors are compared based on the worst-case latency (the best case is one testing round for all); the number of tests per round of tests for the case

Table 2 A comparison of the three failure detectors

Algorithm	Latency (in rounds)	No. tests	No. data items transf.
Brute force	1	$n^2 - n$	0
vRing	$n - 1$	n	$n - 2$
vCube	$\log_2 n$	$n \log_2 n$	$(n - 2)$

where there are no false suspicions (in the worst case, all employ$n^2 - n$ tests); plus the amount of information transferred per test per round in the worst case.

In terms of the number of tests, the Brute-Force algorithm always requires $n^2 - n$ tests per round, while vRing is optimal and requires n tests and vCube requires $n \log_2 n$ tests. On the other hand, the latency of the Brute-Force algorithm is optimal as it requires one testing round for all processes to detect an event, while vRing may require up to $n - 1$ testing rounds, and vCube is logarithmic. Brute force is also the best solution in terms of the amount of information transferred between the tester and the tested processes, as it does not require any information. Both vRing and vCube can require up to $n - 2$ items of information transferred per test, but can easily be configured to transfer only new information, which should be very small in a steady state.

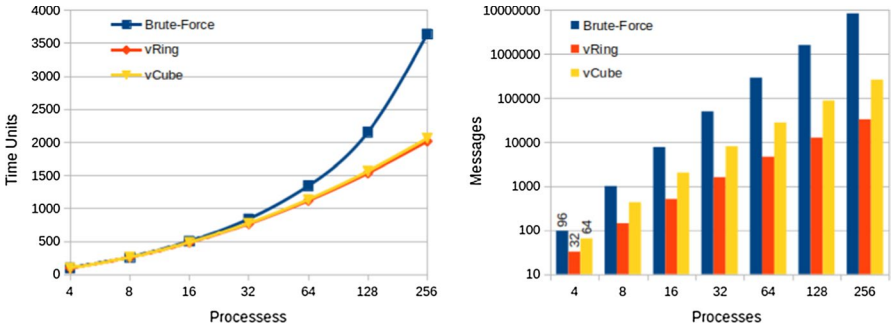
It is important to highlight that depending on the scenario, one or another failure detector may be the best. When the number of processes is small, the Brute-Force algorithm should be the best option, as it presents a latency of a single round, and the cost in terms of the number of tests (n^2) should not be a problem. Whenever it is important to keep the number of tests at the minimum, i.e. requiring the least amount of effort by the processes running the detector, vRing is the best option, as each process is tested exactly once (in case there are no false suspicions). However, vRing presents the largest latency, which means that either detecting failures quickly is not an issue, or an additional event-based notification strategy should be adopted. vCube is the scalable alternative that presents several logarithmic properties and should be used as the size of the system grows. Whenever a quadratic number of tests has to be avoided as well as a linear latency, vCube is the best option.

7.5 Simulation

The algorithms were implemented and compared using the Neko framework [36]. Two scenarios were evaluated: (a) without failures or false suspicions; (b) with *crash* failures. For each scenario, systems with $n = 4, 8, 16, ..., 256$ processes. The time interval since a message is sent until it is received is of 1.0 time unit, of which 0.1 time unit is spent by the source to send the message, and 0.9 time units are for transmission across the network. All experiments were performed in $\log_2^2(n)$ rounds, which is vCube's worst-case latency, in order to guarantee that all processes will be tested by all others. The testing interval for all algorithms was set to 30.0 time units. The timeout is 4.0. Next, the experiment scenarios are described.

No failure. Fig. 9 shows the execution time and the number of messages of the Brute-Force, vCube and the vRing. The execution time varies for the three algorithms, but increases faster for Brute-Force as the number of processes increases. vRing has the same execution time as vCube. The number of messages for Brute-Force is also noticeably higher, as it requires n^2 tests per round. vCube generates $n \log_2 n$ tests per round and vRing just n tests, due to its sequential testing strategy. Each test consists of request and response messages.

Crash. In the scenario with a single failure, the execution time is very close to the scenario without failures, since each execution includes more than one round



(a) Execution Time (u.t.) (b) Number of messages

Fig. 9 Simulation results in a fault-free execution

of tests and the detection latency is diluted across the time. The same happens with the number of messages, although it is slightly lower as the processes detect the failure and stop testing the failed process. Figure 10 shows the latency. A *crash* occurred at time 0.0. As process 0 is the first to be tested by all processes in Brute-Force, the latency is constant and corresponds to the *timeout* interval.

In the first round of vCube, only neighbors virtually connected to the crashed process identify the event. In the second round, neighbors two hops away identify the failure, and so on. The diagnostic latency of the vRing is always $n - 1$ rounds. This result is exactly the same for false suspicion, since information propagation follows the same strategy in each algorithm.

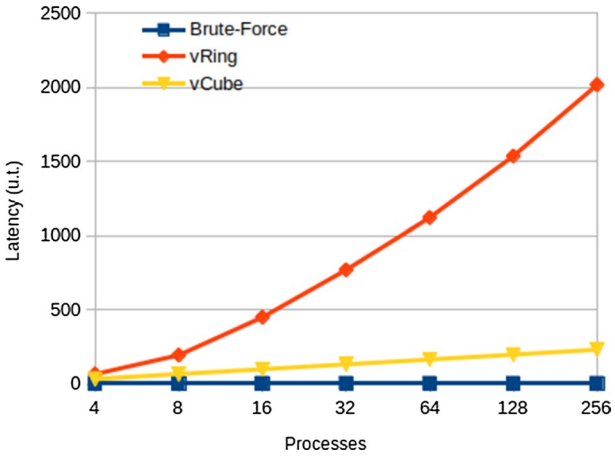


Fig. 10 Latency of diagnosis for the one-faulty process

8 Conclusion

Both distributed diagnosis and failure detection represent abstractions for monitoring processes in distributed systems, with both providing as output a list of processes that are considered to have crashed. The purpose of the proposed model is to bring together the best of both approaches, creating the synergy to foster the development of the next generation of fault monitoring techniques in distributed systems. Several properties are defined and bounds are proved, in particular those related to completeness, accuracy, latency, number of tests, and amount of information transmitted.

In terms of diagnosis, the most significant contribution of the proposed model is to allow a test result to be a mistake, i.e., a correct process can be suspected of having crashed. In addition, the classic properties of failure detectors—completeness and accuracy—were defined for the model and results were derived. From a failure detector perspective, several results were derived for pull-based detectors, including their performance in terms of latency, number of tests, and amount of information transmitted. Three failure detectors were presented and compared: Brute-Force (the traditional all-monitor-all), vRing and vCube. The purpose of the model proposed in this work is thus to bridge the gap between the two abstractions by providing a common framework for defining distributed diagnosis algorithms as unreliable failure detectors.

Future work includes the investigation of other efficient failure detectors based on the proposed model. The model itself can be extended in several ways, for example, to allow the recovery of processes and network partitions. Furthermore, the implementation and empirical evaluation of the proposed detectors based on the quality of service metrics [35] and the adoption of machine learning [37] are also planned as future work. Also planned as applications of the proposed model and algorithms for failure detection in large-scale asynchronous systems, cloud computing [38], and the Internet of Things [39].

Acknowledgements This work was partially supported by the Brazilian Research Council (CNPq—Conselho Nacional de Desenvolvimento Científico e Tecnológico) Grant 308959/2020-5 and FAPESP/MCTIC/CGI Grant 2021/06923-0

Declarations

Conflict of interest The authors hereby ensure that there are no conflicts of interest regarding this manuscript and its publication on Computing. The research/paper is fully compliant with all ethical standards. Elias P. Duarte Jr. is an Associate Editor of the Computing journal.

References

1. NYT: gone in minutes, out for hours: outage shakes facebook (2021) https://www.nytimes.com/2021/10/04/technology/facebook-down.html
2. Codestone: the true impact of IT failures (2017) https://www.codestone.net/our-thoughts/true-impact-of-it-failures

3. Neumann J, Shannon CE, McCarthy J (1956) Probabilistic logics and the synthesis of reliable organisms from unreliable components. Princeton University Press, Princeton, pp 43–98
4. Avizienis A, Laprie J-C, Randell B, Landwehr C (2004) Basic concepts and taxonomy of dependable and secure computing. IEEE Trans Dep Secure Comput 1(1):11–33. https://doi.org/10.1109/TDSC.2004.2
5. Beyer B, Jones C, Petoff J, Murphy NR (2016) Site reliability engineering: how Google runs production systems. O'Reilly, Sebastopol, United States http://landing.google.com/sre/book.html
6. Jha NK (1996) Fault-tolerant computer system design. IEEE Parallel Distrib Technol Syst Appl 4(4):84–84. https://doi.org/10.1109/MPDT.1996.7102341
7. Duarte Jr EP, Santini R, Cohen J (2004) Delivering packets during the routing convergence latency interval through highly connected detours. In: DSN, pp 495–504. https://doi.org/10.1109/DSN.2004.1311919
8. Reynal M (2005) A short introduction to failure detectors for asynchronous distributed systems. SIGACT News 36(1):53–70. https://doi.org/10.1145/1052796.1052806
9. Preparata FP, Metze G, Chien RT (1967) On the connection assignment problem of diagnosable systems. IEEE Trans Electron Comput 16(6):848–854. https://doi.org/10.1109/PGEC.1967.264748
10. Masson GM, Blough DM, Sullivan GF, Pradhan DK (1996) System diagnosis. Prentice-Hall Inc, USA, pp 478–536
11. Duarte EP, Ziwich RP, Albini LCP (2011) A survey of comparison-based system-level diagnosis. ACM Comput Surv. https://doi.org/10.1145/1922649.1922659
12. Fischer MJ, Lynch NA (1985) Impossibility of distributed consensus with one faulty process. J ACM 32(2):374–382. https://doi.org/10.1145/3149.214121
13. Chandra TD, Toueg S (1996) Unreliable failure detectors for reliable distributed systems. J ACM 43(2):225–267. https://doi.org/10.1145/226643.226647
14. Bertier M, Marin O, Sens P (2002) Implementation and performance evaluation of an adaptable failure detector. In: DSN, pp 354–363. https://doi.org/10.1109/DSN.2002.1028920
15. Turchetti RC, Duarte EP, Arantes L, Sens P (2016) A QoS-configurable failure detection service for internet applications. J Internet Serv Appl (JISA) 7(1):1–14. https://doi.org/10.1186/s13174-016-0051-y
16. Turchetti RC, Duarte EP (2017) NFV-FD: implementation of a failure detector using network virtualization technology. Int J Netw Manag 27(6):1988. https://doi.org/10.1002/nem.1988
17. Gupta I, Chandra TD, Goldszmidt GS (2001) On scalable and efficient distributed failure detectors. In: 20th PODCP, ACM, New York, pp 170–179 https://doi.org/10.1145/383962.384010
18. Hakimi SL, Amin AT (1974) Characterization of connection assignment of diagnosable systems. IEEE Trans Comput 23(1):86–88. https://doi.org/10.1109/T-C.1974.223782
19. Hakimi N (1984) On adaptive system diagnosis. IEEE Trans Comput 33(3):234–240. https://doi.org/10.1109/TC.1984.1676420
20. Hosseini, Kuhl, Reddy (1984) A diagnosis algorithm for distributed computing systems with dynamic failure and repair. IEEE Trans Comput 33(3):223–233. https://doi.org/10.1109/TC.1984.1676419
21. Bianchini RP, Buskens RW (1992) Implementation of online distributed system-level diagnosis theory. IEEE Trans Comput 41(5):616–626. https://doi.org/10.1109/12.142688
22. Duarte EP, Nanya T (1998) A hierarchical adaptive distributed system-level diagnosis algorithm. IEEE Trans Comput 47(1):34–45. https://doi.org/10.1109/12.656078
23. Duarte EP, De Bona LCE (2002) A dependable snmp-based tool for distributed network management. In: DSN, IEEE, pp 279–284. https://doi.org/10.1109/DSN.2002.1028911
24. Duarte EP, Bona LCE, Ruoso VK (2014) Vcube: a provably scalable distributed diagnosis algorithm. In: 2014 5th Workshop on latest advances in scalable algorithms for large-scale systems, pp 17–22. https://doi.org/10.1109/ScalA.2014.14
25. Rodrigues LA, Arantes L, Duarte EP (2016) An autonomic majority quorum system. In: 2016 IEEE 30th international conference on advanced information networking and applications (AINA), IEEE, pp 524–531. https://doi.org/10.1109/AINA.2016.73
26. Araujo JP, Arantes L, Duarte EP Jr, Rodrigues LA, Sens P (2019) VCube-PS: a causal broadcast topic-based publish/subscribe system. J Parallel Distrib Comput 125:18–30. https://doi.org/10.1016/j.jpdc.2018.10.011
27. Duarte EP, Weber A, Fonseca KVO (2012) Distributed diagnosis of dynamic events in partitionable arbitrary topology networks. IEEE Trans Parallel Distrib 23(8):1415–1426. https://doi.org/10.1109/TPDS.2011.284

28. Camargo ET, Duarte EP (2018) Running resilient MPI applications on a dynamic group of recommended processes. J Braz Comput Soc 24(1):1–16. https://doi.org/10.1186/s13173-018-0069-z
29. Ziwich RP (2016) A nearly optimal comparison-based diagnosis algorithm for systems of arbitrary topology. IEEE Trans Parallel Distrib 27(11):3131–3143. https://doi.org/10.1109/TPDS.2016.2524004
30. Ziwich RP, Duarte EP, Albini LCP (2005) Distributed integrity checking for systems with replicated data. In: 11th ICPADS'05, vol 1, pp 363–3691. https://doi.org/10.1109/ICPADS.2005.130
31. Song J, Lin L, Huang Y, Hsieh SY (2023) Intermittent fault diagnosis of split-star networks and its applications. IEEE Trans Parallel Distrib Syst 34(4):1253–1264. https://doi.org/10.1109/TPDS.2023.3242089
32. Guo C, Wu C, Xiao Z, Lu J, Liu Z (2023) The intermittent diagnosability for two families of interconnection networks under the PMC model and mm* model. Discret Appl Math 339:89–106. https://doi.org/10.1016/j.dam.2023.05.029
33. Delporte-Gallet C, Fauconnier H, Guerraoui R, Hadzilacos V, Kouznetsov P, Toueg S (2004) The weakest failure detectors to solve certain fundamental problems in distributed computing, ACM, New York, pp. 338–346 https://doi.org/10.1145/1011767.1011818
34. Chandra TD, Hadzilacos V, Toueg S (1996) The weakest failure detector for solving consensus. J ACM 43(4):685–722. https://doi.org/10.1145/234533.234549
35. Chen W, Toueg S, Aguilera MK (2002) On the quality of service of failure detectors. IEEE Trans Comput 51(1):13–32. https://doi.org/10.1109/12.980014
36. Urban P, Defago X, Schiper A (2001) Neko: a single environment to simulate and prototype distributed algorithms. In: 15th ICOIN, pp 503–511. https://doi.org/10.1109/ICOIN.2001.905471
37. Jan SU, Lee YD, Koo IS (2021) A distributed sensor-fault detection and diagnosis framework using machine learning. Inf Sci 547:777–796. https://doi.org/10.1016/j.ins.2020.08.068
38. Bui KT, Van Vo L, Nguyen CM, Pham TV, Tran HC (2020) A fault detection and diagnosis approach for multi-tier application in cloud computing. J Commun Net 22(5):399–414. https://doi.org/10.1109/JCN.2020.000023
39. Zhang W, Lu Q, Yu Q et al (2020) Blockchain-based federated learning for device failure detection in industrial IoT. IEEE Internet Things J 8(7):5926–5937. https://doi.org/10.1109/JIOT.2020.3032544

Authors and Affiliations

Elias P. Duarte Jr.[1] · Luiz A. Rodrigues[2] · Edson T. Camargo[3] · Rogério C. Turchetti[4]

✉ Elias P. Duarte Jr.
elias@inf.ufpr.br

Luiz A. Rodrigues
luiz.rodrigues@unioeste.br

Edson T. Camargo
edson@utfpr.edu.br

Rogério C. Turchetti
turchetti@redes.ufsm.edu.br

[1] Federal University of Paraná, Curitiba, Brazil

[2] Western Paraná State University (UNIOESTE), Cascavel, Brazil

[3] Technological Federal University of Paraná (UTFPR), Toledo, Brazil

[4] Federal University of Santa Maria (UFSM), Santa Maria, Brazil