This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2021.3100680, IEEE Transactions on Dependable and Secure Computing

Optimal Online Liveness Fault Detection for Multilayer Cloud Computing Systems

Yen-Lin Lee
Department of Computer Science and Information Engineering, National Central University,
Taiwan
yenlinlee811109@gmail.com

Deron Liang
Department of Computer Science and Information Engineering, National Central University,
Taiwan
deronliang@gmail.com

Wei-Jen Wang
Department of Computer Science and Information Engineering, National Central University,
Taiwan
wjwang@csie.ncu.edu.tw

***Abstract*—Efficient online liveness fault detection is crucial to cloud systems. Most current online liveness fault detection techniques, such as system layer heartbeating, use a single unreliable detector to detect cloud system liveness. A single unreliable detector requires a certain amount of time to detect faults to avoid misjudgment, regardless of the type of fault detected. However, many faults can be detected by other detectors more quickly. Therefore, this paper proposes an efficient online liveness fault detection mechanism for cloud systems that integrates existing detectors to quickly detect faults. We compared the fault detection efficiency of the proposed mechanism with those of counterpart mechanisms. According to the results, relative to system layer heartbeating without any auxiliary detection mechanism, our proposed mechanism had a 70.3% shorter fault detection time.**

***Index Terms*—Fault detection time, multilayer cloud system, linear layer dependence, online fault detection, transient fault.**

I. Introduction

Fault detection aims at finding major defects in a system, and these defects may appear in the system's components. Fault detection is crucial to ensuring the high availability of systems, and it has been used in a wide variety of critical applications—such as in cloud computing [1], nuclear engineering [2], and aerospace systems [3]. For cloud computing in particular, online fault detection is crucial for ensuring the high availability of cloud systems. Disruptions in the services of cloud computing services could have negative consequences. For example, in November 2020, Amazon Web Services went offline for several hours, resulting in the unavailability of several types of Internet services such as e-commerce and news platforms [4]. Online fault detection for cloud computing is usually used to detect the liveness of the target application and to ensure high availability by reducing downtime; it has been applied in VMWare vSphere [1].

Liveness detection is a common method of online fault detection [5]. An existing fault detector can be classified as reliable or unreliable, per the definition proposed in previous studies [6], [7]. The output of a reliable detector is always accurate; by contrast, an unreliable detector monitors a target component long term, and the response time depends on how long the whole system can tolerably sustain such an operation. In practice, reliable detectors are usually used to detect permanent faults, whereas unreliable detectors, such as heartbeat detectors [9], are usually used to detect transient faults—which are faults of limited duration, caused either by temporary component malfunction or external interference [8]. Note that a transient fault must include a maximum duration parameter; faults that last longer are interpreted as permanent by the recovery algorithm.

In cloud computing, the most common present-day liveness fault detection technique is system layer heartbeating [9]. This technique considers the entire computing system to be a black box. It detects only heartbeats that are regularly received from the target; if no response is received from the target after a user-defined waiting period, an alert is raised. However, fault detection with only the system layer heartbeating technique is inefficient because a single detector cannot distinguish faults in the system—by virtue of the detector's application of the same method to all faults. However, certain faults can be quickly detected by other detectors. For example, when the system power supply is damaged, a power supply detector can quickly detect the fault. By contrast, a detector employing system layer heartbeating can detect faults only after an initial setup time. In other words, the use of other efficient detectors in fault detection can reduce the average fault detection time. We can divide the systems into component groups and install detectors that are most suited to the components in each group. We can then develop an efficient detection strategy that integrates the detection results of each group of detectors.

Several studies [10]–[12] have noted that, a cloud system can provide infrastructure as a service (IaaS), platform as a service (PaaS), or software as a service (SaaS) to end users [38]. They also mentioned that a cloud system comprises numerous components and can be abstractly represented as members of nonoverlapping groups. The most common approach is to group these components into several layers. For example, in the IT industry [32], [39], a cloud system is usually segmented into nine layers (stacks), the functions of which range from networking infrastructure to support for user applications (Fig. 1a). Specifically, an IaaS system comprises four of these nine layers (networking to virtualization), a PaaS system comprises seven of these nine layers (networking to runtime), and an SaaS system comprises all nine layers. Each

1545-5971 (c) 2021 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.
Authorized licensed use limited to: National Central University. Downloaded on September 30,2021 at 04:26:36 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2021.3100680, IEEE Transactions on Dependable and Secure Computing

layer can be further divided conceptually. Studies [33], [37] have provided fine-grained analyses of a virtualized compute host comprising server, virtualization, and operating system (OS) layers (Fig. 1a); the server layer functions as the host hardware and host OS layers (Fig. 1b). On the basis of this idea, a finer-grained view of the server, virtualization, and OS (guest OS) layers is illustrated in Fig. 1c. First, the server layer is divided into the following layers: power component, hardware (CPU), host OS, and network (network service at host OS) layers. Second, the virtualization layer is named the VM process layer. Third, the OS (guest OS) layer is divided into the guest OS and VM network layers.

On the basis of the layering approach, we propose installing a detector for each layer in a fine-grained architecture (Fig. 1c) to accelerate fault detection, as illustrated in Fig. 2. That is, a fast reliable detector can be used if all faults in a particular layer, such as the power layer, are permanent; otherwise, an unreliable detector should be used, such as the host OS layer. As a result, a virtualized compute host in a cloud system can be viewed as a multilayer system, and the liveness of each layer can be detected by either a reliable or an unreliable detector, as illustrated in Table. I. In such an approach, fault detection becomes an online faulty layer identification problem. Notably, we can generalize the approach to different multilayer cloud systems, where we can install a detector on each layer of a multilayer cloud system for rapid fault detection.

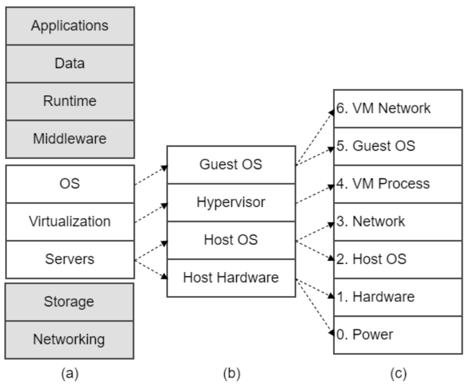


Fig. 1. Different abstractions of the architecture of a virtualized compute host in a cloud system. (a) Three layers (in white) representing a virtualized compute host in the nine layers from the IT industry; (b) corresponding four layers of a virtualized compute host in [33], [37]; (c) finer-grained view of a virtualized compute host which is able to provide rapid fault detection.

To provide an efficient method of identifying the faulty layer, we propose grouping components into linearly dependent layers. Such dependence between layers is a common feature of multilayer systems, as noted by [11], [12], [29], and we term it **linear layer dependence.** Trihinas et al. [11] noted that cloud systems comprise multiple layers and are associated with many service paradigms. They utilized this characteristic of cloud systems to design and implement an automated, layered cloud monitoring framework. Wu et al. [12] also found that a task layer fault is a high layer fault that encapsulates many low layer faults—such as compute

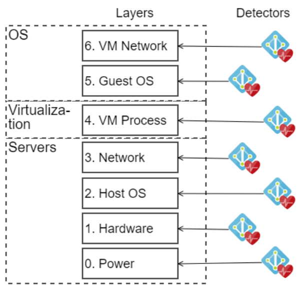


Fig. 2. Layers and their detectors in a virtualized compute host in a cloud system.

TABLE I
LAYER DETECTOR INFORMATION OF FIG.1.

Layer No.	Layer detectors	Objective	Detector types	Response time
6	ICMP for VM	liveness of VM network	unreliable	short
5	Watchdog in VM	guest OS liveness	unreliable	short
4	Libvirt callback func.	existence of VM	reliable	short
3	ICMP for host	network liveness	unreliable	long *
2	Watchdog in host	host OS liveness	unreliable	medium
1	IPMI for CPU	health of CPU	reliable	short
0	IPMI for power	health of power	reliable	short

∗ The network layer detector has a long response time because it must consider transient faults such as network busy.

node and host OS crashes. In summary, online liveness fault detection with linear layer dependence has the following major properties:

- When a fault occurs in a multilayer system, the fault must exist in one of the layers.
- A fault in a lower layer can impair the liveness of all the components in the upper layers; by contrast, a higher layer fault cannot affect the liveness of lower layered components.
- A transient fault in a lower layer can temporarily disrupt the liveness of all components in the upper layers.
- Transient fault detection relies on heartbeating and thus requires a long detection time.
- If the duration of a transient fault exceeds the system's tolerance time, the fault should be interpreted as permanent.
- A permanent fault uses a reliable detector that requires a short response time.

With various layer detectors and linear layer dependence, we can efficiently determine the faulty layer without needing to conduct fault detection for all layers. However, unreliable detectors still take a long time to detect faults. To solve this problem, we propose dividing the detection process used by the unreliable detector into two phases, as illustrated in Fig. 3. The first phase determines whether the fault lies in the identified layer, in which liveness can be detected quickly. The second phase determines whether the fault is transient when the layer is detected to be the faulty layer. This two-phase

1545-5971 (c) 2021 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.
Authorized licensed use limited to: National Central University. Downloaded on September 30,2021 at 04:26:36 UTC from IEEE Xplore. Restrictions apply.

approach decreases the average fault detection time.

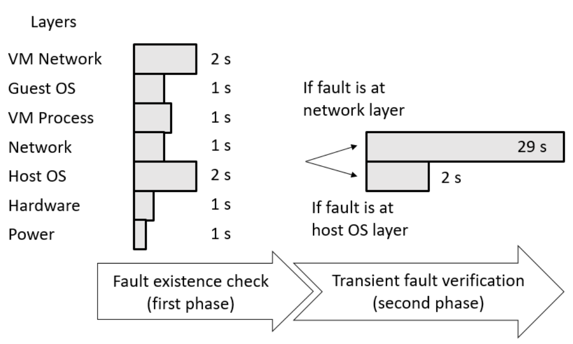

Layers
VM Network
Guest OS
VM Process
Network
Host OS
Hardware
Power
2 s
1 s
1 s
1 s
2 s
1 s
1 s
If fault is at network layer
29 s
2 s
If fault is at host OS layer
Fault existence check (first phase)
Transient fault verification (second phase)

Fig. 3. Use of two-phase fault detection, where the response times are all assumed values.

This study addresses an efficient online liveness fault detection mechanism for multilayer cloud systems, specifically for virtualized compute hosts on the cloud. The proposed mechanism 1) divides a virtualized compute host in a cloud system into linearly dependent layers, 2) integrates several existing detectors for these layers, and 3) uses an optimal tree-based algorithm to quickly detect and identify the faulty layer; the optimal fault detection tree is built through dynamic programming. The proposed mechanism comprises three major steps: 1) confirming fault occurrence, 2) identifying the faulty layer, and 3) confirming fault transience, if required. The contributions of this study are as follows:

1) To determine the faulty layer, we propose an optimal tree-based fault detection algorithm and demonstrate that it maximizes efficiency.
2) We applied the proposed mechanism to a real cloud system. The experimental results demonstrate that, the proposed mechanism has a fault detection time 70.3% shorter than that required for system layer heartbeating [9] without an auxiliary detection mechanism.

The remainder of this paper is organized as follows: Section II introduces the related techniques. Section III describes the multilayer system and the proposed mechanism. Section IV details the proposed fault detection algorithm. Section V presents the experimental results from an application of the proposed mechanism. Section VI discusses the feasibility of applying the proposed mechanism to parallel detection. The final section summarizes this work and suggests future research directions.

II. Related Work

In this section, we first introduce existing online liveness fault detection methods for cloud systems before introducing tree-based fault detection methods that can be used to identify the layer in which a fault occurs. Finally, we present a summary of the properties of the related methods proposed in the literature.

Many current online liveness fault detection methods for cloud computing detect liveness of the software services, physical hosts, VMs or the whole system [1], [9], [17], [24]–[27], [34], [35]. Because these methods are similar to system layer heartbeating, wherein only a single method is used to detect all faults, their detection time is considerably lengthened by faults that require long detection times, such as transient faults. In short, these methods perform inefficiently when detecting certain fault types.

By exploiting linear layer dependence, tree-based methods can identify the faulty layer. Upon reviewing the literature ([15], [16], [18], [23], [28]), we noted that tree-based methods are common in electrical engineering. However, in electrical engineering, diagnostic tests apply a diagnostic tree to diagnose a fault after a system has failed [28], which contrasts with the real-time detection desired for our proposed method. We considered two tree-based methods [18], [23] as examples. Because both are applied after a system has failed, they are not designed for online fault detection or transient fault detection. In other words, directly using these two methods for online fault detection or transient fault detection may result in fault misdiagnosis. The cost of misdiagnosis is high because the fault in the target system not only remains unaddressed but may worsen. In addition, because the relationship between faults is complex (specifically, nonlinear), these two methods are general and heuristic.

In summary, these two methods [18], [23] build two trees that are used for offline detection and are not always optimal. By contrast, our method can be used for optimal online fault detection and transient fault detection, as demonstrated in our results. The trees built by the two aforementioned methods are applicable for online detection only for a limited set of cases, and, only in a subset of those cases, can they achieve the same efficiency as our optimal trees can. The reasons for this are twofold:

- The optimal online detection tree is a special element of the tree set used for online detection.
- The tree set used for online detection is a subset of the trees used for offline detection.

We provide evidence for the aforementioned claim using an example in section V.

A. System Layer Heartbeating

A heartbeat is a periodic signal generated by hardware or software to indicate the liveness of the sender. In general, the sender periodically sends a heartbeat. When the receiver does not receive a heartbeat within a given period (timeout), the sender is determined to have failed [20]. Most liveness fault detection methods for cloud systems use system layer heartbeating, which detects the liveness of the target system by using the heartbeating mechanism. For example, Gokhroo et al. [24] and Villamayor et al. [25] have used system layer heartbeating to detect VM liveness; Yadav et al. [26], Rahman et al. [9], and Zhang et al. [17] have used it to detect physical host and network connection liveness; and Liu et al. [27], Soualhia et al. [34], and Lai et al. [35] have used it to detect cloud service liveness. The length of the timeout for system layer heartbeating is a key parameter. If the timeout period of

1545-5971 (c) 2021 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.
Authorized licensed use limited to: National Central University. Downloaded on September 30,2021 at 04:26:36 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2021.3100680, IEEE Transactions on Dependable and Secure Computing

each heartbeat is too short, the detector misjudges the fault, but if the period is too long, the detection efficiency is low. Accordingly, system layer heartbeating is sensitive to detection time [14]. Therefore, system layer heartbeating [24]–[27], [34], [35] carries a trade-off between accuracy and efficiency. Studies [9], [17] have investigated the optimization of the timeout period of the system layer heartbeating; these studies have demonstrated that detection accuracy can be improved by establishing an adjustable timeout period based on historical data.

VMware vSphere [1] provides an alternative system layer heartbeating approach, which treats the detection target as two independent systems (specifically as hardware and virtualization systems). In particular, VMware vSphere uses two independent system layer heartbeating detectors to detect the liveness of VMs and hosts separately. In VMWare vSphere, a master host monitors the network heartbeats of subordinate hosts every second. When the master host stops receiving these heartbeats from a subordinate host, it checks whether the subordinate host is exchanging heartbeats with one of the datastores to determine whether the fault type is host fault or network isolation. Furthermore, VMware vSphere evaluates whether each VM is running by checking for regular heartbeats and I/O activity from the VM. If no heartbeat or I/O activity is received, VMware vSphere determines that the VM has failed, and the VM is thus rebooted to restore service.

B. Zhao et al. Method

Zhao et al. [18] proposed a fault detection method for permanent faults on hybrid systems and used a printer as an example. In their method, after a fault is detected, the decision tree diagnostics are triggered and executed offline. The recorded data are subsequently analyzed and used for decision tree diagnosis, which identifies no transient faults. In addition, because the relationship between faults is irregular, the aforementioned method is heuristic. The definition of detector cost used by Zhao et al. also differs from ours. Specifically, they considered two types of detectors, built-in sensors and virtual sensors, where built-in sensors have no detection cost but virtual sensors incur additional detection costs.

C. Wang et al. Method

Wang et al. [23] proposed a method for identifying system-level faults. In their method, if there is a fault in the system, the fault must occur prior to fault diagnosis. Therefore, their method is used for offline fault diagnosis and cannot detect transient faults. They also considered faults to be dependent; because these dependencies were complex (i.e., irregular), their method is heuristic.

D. Summary

Table II presents a summary of the properties of methods proposed in the literature. Most existing methods have an architecture based on system layer heartbeating because such an architecture is easy to implement. Such methods use one system-layer heartbeating detector; by contrast, very few methods, such as VMware vSphere, use two independent system-layer heartbeating detectors. The operation of VMware vSphere indicates that fault detection can be more efficient if more layers are used. Tree-based detection methods [18], [23] have been designed for offline hardware fault diagnosis in which the hardware is organized as a multilayer system. These methods cannot be used directly for online liveness detection. By contrast, our proposed mechanism can be used for online liveness detection.

TABLE II
SUMMARY OF THE RELATED WORK

Method	System layer detection		Tree-based detection
	One system	Two systems	Multilayers
Related work	[9], [17], [24]–[27] [34], [35]	[1]	[18], [23]
Application type	Liveness detection for software systems or components	VM liveness detection and host liveness detection	Electronic engineering fault diagnosis
Online liveness detection	Yes	Yes	No
Detection time	30 s to 120 s	VM: 30 to 120 s Host: 13 to 15 s *	N/A
Method overview	[24]–[27], [34], [35] use a user-defined timeout, while [9], [17] use an adjustable timeout	Heartbeat-based method + storage activity verification	Tree-based method

* The detection time is measured based on the system configuration for VMware in [13]. Note that the paper [13] only showed the downtime for each fault cases where downtime is the sum of detection time and recovery time.

III. MINIMIZATION OF MULTILAYER FAULT DETECTION TIME

We focused on minimizing the time required for detecting faults in multilayer systems. In our proposed method, transient faults are identified and then ignored. Thus, nothing happens when our proposed method detects that the target system has recovered from a transient fault. In general, a multilayer system comprises N layers from layer 0 to layer $N-1$. In each layer, a detector can be installed to detect the faults that have occurred in that layer. Given that a fault has occurred, the conditional probability of the fault occurring in layer L_i is P_i, where layers L_i to L_{N-1} are expected to fail because of the fault in L_i. A fault may be transient or permanent. A transient fault in layer L_i can result in temporary failure in layers L_i to L_{N-1}, but the system reverts to a not faulty state after some time. A permanent fault in layer L_i can result in permanent failure in layers L_i to L_{N-1}. To detect a fault, the detection mechanism can ask a layer detector to conduct fault detection, and a detector in layer L_i requires T_i seconds to complete detection and return the result. Therefore, the problem of online liveness fault detection for multilayer cloud computing systems is defined to find whether a permanent fault

1545-5971 (c) 2021 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.
Authorized licensed use limited to: National Central University. Downloaded on September 30,2021 at 04:26:36 UTC from IEEE Xplore. Restrictions apply.

exists and to find the faulty layer of the system. To simplify the problem, we make two assumptions. First, we assume that detectors always return correct results because our focus is rapid liveness detection rather than Byzantine failure. Second, we assume that no other faults occur between the time when a fault occurs to the time when the recovery process ends. The symbols used in this paper are defined in Table III.

TABLE III
DEFINITIONS OF SYMBOLS

Symbol	Description
N	Assume that there are N layers
L_i	The ith layer, $(0 \leqq i \leqq N-1)$
F_i	The fault of L_i, $(0 \leqq i \leqq N-1)$
D_i	The detector for L_i , $(0 \leqq i \leqq N-1)$
T_i	The response time of D_i , $(0 \leqq i \leqq N-1)$
P_i	The conditional probability of F_i (fault percentage), $(0 \leqq i \leqq N-1)$

Based on the problem defined above, we propose a fault detection mechanism that efficiently detects permanent faults in a multilayer system. According to Alwi et al. [21], when a fault occurs in a system, the main problems to be addressed are threefold: raising the alarm, accurately diagnosing the fault, and deciding how to handle the fault. Similar to our mechanism, their fault detection mechanism proceeds according to the first two of the following steps; however, our mechanism adds a transient fault confirmation step, as illustrated in Fig. 4.

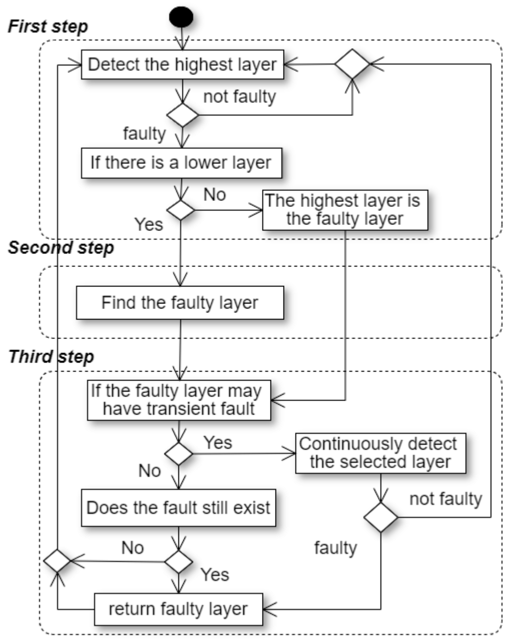

First step
Detect the highest layer
not faulty
faulty
If there is a lower layer
No
The highest layer is the faulty layer
Yes
Second step
Find the faulty layer
Third step
If the faulty layer may have transient fault
Yes
Continuously detect the selected layer
No
not faulty
Does the fault still exist
No
faulty
Yes
return faulty layer

Fig. 4. Fault detection mechanism.

- First step: In this step, we must determine whether a fault has occurred. This step can be achieved through continuous detection in the highest layer, L_{N-1}. After a fault is detected, if a lower layer exists, we perform the second step; otherwise, we perform the third step.
- Second step: In this step, a fault detection algorithm can be utilized to find the faulty layer. Note that the fault detection algorithm affects the efficiency of fault detection.
- Third step: In this step, the aim is to determine whether the fault is a permanent fault and to return the result. Note that for a layer with a possible transient fault, more time is required for detecting transient faults; for other layers, one needs only to determine whether the fault still exists. Here, we can use a highly rapid fault detection approach to verify the existence of the fault; for example, a rapid ping can be used to verify the liveness of the system. If the detector of the faulty layer does not respond within the user-defined waiting period, the fault is considered a permanent fault. If the fault is permanent, the faulty layer is returned as a result.

The goal of the fault detection mechanism is to detect permanent faults in the target system. However, transient faults may occur and obfuscate detection. Because a transient fault occurs for a limited period, when it disappears, the system state changes from faulty to not faulty, and the fault detection algorithm in the second step may identify the wrong faulty layer. Five events are crucial to fault detection:

- tfo: a transient fault occurs
- tfd: the transient fault disappears
- fss: the first step starts
- sss: the second step starts
- tss: the third step starts

Transient faults must occur first to trigger fault detection, and the corresponding partial event order is tfo $\rightarrow$ fss $\rightarrow$ tfd. The partial event order of fault detection is fss $\rightarrow$ sss $\rightarrow$ tss. In addition, because we assume that only one fault occurs before the fault recovery phase, we need not consider the problem of a second fault, which may cause the fault confirmation to fail. To ensure that all transient faults can be identified by the fault detection mechanism, we analyze all possible scenarios. On the basis of the two partial orders, three possible scenarios can be established as follows:

- first scenario (Fig. 5): tfo $\rightarrow$ fss $\rightarrow$ tfd $\rightarrow$ sss $\rightarrow$ tss
- second scenario (Fig. 7): tfo $\rightarrow$ fss $\rightarrow$ sss $\rightarrow$ tfd $\rightarrow$ tss
- third scenario (Fig. 9): tfo $\rightarrow$ fss $\rightarrow$ sss $\rightarrow$ tss $\rightarrow$ tfd

In the following subsections, we discuss the accuracy of the proposed fault detection mechanism for the three scenarios. The proposed mechanism must return "not faulty" for the three scenarios.

A. First Scenario

This scenario involves two cases:

1) For the first case, the transient fault may not be found in the first step of the proposed mechanism; this occurs when the fault has disappeared before the highest layer is detected in the first step. Accordingly, in this case, the proposed mechanism returns "not faulty" and returns to the fault detection routine.

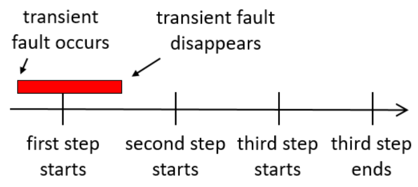


Fig. 5. First scenario of transient fault in fault detection.

2) For the second case, the fault is identified in the first step. In this case, the transient fault is expected to disappear before the start of the second step, according to the description for this scenario. Consequently, in the second step, the mechanism is expected to identify the presence of the fault at the highest layer (Fig. 6). Subsequently, in the third step, the mechanism is expected to verify the existence of the fault or continuously detect the liveness of the highest layer. After verifying the existence of the fault with any of the two verification actions, the mechanism returns "not faulty" because the transient fault has disappeared.

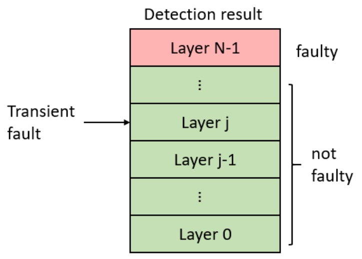


Fig. 6. Detection results of each layer detector in the second case of the first scenario.

B. Second Scenario

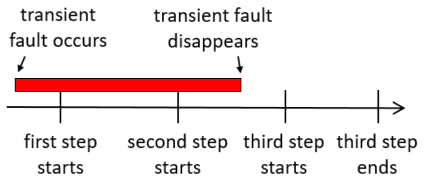


Fig. 7. Second scenario of transient fault in fault detection.

The transient fault is denoted as F_j. The proposed mechanism always returns "faulty" in the first step and begins finding the faulty layer of the system in the second step. The second scenario also involves two cases.

1) For the first case, the transient fault F_j may disappear before detector D_j performs fault detection, and detector D_j must return "not faulty." However, other layers higher than layer L_j may have already been detected as faulty. Consequently, in the second step, the mechanism returns a wrong faulty layer number based on the linear layer dependence. That is, any layer higher than layer L_j could be erroneously identified as the faulty layer, as illustrated in Fig. 8. Subsequently, in the third step, the mechanism verifies the existence of the fault and then returns "not faulty" because the transient fault has disappeared in the second step.

2) For the second case, the transient fault F_j has been detected by detector D_j. In the second step, the mechanism returns the faulty layer L_j based on the linear layer dependence. In the third step, the mechanism then continuously detects the liveness of the selected layer L_j. Because the transient fault has disappeared, in the third step, the mechanism returns "not faulty."

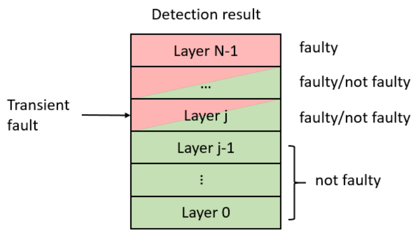


Fig. 8. Possible detection results from each layer's detector in the second scenario.

C. Third Scenario

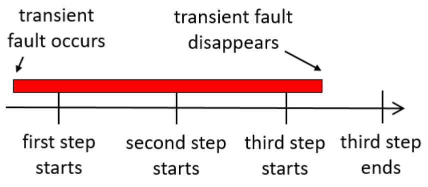


Fig. 9. Third scenario of transient fault in fault detection.

The third scenario is similar to the second case of the second scenario. The first and second steps involve the same behaviour as those observed in the first two scenarios. Therefore, in the second step, the mechanism returns the faulty layer L_j, as presented in Fig. 10. Subsequently, in the third step, the mechanism continuously detects the liveness of the selected layer L_j during the user-defined waiting period. Because the duration of the transient fault F_j must be less than the user-defined waiting period for F_j, the mechanism in the third step eventually returns "not faulty."

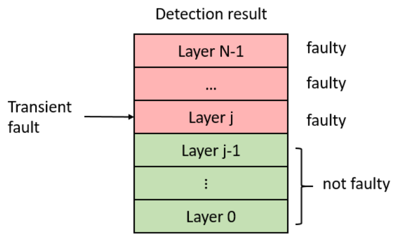


Fig. 10. Possible detection results of each layer's detector in the third scenario.

Authorized licensed use limited to: National Central University. Downloaded on September 30,2021 at 04:26:36 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2021.3100680, IEEE Transactions on Dependable and Secure Computing

Per the preceding discussion, the fault detection mechanism can always identify transient faults. The third step, which is based on the faulty layer, involves either detecting the faulty layer within the user-defined period or detecting a layer that is 1) higher than or equal to the actual faulty layer and 2) requires a shorter response time.

Because the fault detection mechanism is general, it can be applied to other liveness fault detection mechanisms. For example, because the system layer heartbeating method conceives of the entire computing system as a single layer, the method comprises the first and third steps. Our proposed mechanism, by contrast, can utilize various fast detectors to determine the faulty layer in the second step; it thus comprises all three steps. To efficiently identify the faulty layer in the second step, we designed a fault detection algorithm for a multilayer system, and it is detailed in the next section.

IV. Proposed Fault Detection Algorithms

In this section, we describe a proposed fault detection algorithm, the binary search tree algorithm, and a naive algorithm. Note that the naive algorithm only considers linear layer dependence, and the proposed algorithm considers linear layer dependence, T_i, and P_i. Although the proposed algorithm takes time to build a binary search tree, the binary search tree can be used indefinitely until the to-be-detected system changes.

The steps of the naive algorithm are as follows:

- Step 1: After observing F_{N-1} in the highest layer $N-1$, execute the detection method from D_{N-2} to D_0 until its result is FALSE (TRUE represents a numerical anomaly), and then name the last detector D_k.
- Step 2: If every detector returns TRUE, then the faulty layer is layer L_0; otherwise, the faulty layer is L_{k+1}.

A. *Binary Search Tree Algorithm*

Given an N-layer system, a fault detection mechanism can ask any layer detector to conduct fault detection on that layer. When a detector in layer L_i detects a fault, the fault detection mechanism can skip the detection for layers higher than L_i because of linear layer dependence, as illustrated in Fig. 11. This is a typical problem to which the binary search algorithm can be applied. However, because each layer has its own response time T_i and conditional fault probability P_i, the self-balancing binary search [36] does not always identify faults efficiently. For example, as displayed in Fig. 16, the optimal tree in the second step is a rightist tree rather than a balanced tree. To remedy this problem, we propose a new algorithm based on a binary search tree that considers T_i and P_i, thereby minimizing fault detection time. Therefore, the proposed algorithm necessarily outperforms other types of binary search tree algorithms when considering T_i and P_i. Some crucial concepts are defined as follows.

Definition 1. *$ADT(a,b,c)$ is a symbol representing the average detection time (ADT) of a tree comprising D_a to D_b, with root D_c.*

Definition 2. *$T(a,b)$ is a symbol representing the ADT of the optimal tree; the optimal tree is the tree with the smallest ADT among all possible trees and is composed of D_a to D_b.*

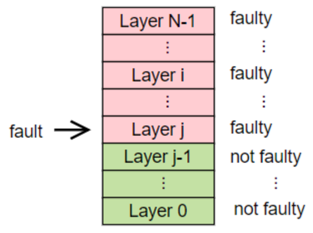


Fig. 11. State of all layers when a fault occurs at layer j.

The binary search tree with the smallest ADT for identifying the faulty layer can be found by building all possible binary search trees and comparing their ADTs. The equations used to calculate the ADT of the binary search trees and identify the optimal tree are described as follows.

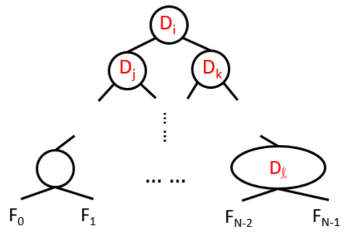


Fig. 12. Proposed binary search tree.

Fig. 12 presents an example of a binary search tree. In Fig. 12, fault detection at the root of the tree (D_i) should be performed regardless of what type of fault occurs. D_j or D_k is then queried respectively if the first result is TRUE or FALSE; TRUE means that a component in this layer does not respond. Therefore, the invocation probability of D_j is equal to the sum of P_0 to P_i divided by the sum of the conditional probability of all faults, and the invocation probability of D_k is equal to the sum of P_{i+1} to P_{N-1} divided by the same denominator. In other words, the invocation probability of each node in the binary search tree is related to the range of its subtrees and the conditional probability of each fault. The ADT of this binary search tree (Fig. 12) can be denoted as $ADT(0,N-2,i)$, which can be calculated as follows:

$$\begin{aligned} ADT(0,N-2,i) = T_i + \frac{\sum_{m=0}^{i} P_m}{\sum_{m=0}^{N-1} P_m} * T_j \\ + \frac{\sum_{m=i+1}^{N-1} P_m}{\sum_{m=0}^{N-1} P_m} * T_k + \cdots \end{aligned} \tag{1}$$

Subsequently, (1) can be reorganized using the concept of the subtrees, as follows:

1545-5971 (c) 2021 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.
Authorized licensed use limited to: National Central University. Downloaded on September 30,2021 at 04:26:36 UTC from IEEE Xplore. Restrictions apply.

$$\begin{aligned} ADT(0,N-2,i) = T_i + \frac{\Sigma_{m=0}^{i} P_m}{\Sigma_{m=0}^{N-1} P_m} ADT(0,i,j) \\ + \frac{\Sigma_{m=i+1}^{N-1} P_m}{\Sigma_{m=0}^{N-1} P_m} ADT(i+1,N-2,k) \end{aligned} \tag{2}$$

$ADT(0,i,j)$ is the ADT of the left subtree, and $ADT(i+1,N-2,k)$ is the ADT of the right subtree. Equation (2) reveals that if we try each candidate root i to determine which detector D_i to use as the root of an optimal binary search tree comprising detectors D_a to D_b, where $a \leq i \leq b$, then we are guaranteed to find the optimal binary search tree. Therefore, the ADT of the optimal tree, $T(a,b)$, can be calculated as follows:

$$\begin{aligned} T(a,b) = \min_{i=a\sim b}[T_i + \frac{\Sigma_{m=a}^{i} P_m}{\Sigma_{m=a}^{b+1} P_m} T(a,i-1) \\ + \frac{\Sigma_{m=i+1}^{b+1} P_m}{\Sigma_{m=a}^{b+1} P_m} T(i+1,b)],\ if\ a \leq b \end{aligned} \tag{3}$$

Because a binary search tree where $a > b$ does not exist, (3) can only be used when $a \leq b$.

B. Proof of Optimality of Proposed Algorithm

We now prove that the binary tree built by the proposed method is optimal. Let S be the set of trees containing all binary trees with k detectors D_a to D_b, where $b = a + k - 1$ and $k \geq 1$. Then (3) can find the tree with the smallest ADT in S. This proof is based on the following lemma.

Lemma 1. *If a tree X ($X \in S$) is found by (3), then there is no tree Y ($Y \in S$) such that ADT of Y < ADT of X.*

Proof. A binary tree can be divided into three parts: the root, left subtree, and right subtree. Equation (3) reveals that it uses each possible D_i as the root respectively, where $a \leq i \leq b$, to find the optimal tree. To be precise, (3) traverses all possible trees in S to find the optimal tree. Therefore, the ADT of X is equal to the minimal ADT among the elements in S. That is, the ADT of each element in S cannot be smaller than the ADT of X. Because the existence of Y contradicts these facts, Y does not exist; thus, the proof is complete. □

C. Binary Search Tree Algorithm Based on Dynamic Programming

According to (3), the problem of building an optimal binary search tree is a typical optimization problem, and all the conditions must be met before dynamic programming can be applied. We now present an example to illustrate the bottom-up approach of dynamic programming.

This example features a five-layer system, L_0 to L_4, where D_0 to D_3 are used to build a binary search tree. The computation for finding an optimal binary search tree $T(0,3)$ is presented in Fig. 13. A table for recording the optimal results of the subproblems should be filled from left to right and then from bottom to top, as illustrated in Fig. 14.

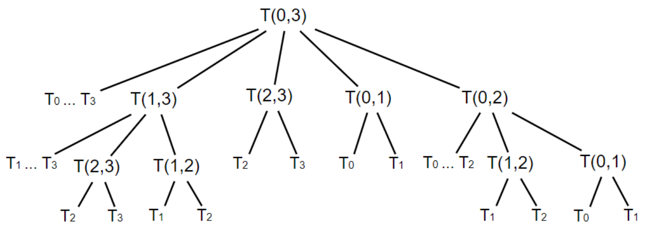


Fig. 13. Recursion tree for computation of $T(0,3)$.

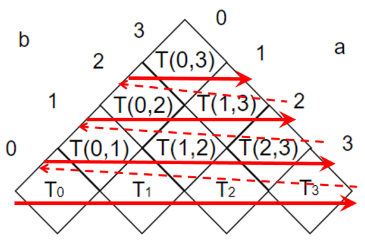


Fig. 14. Table for recording $T(a,b)$; the table is rotated so that the diagonals run horizontally.

After detailing the concept underlying the application of dynamic programming, we explain the algorithm for finding the optimal binary search tree (Algorithm 1) as follows.

Algorithm 1 *tree_building*

Input: detector response time list $dtime\_list$, conditional fault probability list p_list

Output: $<$ ADT $tree\_time$, Binary Search Tree $tree$ $>$

```
1:  set a global variable $time\_list$
2:  set a global variable $tree\_list$
3:  set a global variable $probability\_list$ = $p\_list$
4:  $list\_length$ = length of $dtime\_list$
5:  for $index = 0$ to $list\_length$ -1 do
6:      $time\_list[index][index]$ = $dtime\_list[index]$
7:      $tree\_list[index][index]$ = $[index]$
8:  end for
9:  for $size = 2$ to $list\_length$ do
10:     for $x = 0$ to $list\_length$-size do
11:         y = x+size-1
12:         $op\_time$, $op\_tree$ = $find\_optimal\_subtree$(x, y)
13:         $time\_list[x][y]$ = $op\_time$
14:         $tree\_list[x][y]$ = $op\_tree$
15:     end for
16: end for
17: $tree\_time$ = $time\_list[0][list\_length-1]$
18: tree = $tree\_list[0][list\_length-1]$
19: return $<tree\_time, tree>$
```

We use Algorithm 1 to find the optimal binary search tree structure and its ADT through dynamic programming and Algorithm 2. In Algorithm 1, the input variables $dtime\_list$ and p_list are lists, where $dtime\_list$ stores the response time of the detectors from T_0 to T_{N-2} and p_list stores the conditional probability of the faults from P_0 to P_{N-1}.

Authorized licensed use limited to: National Central University. Downloaded on September 30,2021 at 04:26:36 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2021.3100680, IEEE Transactions on Dependable and Secure Computing

Algorithm 2 $find\_optimal\_subtree$

```
Input: beginning layer number $i$, end layer number $j$
Output: $<$ ADT $op\_time$, Binary Search Tree $op\_tree$
1: $op\_time$ = maximum number
2: for $root = i$ to $j$ do
3:     if (no left subtree) then
4:         $ltree\_time$ = 0
5:     else
6:         $ltree\_time$ = $time\_list[i][root-1]$
7:     end if
8:     if (no right subtree) then
9:         $rtree\_time$ = 0
10:    else
11:        $rtree\_time$ = $time\_list[root+1][j]$
12:    end if
13:    for $ind = i$ to $root$ do
14:        $left\_probability$ += $probability\_list[ind]$
15:    end for
16:    for $ind = root+1$ to $j+1$ do
17:        $right\_probability$ += $probability\_list[ind]$
18:    end for
19:    $p$ = $left\_probability$ + $right\_probability$
20:    $ltree\_p$ = $left\_probability \div p$
21:    $rtree\_p$ = $left\_probability \div p$
22:    time = $time\_list[root][root]$ + $ltree\_p$ * $ltree\_time$
       + $rtree\_p$ * $rtree\_time$
23:    if (time $< op\_time$) then
24:        $op\_time$ = time
25:        record the tree structure into $op\_tree$
26:    end if
27: end for
28: return $< op\_time, op\_tree >$
```

Lines 1–3 of the algorithm declare three global variables, $time\_list$, $tree\_list$, and $probability\_list$, which are used in both Algorithm 1 and Algorithm 2. The variable $time\_list$ is an $(N-1) \times (N-1)$ matrix that stores the ADTs of subtrees, and the variable $tree\_list$ is an $(N-1) \times (N-1)$ matrix that stores the structures of subtrees. Note that $time\_list[m][n]$ is used to store the ADT of the subtree comprising detectors D_m to D_n, and $tree\_list[m][n]$ is used to store the structure of the subtree comprising detectors D_m to D_n. The **for** loop of lines 5–8 initializes the values of $time\_list[index][index]$ and $tree\_list[index][index]$. The **for** loop of lines 9–16 then uses Algorithm 2 to compute $time\_list[x][y]$ and $tree\_list[x][y]$ for all $0 \leq x < y \leq (N-2)$. In the first iteration, when *size* = 2, the loop computes $time\_list[x][x+1]$ and $tree\_list[x][x+1]$ for $x = 0, 1, ..., (N-3)$. The second iteration, with *size* = 3, computes $time\_list[x][x+2]$ and $tree\_list[x][x+2]$ for $x = 0, 1, ..., (N-4)$, and so forth. Finally, Algorithm 1 returns the ADT and structure of the optimal binary search tree comprising detectors D_0 to D_{N-2}.

Algorithm 2 is a subfunction of Algorithm 1. In Algorithm 2, the input variables i and j are integers, which represent an optimal binary search tree to be found that comprises detectors D_i to D_j. Equation (3) demonstrates that the **for** loop of lines 2–27 tries each candidate index *root* to determine which detector D_{root} to use as the root of the optimal binary search tree. In lines 3–7 and lines 8–12, the procedure yields the ADTs of the optimal left subtree and optimal right subtree, respectively. In lines 13–21, the procedure computes the invocation probabilities of the left and right subtrees. Subsequently, the procedure computes the ADT in line 22, based on (2). In lines 23–26, as long as the procedure finds a more optimal detector D_{root} to use as the root, it saves the current ADT and tree structure in op_time and op_tree, respectively. Finally, Algorithm 2 returns a data structure comprising op_time and op_tree after the loop ends.

As evident in the preceding discussion, an optimal binary search tree can be built by the proposed algorithm. The time complexity of a naive and optimal binary tree is $O(n^2)$ and $O(log\ n)$, respectively. Therefore, the proposed algorithm is more efficient.

V. Performance Evaluation in a Multilayer Cloud Computing System

In this section, we test the performance of our proposed mechanism by applying it on a real cloud computing system (OpenStack). We implemented a fault detection system based on the proposed mechanism and algorithms (Fig. 15). This cloud computing system comprises a detection machine along with several virtualized compute hosts (in the compute pool) to be detected. The machine specifications are presented in Table IV. The fault detection system operates on the detection machine and can query all layer detectors. Based on the liveness of each layer, all layer detectors can return only TRUE or FALSE, where TRUE indicates that the layer is faulty and FALSE indicates that the layer is not faulty. The liveness of each layer is determined from the perspective of the user; that is, the layer is not faulty only when the user can recognize that the layer is alive. A compute host to be detected can be abstracted as comprising a host part and a VM part. The host part comprises four layers, namely, the power, hardware, host OS, and network layers, of which the host OS layer and the network layer have medium and long transient faults respectively. The VM part comprises three layers as presented in Table V: the VM process, guest OS, and VM network layers, of which the guest OS layer and the VM network layer have short transient faults.

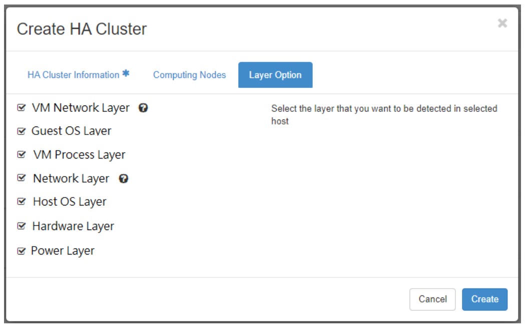
TABLE IV
Machine Information

Role	Machine type	Operating system
Detection machine	ASUS MD790	Ubuntu 16.04
Compute hosts	ASUS MD790	Ubuntu 16.04

With regard to the channel for querying the detector, the detectors for the VM network, guest OS, and VM process layers return results over the network because the user also controls the VM through the network. The detectors for the host OS, hardware, and power layers return results via the

1545-5971 (c) 2021 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.
Authorized licensed use limited to: National Central University. Downloaded on September 30,2021 at 04:26:36 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2021.3100680, IEEE Transactions on Dependable and Secure Computing



☑ VM Network Layer
☑ Guest OS Layer
☑ VM Process Layer
☑ Network Layer
☑ Host OS Layer
☑ Hardware Layer
☑ Power Layer

Select the layer that you want to be detected in selected host

Fig. 15. The user interface to enable layer detectors for the proposed mechanism on OpenStack.

intelligent platform management interface (IPMI) channel. Because the response time of the VM network, guest OS, and host OS layer detectors are relatively short, these detectors can perform complete detection. Therefore, in this case, only detection by an unreliable detector in the network layer is divided into two phases. The user-defined waiting period for the general network layer detector is 30 s. In the proposed mechanism, we set the fault detection timeout period to 1 s (in the second step) and the transient fault detection timeout period to 29 s (in the third step). In addition, we assume that the fault detection system can query only one layer detector at a time. We made this assumption considering the fact that many layer detectors share the same channel, such as the IPMI [13].

TABLE V
DESCRIPTION OF EACH LAYER

Layer	Detector	Faults
VM Network	ICMP query	Permanent and transient faults
Guest OS	Watchdog in VM	Permanent and transient faults
VM Process	Software detector based on Libvirt	Only permanent faults
Network	ICMP	Permanent and transient faults
Host OS	Watchdog in host	Permanent and transient faults
Hardware	IPMI	Only permanent faults
Power	IPMI	Only permanent faults

We then compared the performance of two detectors—one with no auxiliary detection mechanism (system layer heartbeating) and one with an additional detection mechanism (our proposed mechanism)—in handling faults that we injected into a virtualized compute host in the cloud computing system.

We now demonstrate that our proposed mechanism performed well in the cloud computing system. To construct the binary search tree, we used data from Lu's report [22] on the number of outages and outage types of two clusters: Platinum and Titan (Table VI). Because the data in Table VI cover only three layers and because our virtualized compute host has seven layers, we mapped the software layer described in Table VI to the host OS, network, VM, guest OS, and VM network layers in our system. To estimate the fault percentages in the virtualized compute host, we used the average fault percentages listed in Table VI; we assumed the fault percentages of the five layers to be equal. Therefore, the percentages of the faults for the five layers should be $80.3\%/5 = 16.1\%$. The hardware and the power layers are unchanged, and their fault percentages should thus be 2.6% and 17.2%, respectively. The fault percentages P_i are listed in Table VII. The table also lists the response time T_i for each layer detector D_i. In this case study, the data in Table VII were used to construct the proposed binary search tree for fault detection in the second step of the proposed mechanism, which is illustrated in Fig. 16.

To evaluate the performance of the proposed mechanism, we injected a fault into each layer of the virtualized compute host and measured the time from fault injection to fault detection. Each injected fault for a particular layer is defined as a fault case in this experiments. We repeated each fault case 10 times and calculated the corresponding average fault-case detection time, as shown in Table VIII. The fault cases with the corresponding injection methods used in this study are listed as follows:

- VM network: disable the VM network interface.
- Guest OS: crash the guest OS kernel.
- VM process: kill the VM process.
- Network: disable the host network interface.
- OS: crash the host OS kernel.
- Hardware: simulate high CPU temperature by injecting error values into the detector and immediately crash the host OS.
- Power: power off the host.

The experimental results obtained for the fault cases are presented in Table VIII. Notably, the fault detection times for both the network layer fault and the hardware layer fault were relatively long. The average fault-case detection time for the network layer fault was long because the network fault might have been transient (e.g., due to a busy network); thus, the proposed mechanism required 29 s in the third step to distinguish whether the fault was transient. In addition, the average fault-case detection time for the hardware layer was long because the proposed mechanism sequentially queried multiple detectors (D_6, D_0, D_3, D_2, and D_1) to identify that the fault was at the hardware layer (Fig. 16).

On the basis of the fault type percentage (P_i) in Table VII and the average fault-case detection times in Table VIII, the ADT of the proposed mechanism from the experiment was 8.91 s, which is 70.3% faster than the detection time (30 s) of the system layer heartbeating approach [1], [30], [31]. The ADT from experiments was slightly lower than the theoretical ADT, which was calculated to be 9.4 s. This was possibly because a detector might have returned the detection result immediately when the layer was healthy. For example, the detector of the VM network layer takes less than 0.1 s to

1545-5971 (c) 2021 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.
Authorized licensed use limited to: National Central University. Downloaded on September 30,2021 at 04:26:36 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2021.3100680, IEEE Transactions on Dependable and Secure Computing

obtain the detection result if the network layer is healthy (not faulty).

TABLE VI
PERCENTAGE OF SYSTEM OUTAGE TYPES WITH NO CONSIDERATION OF SYSTEM MAINTENANCE IN TWO CLUSTERS: PLATINUM AND TITAN [22].

	Software(%)	Hardware(%)	Power(%)
Platinum	99.9	0.1	0
Titan	60.6	5.1	34.3
Average	80.3	2.6	17.2

TABLE VII
INFORMATION FOR EACH LAYER OF A VIRTUALIZED COMPUTE HOST IN THE CLOUD SYSTEM

Layer i	Layer name	T_i (s)	P_i (%)
6	VM Network	2.0	16.1
5	Guest OS	1.0	16.1
4	VM Process	1.1	16.1
3	Network	0.53	16.1
2	Host OS	3.45	16.1
1	Hardware	0.81	2.6
0	Power	0.06	17.2

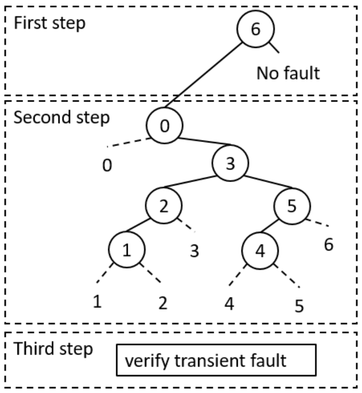


Fig. 16. Proposed mechanism for case study.

TABLE VIII
AVERAGE FAULT-CASE DETECTION TIME (IN SECONDS) FOR EACH FAULT CASE (ED: EXPERIMENTAL DATA; TD: THEORETICAL DATA; NET: NETWORK)

	Power	HW	Host OS	Net	VM process	Guest OS	VM Net
ED	1.64	5.96	6.28	36.28	3.68	3.68	2.68
TD	2.06	6.85	6.85	35.04	4.69	4.69	3.59

A. Comparison with Zhao et al. Method

The decision tree built by the Zhao et al. method is shown in Fig. 17. Using their detector definitions, we treat detectors in the power and hardware layers as built-in sensors, and we treat the other detectors as virtual sensors. According to Fig. 17, five detectors are required to accurately determine that there are no faults in the target system. If the target system fails during VM network detection, any fault is identified as a VM network fault; this means that the Zhao et al. method cannot be used for online fault detection.

To apply the Zhao et al. method to online fault detection, we must first determine that a fault has occurred, which can be achieved by performing detection on the highest layer. The new corresponding decision tree is presented in Fig. 18. This method, however, still cannot detect transient faults. For example, if a transient fault occurs, the detection system considers the target system to be faulty. This is an incorrect judgment because the target system should be judged as healthy after the transient fault has disappeared.

Therefore, in the experiment, we assumed that all faults were permanent. In the experiment, the ADT of the Zhao et al. method was 4.58 seconds.

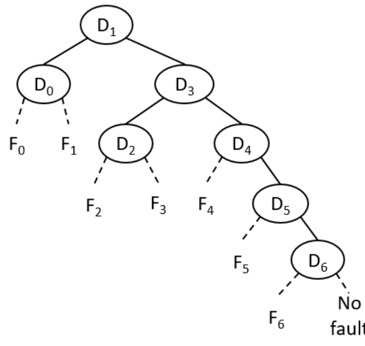


Fig. 17. Decision tree from Zhao et al. method.

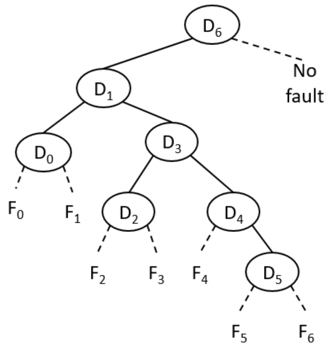


Fig. 18. New decision tree.

B. Comparison with Wang et al. Method

The fault diagnostic tree built by the Wang et al. method is presented in Fig. 19. According to the fault diagnostic tree, two detectors are required to determine that there are no faults in the target system. As with the Zhao et al. method and for the same reasons, the Wang et al. method cannot be used for online fault detection.

To apply the Wang et al. method for online fault detection, we must first determine that a fault has occurred, which can be achieved by performing detection on the highest layer. The

1545-5971 (c) 2021 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.
Authorized licensed use limited to: National Central University. Downloaded on September 30,2021 at 04:26:36 UTC from IEEE Xplore. Restrictions apply.

new corresponding fault diagnostic tree is presented in Fig. 20. However, as is the case with the Zhao et al. method and for the same reasons, the Wang et al. method still cannot detect transient faults.

Therefore, in the experiment, we assumed all faults to be permanent. In the experiment, the ADT of the Wang et al. method was 6.17 seconds.

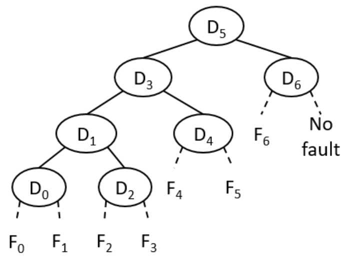


Fig. 19. Fault diagnostic tree from the Wang et al. method.

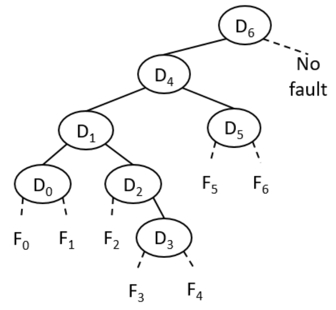


Fig. 20. New fault diagnostic tree.

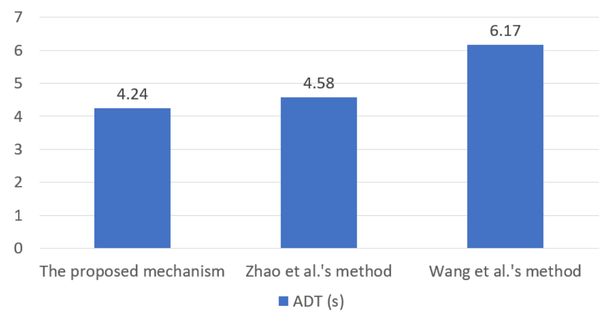


Fig. 21. Comparison of our proposed mechanism with counterpart methods with respect to performance in the absence of transient faults.

The proposed mechanism can save time in cases where transient fault identification is relevant. Specifically, in the experiments, the ADT of the proposed mechanism was 4.24 seconds when there were no transient faults, outperforming its counterparts (Fig. 21). Furthermore, the detection time of the proposed mechanism was 70.3% shorter than that required by system layer heartbeating without any auxiliary detection mechanism.

C. Influence of Network Fault

The experimental results demonstrated that network faults greatly affect the ADT of the proposed mechanism; this is indicated by the finding that the detection time for the network fault was at least five times as long as those for other fault cases. This is because the network layer detector must consider the transient fault problem (such as packet loss or a busy network). On the basis of the default fault detection time of existing high-availability cloud systems (such as VMware vSphere HA [1], HAProxy [30], and Pacemaker [31]), we set the maximum duration of network layer faults to 30 s. However, the maximum duration of network faults is tunable. The value could be as high as 120 s in an unreliable network environment, according to the operation of VMware vSphere HA. When we increased the maximum duration of network layer faults to 120 s in our mechanism, the new theoretical ADT based on Table VII became 23.85 s, which was more than twice as long as the original theoretical ADT. In scenarios where the system is running in a highly reliable network environment, the maximum duration of network faults can be reduced further. If we assume that the value could be decreased to 10 s, the new theoretical ADT would be 6.14 s, which is 65% of the original theoretical ADT. In conclusion, the ADT of the proposed fault detection mechanism can be very short only if the cloud system has a highly reliable networking infrastructure. In either case, the proposed mechanism is expected to outperform the system layer heartbeating approach, especially in an unreliable networking environment (23.85 vs. 120 s, with the assumption of the fault percentages in Table VII).

VI. Extension to Parallel Detection

In this section, we discuss the application of our proposed mechanism to the case of parallel detection. First, we explain why direct parallel detection cannot be used in many cases and then present how one ought to use our proposed mechanism in parallel detection. Second, we evaluated the performance of the proposed parallel detection mechanism on a system of five layers. We used five instead of seven layers (as shown in Table VII) in the experiments because the power, hardware, and OS detectors cannot run concurrently in a physical host.

A. Parallel Detection Mechanism

In practice, some detectors, especially hardware detectors, cannot be queried in parallel. For example, the detectors for the host OS, hardware, and power components cannot be queried concurrently if they are implemented on IPMI. To support parallel detection, we can only use fault detectors that can be queried concurrently. Therefore, we cannot use a parallel detection mechanism for the seven-layer system shown in Table VII. To enable parallel detection, we must reorganize the seven-layer system into a five-layer system, comprising the host OS, network, VM process, guest OS, and VM network layers.

A naive parallel detection mechanism, which periodically queries all detectors in parallel and collects the detection

Authorized licensed use limited to: National Central University. Downloaded on September 30,2021 at 04:26:36 UTC from IEEE Xplore. Restrictions apply.

results, can be used in the case of parallel detection without any consideration of transient faults. However, in cases where transient faults must be considered, this approach may lead to misjudgment. For example, as illustrated in Fig. 22, a transient network fault was injected into the aforementioned five-layer cloud system at 0.8 s after the start of periodic parallel detection. The network detector responded that the network service was not faulty at the time point of 0.6 s, and other detectors subsequently responded by sending their detection results after the time point of 0.8 s, as shown in Fig. 22 (right panel). Accordingly, the system misjudged the VM process as faulty. This could lead to a catastrophe where the VM is destroyed and then restarted. To solve the problem shown in Fig. 22, we can reuse the idea underlying the mechanism proposed in Section III, as follows:

1) The system raises an alarm when a fault occurs. This can be achieved through continuous detection of the highest layer (L_{N-1}).
2) The system executes the following tasks after an alarm has been issued:
    a) Send a message to each layer detector (D_0 to D_{N-2}) for fault detection.
    b) Wait until all response messages from the layer detectors have been received.
    c) Use a binary search algorithm to find the faulty layer based on the response results. Notably, step 2c occurs very quickly. Therefore, in practice, a sequential search algorithm can be used in place of a binary search algorithm.
3) The system determines whether the fault is a permanent fault and returns the result.

The soundness of this parallel detection mechanism is demonstrated in Sections III-A–III-C.

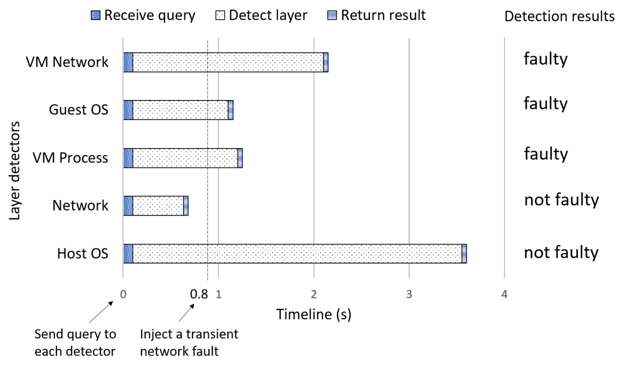


Fig. 22. Case of transient fault in naive periodic parallel detection, where a network transient fault is injected at the time point of 0.8 s.

B. Experiment Results of Parallel Fault Detection

Table IX lists the average fault-case detection times for each fault case detected using the proposed parallel detection mechanism on the five-layer system. In the experiments, we reused the settings of the environment and the fault injection methods described in Section V. The major differences between the settings pertained to the inapplicability of the power and hardware detectors to the experiment featuring parallel detection. The experimental results listed in Tables VIII and IX demonstrate the following.

1) As presented in Table IX, the average fault-case detection times were very similar, except for the average fault-case detection time for the network layer. This similarity is because Steps 1 and 2 in the parallel detection mechanism had the same execution time for every fault case. The average fault-case detection time for the network layer was much longer than that for the other layers because in Step 3, a waiting time of 29 s was required for transient fault verification.
2) The parallel detection mechanism outperformed the proposed mechanism with the proposed sequential dynamic-programming–based binary search tree algorithm in terms of the average fault-case detection time for the host OS layer. Nonetheless, the sequential mechanism was approximately 1 s faster (1.17 and 1.49 s faster, respectively) when the hardware and host OS faults were injected; the mechanism was 3.15 s slower when the power fault was injected. This is because the tree structure of the sequential algorithm prefers power fault detection.
3) The sequential mechanism outperformed the parallel mechanism in terms of the average fault-case detection time for the VM process, guest OS, and VM network layers by 1.15, 1.12, and 2.14 s, respectively. This is because the parallel mechanism had to wait for the slowest fault detector (host OS detector) and because the software-based detectors were faster than the slowest fault detector.
4) The ADT of the parallel mechanism was 9.48 s, which was 0.57 s slower than that of the sequential mechanism. This is because the parallel mechanism could not fully utilize fast hardware-based detection components and had to wait for the response of the slowest fault detector.

TABLE IX
AVERAGE FAULT-CASE DETECTION TIME (IN SECONDS) FOR EACH FAULT CASE IN PARALLEL DETECTION.

Host OS	Network	VM process	Guest OS	VM Network
4.79	33.76	4.83	4.80	4.82

VII. Conclusion and Future Research Directions

In this paper, we propose an efficient online liveness fault detection mechanism for multilayer cloud computing systems, in particular for virtualized compute hosts on the cloud. The proposed mechanism is based on the integration of existing detectors to quickly detect liveness faults, rather than on new detectors. In the virtualized compute hosts designed for the experiment, the proposed mechanism significantly reduced the

Authorized licensed use limited to: National Central University. Downloaded on September 30,2021 at 04:26:36 UTC from IEEE Xplore. Restrictions apply.

time required for detecting certain liveness faults by using detectors in each layer, thereby improving average fault detection efficiency. Our proposed mechanism also exhibited the highest efficiency under experimental conditions. According to the experimental results in Section V, the proposed mechanism, relative to system layer heartbeating, required a 70.3% shorter ADT and had the added ability to locate the specific layer of the fault while yielding comparable reliability. Locating faults within specific layers allows for effective application of fault recovery methods. In addition, our proposed mechanism had a 7.4% and 31.3% shorter ADT relative to two counterpart methods in the literature.

Our proposed mechanism is limited in that detectors at a given layer are unable to detect faults in other layers. Detection time in our method can be reduced if a layer's detector can detect and distinguish faults in both its layer and other layers. This characteristic within a multilayer system is termed detector dependency. The following is an example. Because a VM network detector detects faults through Internet control message protocol, VM network detection is based on the network layer. If the VM network layer detector can detect a network layer fault, there should be a shortcut from the VM network detector node to the network detector node in the proposed binary search tree. Thus, detection on all layers above the network layer can be skipped, and detection time can be reduced. This is a potential topic for future research. Future studies can construct a more reliable model that covers a sequence of faults (i.e., more than one fault) occurring in one detection round.

Acknowledgments

This study was supported by the Ministry of Science and Technology of Taiwan, under Grants 108-2221-E-008-032-MY3, and the Software Research Center, National Central University.

References

[1] VMware. "vSphere Availability." https://docs.vmware.com/en/VMware-vSphere/6.7/vsphere-esxi-vcenter-server-671-availability-guide.pdf (accessed 27 Jun., 2021).

[2] K. Kim and E. B. Bartlett, "Nuclear power plant fault diagnosis using neural networks with error estimation by series association," IEEE Trans. Nucl. Sci., vol. 43, no. 4, pp. 2373 - 2388, Aug 1996.

[3] R. Fonod, D. Henry, C. Charbonnel, and E. Bornschlegl, "Robust thruster fault diagnosis: application to the rendezvous phase of the Mars sample return mission," presented at the 2nd CEAS Specialist Conference on Guidance, Navigation and Control, Delft, Netherlands, Apr, 2013.

[4] J. Peters. "Prolonged AWS outage takes down a big chunk of the internet." https://www.theverge.com/2020/11/25/21719396/amazon-web-services-aws-outage-down-internet (accessed 5 May, 2021).

[5] W.-J. Wang, H.-L. Huang, S.-H. Chuang, S.-J. Chen, C. H. Kao, and D. Liang, "Virtual machines of high availability using hardware-assisted failure detection," presented at the International Carnahan Conference on Security Technology (ICCST), Taipei, Taiwan, Sept., 2015.

[6] Z. Amin, N. Sethi, and H. Singh, "Review on Fault Tolerance Techniques in Cloud Computing," International Journal of Computer Applications, vol. 116, no. 18, pp. 11-17, April 2015.

[7] A. G. d. M. Rossetto et al., "A New Unreliable Failure Detector for Self-Healing in Ubiquitous Environments," presented at the International Conference on Advanced Information Networking and Applications, Gwangiu, South Korea, March, 2015.

[8] A. Aviziens, "Fault-Tolerant Systems," IEEE Transactions on Computers, vol. C-25, no. 12, pp. 1304 - 1312, Dec. 1976.

[9] M. S. Rahman, M. Y. S. Uddin, T. Hasan, M. S. Rahman, and M. Kaykobad, "Using adaptive heartbeat rate on long-lived TCP connections," IEEE/ACM Transactions on Networking, vol. 26, no. 1, pp. 203-216, 15 December 2017.

[10] M. Nabi, M. Toeroe, and F. Khendek, "Availability in the cloud: State of the art," Journal of Network and Computer Applications, vol. 60, pp. 54–67, 2016.

[11] D. Trihinas, G. Pallis, and M. Dikaiakos, "Monitoring elastically adaptive multi-cloud services," IEEE Transactions on Cloud Computing (Early Access), pp. 1-14, 23 December 2015.

[12] Y. Wu, Y. Yuan, G. Yang, and W. Zheng, "An adaptive task-level fault-tolerant approach to grid," The Journal of Supercomputing, vol. 51, no. 2, pp. 97–114, 14 March 2009.

[13] Y.-L. Lee, M.-H. Ho, A. Suharsono, Y.-C. Pan, W.-J. Wang, and D. Liang, "NCU-HA: a lightweight HA system for kernel-based virtual machine," presented at the 2017 International Conference on Platform Technology and Service (PlatCon), Busan, South Korea, 15 Feb., 2017.

[14] C. M. Dobre, F. Pop, A. Costan, M. I. Andreica, and V. Cristea, "Robust Failure Detection Architecture for Large Scale Distributed Systems," presented at the Proceedings of the 17th International Conference on Control Systems and Computer Science, Bucharest, Romania, 2009.

[15] Z. Gao, C. Cecati, and S. X. Ding, "A survey of fault diagnosis and fault-tolerant techniques—part i: fault diagnosis with model-based and signal-based approaches," IEEE Transactions on Industrial Electronics, vol. 62, no. 6, pp. 3757-3767, 26 March 2015.

[16] Z. Gao, C. Cecati, and S. X. Ding, "A survey of fault diagnosis and fault-tolerant techniques—part ii: fault diagnosis with knowledge-based and hybrid/active approaches," IEEE Transactions on Industrial Electronics, vol. 62, no. 6, pp. 3768-3774, 01 April 2015.

[17] X. Zhang, L. Luan, L. Han, and Z. Lu, "Research and improvement on failure detection algorithm," presented at the 2008 Third International Conference on Pervasive Computing and Applications, Alexandria, Egypt, 8 Oct., 2008.

[18] F. Zhao, X. Koutsoukos, H. Haussecker, J. Reich, and P. Cheung, "Monitoring and fault diagnosis of hybrid systems," IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics), vol. 35, no. 6, pp. 1225-1240, 21 November 2005.

[19] Y. Mo, "A multiple-valued decision-diagram-based approach to solve dynamic fault trees," IEEE Trans. Rel., vol. 63, no. 1, pp. 81 - 93, March 2014.

[20] W. D. Shambroom, "Use of protocol validation and verification techniques in the design of a fault-tolerant computer architecture," presented at the FTCS-23 The Twenty-Third International Symposium on Fault-Tolerant Computing, Toulouse, France, 24 June, 1993.

[21] H. Alwi, C. Edwards, and C. P. Tan, "Fault tolerant control and fault detection and isolation," in Fault Detection and Fault-Tolerant Control Using Sliding Modes: Springer, 2011, pp. 7-27.

[22] C.-D. Lu, "Scalable Diskless Checkpointing for Large Parallel Systems," PhD dissertation, Univ. of Illinois at UrbanaChampaign, 2005.

[23] F. Wang, J. Shi, and L. Wang, "Method of diagnostic tree design for system-level faults based on dependency matrix and fault tree," presented at the IEEE 18th International Conference on Industrial Engineering and Engineering Management, Changchun, China, 3-5 Sept., 2011.

[24] M. K. Gokhroo, M. C. Govil, and E. S. Pilli, "Detecting and mitigating faults in cloud computing environment," presented at the 3rd IEEE International Conference (2017), Ghaziabad, India, 9-10 Feb., 2017.

[25] J. Villamayor, D. Rexachs, E. Luque, and D. Lugones, "RaaS: Resilience as a Service," presented at the 2018 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID), Washington, DC, 1-4 May, 2018.

[26] R. Yadav and A. S. Sidhu, "Fault tolerant algorithm for Replication Management in distributed cloud system," presented at the 2015 IEEE 3rd International Conference on MOOCs, Innovation and Technology in Education (MITE), Amritsar, 1-2 Oct., 2015.

[27] J. Liu, Z. Wu, J. Wu, J. Dong, Y. Zhao, and D. Wen, "A Weibull distribution accrual failure detector for cloud computing," PloS one, vol. 12, no. 3, p. e0173666, 2017.

[28] M. Bushnell and V. Agrawal, Essentials of electronic testing for digital, memory and mixed-signal VLSI circuits. Springer Science and Business Media, 2004.

1545-5971 (c) 2021 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.
Authorized licensed use limited to: National Central University. Downloaded on September 30,2021 at 04:26:36 UTC from IEEE Xplore. Restrictions apply.

[29] T. Zoppi, A. Ceccarelli, and A. Bondavalli, ”MADneSs: a Multi-layer Anomaly Detection Framework for Complex Dynamic Systems,” IEEE Transactions on Dependable and Secure Computing, pp. 1-14, 2019.
[30] W. Tarreau. ”HAProxy Documentation.” https://cbonte.github.io/haproxy-dconv/1.7/configuration.html#timeout%20server (accessed 3 May, 2021).
[31] S. Levine. ”Configuring the Red Hat high availability add-on with Pacemaker - additional fencing configuration options.” https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/configuring_the_red_hat_high_availability_add-on_with_pacemaker/s1-fencedevicesadditional-haar (accessed 3 May, 2021).
[32] Red Hat. ”Cloud Computing - What is PaaS?” https://www.redhat.com/en/topics/cloud-computing/what-is-paas (accessed 28 May, 2021).
[33] T. Taleb, K. Samdanis, B. Mada, H. Flinck, S. Dutta, and D. Sabella, ”On Multi-Access Edge Computing: A Survey of the Emerging 5G Network Edge Cloud Architecture and Orchestration,” IEEE Communications Surveys & Tutorials, vol. 19, no. 3, pp. 1657-1681, 2017, doi: 10.1109/COMST.2017.2705720.
[34] M. Soualhia, F. Khomh, and S. Tahar, ”A Dynamic and Failure-Aware Task Scheduling Framework for Hadoop,” IEEE Transactions on Cloud Computing, vol. 8, no. 2, pp. 553-569, 1 Jun. 2020, doi: 10.1109/TCC.2018.2805812.
[35] X. Lai, H. Wang, J. Zhao, F. Zhang, C. Zhao, and G. Wu, ”Research on High Availability Architecture of Cloud Platform,” Journal of Physics: Conference Series, vol. 1345, no. 2, pp. 1-4, Nov. 2019, doi: 10.1088/1742-6596/1345/2/022044.
[36] S. Senbel, ”Teaching Self-Balancing Trees Using a Beauty Contest,” presented at the Proceedings of the 2019 ACM Conference on Innovation and Technology in Computer Science Education, Aberdeen, Scotland Uk, 15-17 Jul., 2019.
[37] J. Daniels, ”Server virtualization architecture and implementation,” XRDS: Crossroads, The ACM Magazine for Students, vol. 16, no. 1, pp. 8-12, 2009, doi: https://doi.org/10.1145/1618588.1618592.
[38] P. Mell and T. Grance, ”The NIST definition of cloud computing,” National Institute of Standards and Technology, Gaithersburg, Maryland, Special Publication 800-145, 2011.
[39] U. Parui and V. Sanil, ”Introduction to Microsoft Azure,” in Pro SQL Server Always On Availability Groups: Springer, 2016, pp. 277-281.

Yen-Lin Lee received his BS degree in Software Engineering and Management from National Kaohsiung Normal University in Taiwan. He is currently working towards the Ph.D. degree at the Department of Computer Science and Information Engineering, National Central University, Taiwan. His recent research interests include high availability, fault tolerance, cloud computing, and edge computing.

Deron Liang received his Ph.D. degree in Computer Science from University of Maryland at College Park in the United States. He is currently a Professor and Director for Software Research Center in National Central University, Taiwan. His research interests include intelligent systems software, smart manufacturing, data analysis, software engineering, and information system security.

Wei-Jen Wang received his Ph.D. degree in Computer Science from Rensselaer Polytechnic Institute (RPI) in the United States on December, 2006. He is currently an Associate Professor in Department of Computer Science and Information Engineering, National Central University, Taiwan. His research interests include distributed programming technology, cloud computing, edge computing, high availability, and fault tolerance.

Authorized licensed use limited to: National Central University. Downloaded on September 30,2021 at 04:26:36 UTC from IEEE Xplore. Restrictions apply.