International Journal of Innovative Computing, Information and Control
Volume **13**, Number **2**, April **2017**
ICIC International ©2017 ISSN 1349-4198
pp. 365–379

DISTRIBUTED SRN MANAGER ON A RESILIENT SERVER WITH MULTIPLE VIRTUALIZATION ENGINES

Idris Winarno[1], Takeshi Okamoto[2], Yoshikazu Hata[1] and Yoshiteru Ishida[1]

[1]Department of Computer Science and Engineering
Toyohashi University of Technology
1-1 Hibarigaoka, Tempaku-cho, Toyohashi, Aichi 441-8580, Japan
{ idris; hata }@sys.cs.tut.ac.jp; ishida@cs.tut.ac.jp

[2]Department of Information Network and Communication
Kanagawa Institute of Technology
1030 Shimo-ogino, Atsugi, Kanagawa 243-0292, Japan
take4@nw.kanagawa-it.ac.jp

Received September 2016; revised December 2016

Abstract. *Presently, modern network servers run using virtualization technology since this technology offers flexibility and ease of administration. However, we have to consider that this technology also delivers a new problem that may hamper services that run on the server. In our previous work, we introduced the combination of a self-repair network model and virtualization technology to solve failures that can occur on the server. However, we need to increase the resilience of the server since the failure not only occurs in the guest operating system, but also in the host operating system. We proposed a new design for a resilient server with multiple virtualization engines and distributed the SRN manager to all of the physical servers. The result shows that our new design is able to remedy failures not only in the guest OS, but also in the host OS (virtualization engines).*
Keywords: Virtualization engines, Self-repair network, Server, Fault-tolerant

1. Introduction.

More and more people rely on an information system to satisfy their needs in daily life. They can obtain information relating to commerce, education, news, and a myriad of other topics via the Internet. This reality is evidenced by the forecasting of the Cisco Visual Networking Index (Cisco VNI^{TM}) where it predicts that the level of global IP traffic will surpass the zettabyte [1]. Cisco VNI^{TM} also forecasts that the number of devices connected to networks will be three times higher than the global population by the year 2020. The computer (known as a server) serves the information system that is accessed by users. This machine plays an important role, making it imperative to preserve the operational existence of this machine.

The server can be constructed using either a traditional or a virtualization model. With the traditional model, one physical server consists of a single operating system while on the virtualization model, we can install more than one operating system inside a single physical server depending on the availability of its hardware resources (e.g., processor and memory). Consequently, many traditional servers have shifted to a virtualization model since this model offers numerous tangible benefits (i.e., portability and energy efficiency). Figure 1 illustrates the comparison between a traditional and virtualization model of a server.

When we choose to use a virtualization model for our server, it does not mean that our server will be spared from potential failure. By migrating our server model from traditional to virtualization leads to a new problem. We have simulated a resilient server

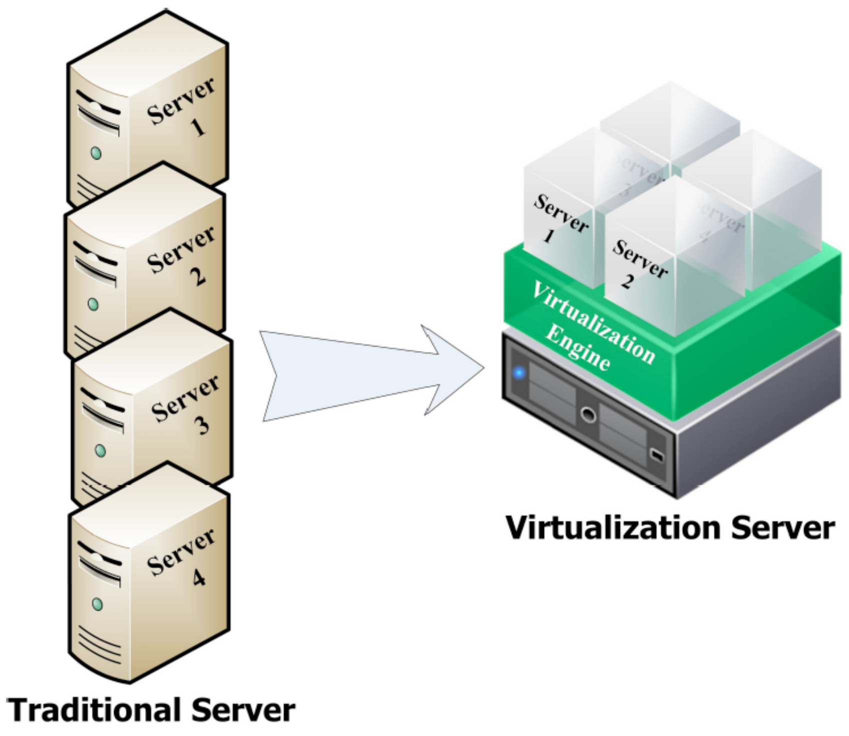

FIGURE 1. Comparison between traditional and virtualization server models

with a homogenous guest operating system (OS) using XEN as the Virtual Machine Monitor (VMM) [2] and Docker as the Container [3]. In the simulation, we implement a Self-Repair Network (SRN) model [4] to solve the failure that occurs on the server. Furthermore, we enhanced our work by increasing the diversity of the guest OS where we created a heterogeneous guest OS [5]. However, this solution only rescues the server from the perspective of the guest OS; in the other words, we also need to rescue the server from the virtualization engine since there is a type of server failure that can be caused by the virtualization engines themselves (e.g., CVE-2016-1571 [6]).

We start to build heterogeneous virtualization engines by utilizing multiple virtualization engines (XEN as VMM and LXC as Container) to solve the problems that can occur on them [7]. We equipped the virtualization engine with an application that operates with a role to monitor and respond to failures called the SRN manager. In this work, the SRN manager implements the SRN model that works to monitor and respond to all of the virtualization engines. Since we use a single (centralized) SRN manager for all of the virtualization engines and if the SRN manager has a problem, then all the virtualization engines will operate without any supervision.

To solve the issues on the resilient server with the centralized SRN manager, where it acts as a single central point for system failures, we were motivated to introduce a new design for a resilient server with multiple virtualization engines by distributing (decentralizing) the SRN manager in each of the physical server machines. The goal of this new design is to increase the availability of the SRN manager on the resilient server so that it is able to monitor and respond to failures occurring on the guest OS and the virtualization engines.

This work is organized as follows. First, in Section 2, we discuss the work that is related to resilient servers and compare it to our own work. In Section 3, we review the types of resilient server that we have developed, including the one with the centralized SRN manager with multiple virtualization engines. Since the resilient server is implementing the SRN model, in Section 4 we explain the relationship between the SRN model and the resilient server. Then we describe system design and the implementation of our work and reveal the results of the experiment in Sections 5 and 6, respectively. Finally, Section 7 concludes our present work and explains our intended future work.

2. **Related Work.** There are several ways to make our server work even when there is a system failure. The first way is in implementing a fault-tolerant system, where redundancy is its primary element [8]. In contrast to the resilient server, the fault-tolerant system focuses on "passive activity" where the system keeps functioning without significant changes. Meanwhile, the resilient server focuses on "active activity" whereby the system adapts to environmental changes by altering the fundamental structures to preserve their function. A honeypot using N-version programming is one of the implementations of a fault-tolerant system [9] where it runs multiple OSes and web services. Further, this project only runs in single virtualization engines called VMware; meanwhile in our work, the server not only provides multiple OSes and web services, but also uses multiple virtualization engines.

The other way to reduce the risk of server failure is by involving a virtualization technique. This technique can help the server in solving system failures, such as hang [5,7] and cyber attack [10]. We can combine the fault-tolerant system and virtualization technique to improve the availability of the server. This arrangement has already been introduced by several vendors, e.g., VMware [11]. This combination can be achieved when there is more than one physical server machine as primary and secondary machines (Figure 2). Remus [12] is one of the implementations of this combination. Since we have to use more than one physical server machine, we need to utilize a virtualization manager administrator to manage or control all the virtualization engines that run on each of the physical server machines. However, this virtualization manager administrator works only

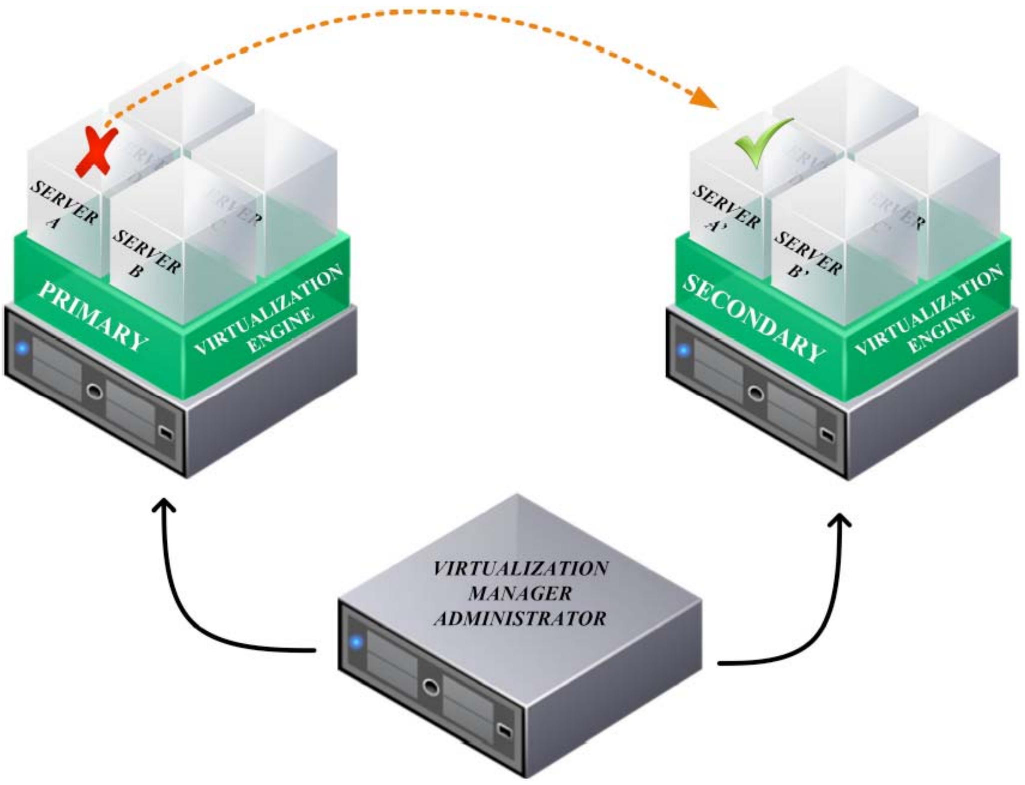

FIGURE 2. Combination of fault-tolerant system and virtualization

with the same type of virtualization engines (e.g., VMM) [13]. Meanwhile, to increase the availability of the server we need to use different types of virtualization engines (VMM and Container) [7] and these virtualization engines should be under supervision by the virtualization manager administrator that we refer to as the SRN Manager.

This work focuses on how to improve the existence of the SRN manager to monitor and respond to failures on VMM and Container since we have to consider that the failure of the server not only occurs on the guest OS of the virtual machine, but also on the host OS (virtualization engines).

3. Resilient Server Types.

As we explained earlier, the resilient server runs an active activity to preserve its function when the failure occurs. The active activity intends to adapt to environmental changes by changing fundamental methods or structure. We implement an SRN model [4] as the active activity to the resilient server. This model is inspired by the immunity-based system which is discussed further in Section 4.

The resilient server can be developed with the involvement of several kinds of technology. In this work, we focus on developing a resilient server utilizing virtualization technology. We can develop several types of resilient server by involving this technology. The types of the resilient server that can be realized by utilizing specific combinations with several parameters include:

a) Virtualization engine: software that provides an environment where the virtual machine can be run.
b) Guest OS: a virtual machine that is used to create the server.
c) Application (service): an application that provides a service, e.g., web server.

As shown in Figure 3, the combination of each parameter on type #1 to #8 shows that diversity becomes high. For example, type #1 has the same specification (homogeneous)

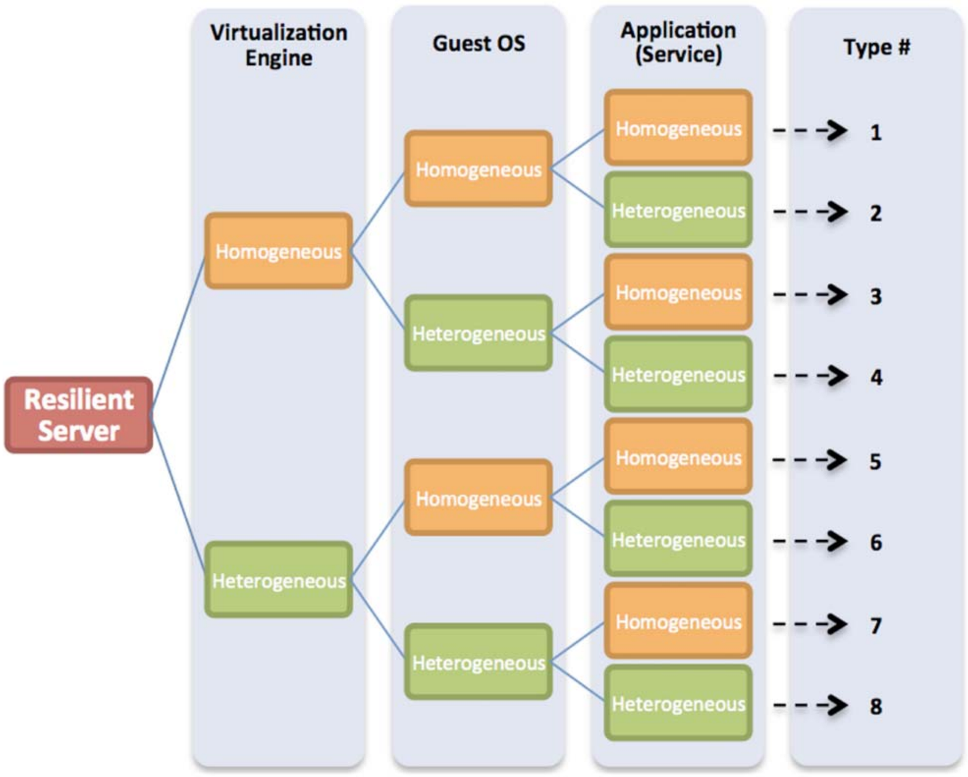

FIGURE 3. Types of resilient server with three parameter combination

on each parameter so that we can call type #1 as a homogeneous resilient server. Meanwhile, types #2 to #8 have different specifications on each parameter (heterogeneous) so that we can call them heterogeneous resilient servers. Since type #8 of the resilient server shows that this type is the most heterogeneous combination compared to the other types, consequently, this work only focuses on resilient server type #8. Moreover, system design and implementation of resilient server type #8 will be explained further in Section 5.

4. **Resilient Server with a Self-Repair Network Model.**

The resilient server that we previously developed [2,3,5,7] is equipped with an application called an SRN Manager that implements a Self-Repair Network (SRN) model. This application has to monitor and respond to failures that occur on the virtual machine that operates as the server. This virtual machine runs a guest OS that we henceforth refer to as a node. There are four models of the SRN model that we will implement in this work which are described as follows.

4.1. **Self-repair model.**

This model has an ability to repair a failure that occurs on the nodes by itself. One of the failures that can be solved using this model is hang failure. This failure occurs when the node cannot respond to a request from users. There are a number of possible events that can cause hang failures such as infinite loops and indefinite wait [14]. Since we use virtualization technology, we can utilize the virtualization engine to reset the node that encounters the hang failure. The SRN manager that monitors the node will detect the hang failure and instruct the virtualization engine to reset the node. However, despite the node running normally after being reset by the virtualization engine, this solution cannot guarantee that the same hang failure will not occur again until the system administrator finds the actual cause.

4.2. **Mutual-repair model.**

The second model of SRN is a *mutual-repair* model that has an ability to repair the other nodes. This ability can be used to solve a problem that occurs on a node, for example, copying the normal part of a node to the abnormal node so that the abnormal node can work normally. However, it is hard to realize this model since all of the nodes have to be identical (homogeneous). An alteration of files on the web contents by the attacker is another failure that possibly occurs on the server. The attacker tries to modify web content by inserting an iframe element to redirect a user's access to their exploit kit server. Once the users have been successfully redirected to their exploit kit server, they can attack various vulnerabilities on the user's PC. The alteration can be detected using a security toolkit such as Tripwire. When the Tripwire detects the missing or modified files, then the compromised node can copy the missing or modified file from the normal node. However, copying files from the other nodes is not a complete solution to solve this problem since we have to be aware of the "double edged sword" phenomena [4].

4.3. **Mixed-repair model.**

This model contains a combination of two basic models of the SRN including the *self-repair* and *mutual-repair* model. In other words, a *mixed-repair* model has two abilities for repairing the node from the failure. We can use these abilities to solve a problem that occurs on a node such as a denial of service (DoS) attack. This attack is meant to hamper particular services, such as a web server, by flooding a lot of TCP packets making the service unable to process further connections from other users. Several researchers use a firewall feature inside the operating system (e.g., `iptables`) to solve this problem [15,16]. Therefore, we can utilize `iptables` to drop a DoS packet to secure the nodes with a limited scenario of DoS attack (i.e., TCP DoS attack). Since *mixed-repair* is used to solve a DoS attack, then *self-repair* is indicated by adding the attackers IP address to the firewall rule of the node itself. After that, information of the attackers IP address is sent to the virtualization engines in order to secure the other nodes by dropping all of the DoS packets indicating a *mutual-repair*.

4.4. **Switching-repair model.** The fourth model of SRN is the *switching-repair* model where this model has an ability to migrate the faulty node to the normal node. In the other words, we stop the faulty node and replace it with a normal node. This solution creates a higher cost of operation compared to other models since we need to provide a normal node to replace the faulty node. Therefore, we only use this model if the *self-repair*, *mutual-repair* and *mixed-repair* models are unable to solve the problem. Further, this solution targets the heterogeneous nodes and offers higher resilience than the other models.

5. **System Design and Implementation.** As we mentioned in Section 2 the server malfunction not only occurs on the application or guest OS, but also occurs on the host OS (virtualization engine). When the virtualization engine is faulty, then this condition will lead to all of the guest OS's inside the host OS to cease functioning. To address the problem that possibly occurs on the virtualization engines, we need to provide multiple virtualization engines for the resilient server. We built the resilient server with multiple virtualization engines (type #8) and placed the SRN manager to monitor and respond to failures. As the SRN manager plays an important role on the resilient server, we need to increase the availability of the SRN manager by modifying its availability from centralized to distributed. The difference between the centralized and distributed positions of the SRN manager and a description of the SRN manager is outlined as follows.

5.1. **Centralized SRN manager.** Figure 4 shows the design of a centralized SRN manager on type #8 of the resilient server. There are three physical servers used; two of them are used for the operation of virtualization engines while the other one is used for the SRN manager. Further, two types of virtualization engine are placed on each of the physical

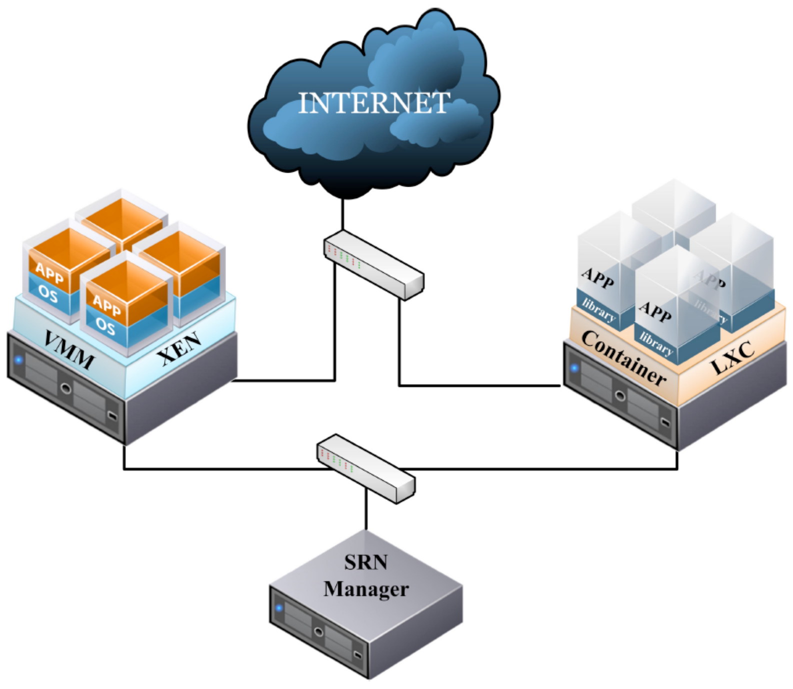

FIGURE 4. Centralized SRN manager on type #8 of the resilient server

servers in this paper: VMM and Container. We use XEN as VMM and LXC as Container. Meanwhile, the SRN manager has to monitor and respond to failures occurring on both of the virtualization engines that run on separate physical servers. This design implies a single point of failure since only a single SRN manager is run. When the SRN manager encounters a failure, there is no SRN manager available to respond to a failure that occurs on both of the virtualization engines simultaneously.

5.2. **Distributed SRN manager.** In order to eliminate the risk that appears in type #8 of the resilient server with a centralized SRN manager, we modify the design so that the SRN manager is distributed to all of the physical servers as shown in Figure 5. Only two physical servers are used in this design while in the centralized SRN manager we use three physical servers. Details of their technical specifications refer to our previous work [7] and is shown in Table 1. However, this table shows that node id's #5 to #8 have the same version due to the LXC characteristic which uses a shared kernel of the host OS. This condition will likely lead to trouble when the LXC kernel has a vulnerability that allows privilege escalation because the host OS and all of the nodes can be compromised.

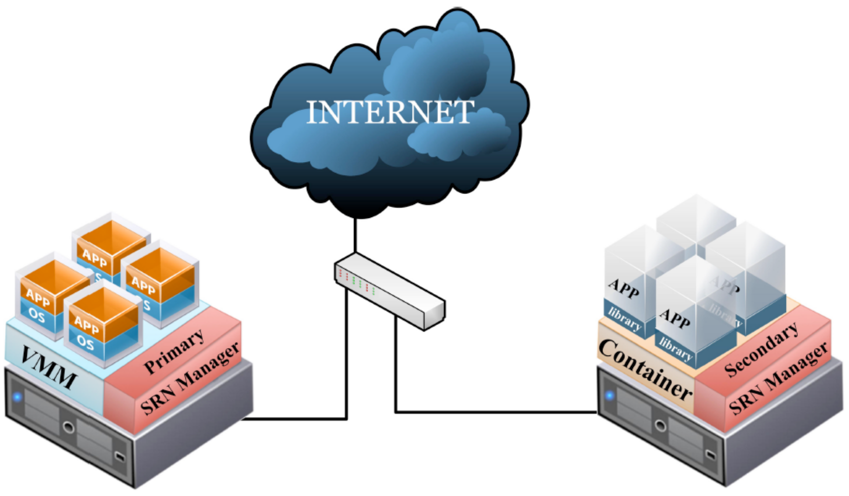

FIGURE 5. Distributed SRN manager on type #8 of the resilient server

TABLE 1. Technical specification of the physical server machine [7]

Virtualization Engines	**Node ID #**	**Guest OS**	**Build number (release)**	**Kernel version**	**Application**
VMM: XEN 4.1 on Debian GNU /Linux 7.8 with kernel version: 3.2.0-4-amd64	1	Debian GNU/Linux	7.8 (Wheezy)	3.2.0-4-amd64	Apache
	2	Ubuntu	12.04 (Precise)	3.11.0-15-generic	Apache Tomcat
	3	Fedora	22	4.0.4-301.fc22.x86_64	Nginx
	4	Windows 7	7601	–	IIS
Container: LXC 1.0.6 on Debian GNU/Linux 8.3 with kernel version: 3.16.0-4-686-pae	5	CentOS	6.7 (Final)	3.16.0-4-686-pae	Lighttp
	6	Debian GNU/Linux	8 (Jessie)	3.16.0-4-686-pae	Apache Tomcat
	7	Fedora	23	3.16.0-4-686-pae	Apache
	8	Ubuntu	16.04 (Xenial Xerus)	3.16.0-4-686-pae	Nginx

To address this problem, we should apply mandatory access control (e.g., AppArmor) for the LXC.

Since every physical server or virtualization engine is equipped with the SRN manager, then it should determine which one of the physical servers will operate as the primary SRN manager. If the primary SRN manager stops operating then the secondary SRN manager will replace the position.

5.3. **SRN manager.** Since the SRN manager is placed in each of the physical servers, then the SRN managers should determine their operation as either a primary or secondary SRN manager. To achieve this, they need to communicate and decide which will become the primary SRN manager. Algorithm 1 shows the communication between two SRN managers. The communication starts by sending a broadcast signal to the network. The SRN manager informs their priority number (line 5 of Algorithm 1). When the SRN manager listens to the other SRN manager, that has a higher priority number, then the lower priority number should change their status to an idle position (lines 7 and 8 of Algorithm 1). Further, when the primary SRN manager that has the highest priority number is working, it will trigger the other scripts (Algorithms 2, 3 and 4) to activate their function to monitor and respond to failures occurring on the server (line 6 of Algorithm 1).

Algorithm 1 Pseudo-code implementation of the script for communication between SRN Manager

```
1  brIpAddr ← defineBroadcastIpAddress()
2  listenTimeout ← generateRandomValue()
3  priority ← definePriority()
4  repeat
5      sendActiveSrnmSignal (brIpAddr, priority)
6      activeSrnm ← TRUE
7      check ← isThereAnyActiveSrnmSignalWithHigherPriority()
8      if check = TRUE then
9          otherSrnmIsActive ← TRUE
10         repeat
11             i ← 0
12             repeat
13                 otherSrnmIsActive ← listenToActiveSrnmSignal(brIpAddr)
14                 activeSrnm ← FALSE
15                 i ← i + 1
16             until listenTimeout < i
17         until otherSrnmIsActive = FALSE
18     end if
19 until scriptIsStopped
```

We create several scenarios of failure of the server that are similar to those used in our previous works, including hang, DoS attack, and malware. Furthermore, the failures that are described in these scenarios not only occur in the guest OS (node) but also in the host OS (virtualization engine). Since we use a distributed SRN manager, we need to modify our previous algorithm [5]. The modification aims to adapt to the new design of the SRN manager as shown in lines 4 and 5 of Algorithms 2, 3, and 4. We also need to consider implementation of Algorithms 2, 3, and 4, since we use multiple virtualization engines. For example, in Algorithm 2 (hang scenario), when the credibility (R) of the node is higher than the threshold (line 7 of Algorithm 2) then the node will be defined as the hang node. The SRN manager will respond to this failure by resetting the hang node (line 8 of Algorithm 2). If the hang node is running under VMM (XEN) then the SRN manager resets the hang node using the following commands:

```
# xm destroy <domain>
# xm start <domain>
```

Meanwhile, when a failure is detected on a node that is running under the container (LXC) then the way to reset the hang node is by using the following commands:

```
# lxc-stop -n <container_name>
# lxc-start -n <container_name>
```

Algorithm 2 Pseudo-code implementation of the script for hang scenario with the distributed SRN manager

```
$R \leftarrow 0$
$threshold \leftarrow 0.5$
repeat
    $activeSrnm \leftarrow$ checkTheSrnmStatus()
    if receiveRespondRequest = FALSE and $activeSrnm$ = TRUE then
        if $R \geq 1$ then
            if $R \geq threshold$ then
                resetTheHangNode()
                makeAReportToTheAdministrator()
            end if
        else
            $R \leftarrow R + 0.1$
        end if
    else
        if $R \leq 0$ then
            $R \leftarrow R - 0.1$
        end if
    end if
until scriptIsStopped
```

The same condition also occurs to Algorithm 3 at line 8 where it shows the response of the SRN manager when a DoS attack is detected. The node will be defined as an attacked node when the credibility is higher than the threshold (line 7 of Algorithm 3). The credibility of the node will increase if the number of existing connections (N_c) is higher than the allowed maximum number of connections (I_c) as shown in Algorithm 3 at line 5. When a DoS attack is detected, then the SRN manager will respond to the DoS attack by adding the suspected IP address of the attacker to the firewall rule in the node that is being attacked using the `iptables` command. However, to prevent the attacker from attacking the other nodes, then the SRN manager needs to place the suspected IP address on the firewall rule on both virtualization engines (VMM and container).

6. **Experimental Results and Discussions.** We test our design (type #8 of the resilient server) shown in Figure 5 to know the response of the SRN manager when the services (application), nodes (guest OS) and virtualization engine (host OS) encounter a failure based on the scenario explained previously. The server failure includes hang, DoS, and malware. The experimental results of the failure scenarios are shown as follows.

Algorithm 3 Pseudo-code implementation of the script for DoS scenario with the distributed SRN manager

```
$R \leftarrow 0$
$threshold \leftarrow 0.5$
repeat
    $activeSrnm \leftarrow$ checkTheSrnmStatus()
    if $N_c \geq I_c$ and activeSrnm = TRUE then
        if $R \geq 1$ then
            if $R \leftarrow threshold$ then
                addTheSuspectedIpToFirewall()
                makeAReportToTheAdministrator()
            end if
        else
            $R \leftarrow R + 0.1$
        end if
    else
        if $R \leq 0$ then
            $R \leftarrow R - 0.1$
        end if
    end if
until scriptIsStopped
```

Algorithm 4 Pseudo-code implementation of the script for Malware scenario with the distributed SRN manager

```
$R \leftarrow 0$
$threshold \leftarrow 0.5$
repeat
    $activeSrnm \leftarrow$ checkTheSrnmStatus()
    if $detectMalware$ = TRUE and $activeSrnm$ = TRUE then
        cleanTheMalware()
        $R \leftarrow R + 0.1$
        if $R \geq threshold$ then
            $isFound \leftarrow$ findNormalNodeWithSameAppsButDifferentOs()
            if $isFound$ = TRUE then
                switchTheAbnormalNodeWithStandbyNode()
                makeAReportToTheAdministrator()
            else
                isolateTheNode()
                makeAReportToTheAdministrator()
            end if
        end if
    else
        if $R \leq 0$ then
            $R \leftarrow R - 0.1$
        end if
    end if
until scriptIsStopped
```

6.1. **Hang.** There are several possibilities that can trigger a hang condition such as exploitation of a vulnerability, misconfiguration of a service, physical failures, and unknown reasons. In this scenario, we only focus on the hang condition caused by an unknown reason, since the other possibilities are unable to be solved using *self-repair* but may be able to be solved using *switch-repair*. Further, although the node can be successfully reset, it cannot be guaranteed that the hang problem is permanently solved. We use ICMP ping to ensure the responsiveness of the guest OS and the host OS. Meanwhile, TCP and UDP ping are used to ensure the responsiveness of the application that runs on the guest OS. To simulate the hang scenario, we use the `halt` command to trigger a hang condition. The `halt` command applied not only to the node (guest OS) but also to the virtualization engine (host OS) so that all the nodes under VMM (XEN) stop running as shown in Table 2. When the SRN manager on VMM stops running, the SRN manager switches to the secondary SRN manager that is running under the container (LXC) and switches the failed nodes (X) to the normal node (O) that have the identical services.

When the resilient server uses the centralized SRN manager, then it will be vulnerable to hang failure as shown in Table 3. This table shows that a resilient server where the SRN manager is in an abnormal state (X) whereby the guest OS that is having a hang failure on both virtualization engines is unable to be recovered (reset/migrate).

6.2. **Denial of service (DoS).** In this scenario, we assume that the DoS attack is based on the TCP connection. There are various applications that we can use to simulate DoS attacks such as `slowhttptest` [17], `slowloris` [18], and `httperf` [19]. When an attacker

Table 2. Experimental result of the hang failure with the distributed SRN manager

Virt. Engine	**Node ID #**	**Guest OS**	**Guest OS Credibility**	**Guest OS State**	**SRN Man. State**	**SRN Man. Respond**	**SRN Model**
VMM	1	Debian	0.9	X	X	*Migrate*	*switch-repair*
	2	Ubuntu	0.9	X		*Migrate*	*switch-repair*
	3	Fedora	0.9	X		*Migrate*	*switch-repair*
	4	Win. 7	0.9	X		*Migrate*	*switch-repair*
Container	5	CentOS	0.6	X	O	*Reset*	*self-repair*
	6	Debian	0	O		–	–
	7	Fedora	0	O		–	–
	8	Ubuntu	0	O		–	–

Table 3. Experimental result of the hang failure with the centralized SRN manager

Virt. Engine	**Node ID #**	**Guest OS**	**Guest OS Credibility**	**Guest OS State**	**SRN Man. State**	**SRN Man. Respond**	**SRN Model**
VMM	1	Debian	0.6	X	X	*(unable to respond)*	–
	2	Ubuntu	0.6	X		*(unable to respond)*	–
	3	Fedora	0.6	X		*(unable to respond)*	–
	4	Win. 7	0.6	X		*(unable to respond)*	–
Container	5	CentOS	0.6	X		*(unable to respond)*	–
	6	Debian	0	O		*(unable to respond)*	–
	7	Fedora	0	O		*(unable to respond)*	–
	8	Ubuntu	0	O		*(unable to respond)*	–

Table 4. Experimental result of the DoS attack failure with the distributed SRN manager

Virt. Engine	**Node ID #**	**Guest OS**	**Guest OS Credibility**	**Guest OS State**	**SRN Man. State**	**SRN Man. Respond**	**SRN Model**
VMM	1	Debian	0.6	X	X	*Block*	*mixed-repair (self-repair)*
	2	Ubuntu	0.1	O		*Block*	*mixed-repair (mutual-repair)*
	3	Fedora	0	O		*Block*	*mixed-repair (mutual-repair)*
	4	Win. 7	0	O		*Block*	*mixed-repair (mutual-repair)*
Container	5	CentOS	0	O	O	*Block*	*mixed-repair (mutual-repair)*
	6	Debian	0	O		*Block*	*mixed-repair (mutual-repair)*
	7	Fedora	0	O		*Block*	*mixed-repair (mutual-repair)*
	8	Ubuntu	0	O		*Block*	*mixed-repair (mutual-repair)*

Table 5. Experimental result of the DoS attack failure with the centralized SRN manager

Virt. Engine	**Node ID #**	**Guest OS**	**Guest OS Credibility**	**Guest OS State**	**SRN Man. State**	**SRN Man. Respond**	**SRN Model**
VMM	1	Debian	0.6	X	X	*(unable to respond)*	–
	2	Ubuntu	0.6	X		*(unable to respond)*	–
	3	Fedora	0	O		*(unable to respond)*	–
	4	Win. 7	0	O		*(unable to respond)*	–
Container	5	CentOS	0	O		*(unable to respond)*	–
	6	Debian	0	O		*(unable to respond)*	–
	7	Fedora	0	O		*(unable to respond)*	–
	8	Ubuntu	0	O		*(unable to respond)*	–

attacks one of the nodes under VMM or container then the SRN manager will respond by adding the suspected IP address of the attacker to the firewall rule inside the attacked node. In addition, at the same time, the SRN manager also adds the suspected IP address to the firewall rule on both virtualization engines (VMM and container). Table 4 shows that node id #1 is being attacked (X), and the SRN manager responds to protect the normal nodes (O) by notifying the IP address of the attacker to the virtualization engines.

Although one of the SRN managers is being attacked making it unable to protect the normal nodes, then the other SRN manager will replace its position. In contrast to the resilient server with the distributed SRN manager, Table 5 shows the resilient server with a centralized SRN manager unable to protect the normal node since the SRN manager is in the abnormal state (X).

6.3. **Malware.** The same situation occurs in this scenario where malware has the possibility to infect either the node (guest OS) or the virtualization engine (host OS). When malware is detected by the anti-malware (e.g., `rkhunter` and `chkrootkit` [20]) then the SRN manager responds by cleaning the malware as shown by node id #1 in Table 6. If the malware still resides after cleaning process, then the SRN manager responds by switching the abnormal node (X) to the normal node (O) that has the same service as shown by node id #2 in Table 6.

The worst case happens when the malware attacks the resilient server with the centralized SRN manager as shown in Table 7. When the malware successfully infects the centralized SRN manager and cannot be cleaned or solved by the anti-malware then the entire nodes are left without any supervision.

Table 6. Experimental result of the malware failure with the distributed SRN manager

Virt. Engine	**Node ID #**	**Guest OS**	**Guest OS Credibility**	**Guest OS State**	**SRN Man. State**	**SRN Man. Respond**	**SRN Model**
VMM	1	Debian	0.6	X	X	*Clean*	*self-repair*
	2	Ubuntu	0.9	X		*Migrate*	*switch-repair*
	3	Fedora	0	O		–	–
	4	Win. 7	0	O		–	–
Container	5	CentOS	0	O	O	–	–
	6	Debian	0	O		–	–
	7	Fedora	0	O		–	–
	8	Ubuntu	0	O		–	–

Table 7. Experimental result of the malware failure with the centralized SRN manager

Virt. Engine	**Node ID #**	**Guest OS**	**Guest OS Credibility**	**Guest OS State**	**SRN Man. State**	**SRN Man. Respond**	**SRN Model**
VMM	1	Debian	0.6	X	X	*(unable to respond)*	–
	2	Ubuntu	0.6	X		*(unable to respond)*	–
	3	Fedora	0	O		*(unable to respond)*	–
	4	Win. 7	0	O		*(unable to respond)*	–
Container	5	CentOS	0	O		*(unable to respond)*	–
	6	Debian	0	O		*(unable to respond)*	–
	7	Fedora	0	O		*(unable to respond)*	–
	8	Ubuntu	0	O		*(unable to respond)*	–

We also tested our system by counting the time spent by the server recovering from failure, especially in the hang scenario since it is required to reset the abnormal node (*self-repair*) or even to switch to the normal node (*switch-repair*). We commence counting when the virtualization engine resets the hang node, and the count is ended when the service is ready to be accessed. After conducting 10 trials of the test, a node using Linux OS spent 17.5 seconds on average and a node using Windows OS took 58.3 seconds on average. The time spent shown by our system is acceptable because if we did it manually

Table 8. Comparison of resilient server types

Failure \ Resilient Server	Type #2 to #4	Type #8
Hang on the node (guest OS)	*self-repair*	*self-repair*
Hang on the Virtualization Engine (host OS)	(unable to repair)	*switching-repair*
DoS on the node (guest OS)	*mixed-repair*	*mixed-repair*
DoS on the Virtualization Engine (host OS)	(unable to repair)	*mixed-repair*
Malware on the node (guest OS)	*self-repair*, *mutual-repair*, *switching-repair*	*self-repair*, *mutual-repair*, *switching-repair*
Malware on the Virtualization Engine (host OS)	(unable to repair)	*switching-repair*

then we would need to add additional time for getting access to the host OS and inputting the reset instruction to the virtualization engine.

Based on the experimental result of resilient server type #8 and compared to the previous resilient server [5], this design is able to solve a failure that occurs not only on the node (guest OS) but also in the host OS where the virtualization engine is running. The distributed SRN manager that is placed on all of the physical servers can provide higher availability than that seen in our previous work [7]. Further, this design is also inspired by the diversity of operating systems [21] and an immunity-based system [22]. Table 8 shows a comparison of the resilient server between type #2 to #4 and type #8. The resilient server type #8 is able to recover from limited scenarios of failure (i.e., hang, DoS attack, and malware) by implementing an SRN model.

In order to increase the resilience of the server, we need to involve other mechanisms or technologies, such as the Software-Defined Networking (SDN) concept that offers flexibility to organize network topology so that the server is more resilient to failure. When the server can organize the network using the SDN concept, then it can avoid particular failure, e.g., DoS attack [23]. Moreover, we also have to consider physical failure due to natural events (e.g., earthquakes, floods) that can disturb the functionality of the server. To address this failure, we can implement a disaster recovery system where this system provides two or more redundant fault-tolerant systems.

7. Conclusion.

There were three parameters, including service (application), guest OS, and virtualization engine that we used in this work to design a new type of resilient server. The SRN manager that implements a Self-Repair Network model is utilized to solve failures that possibly occur on the guest OS and virtualization engines. What is different to our previous work is that we distributed the SRN manager to all of the physical servers so that the availability of the SRN manager is higher than that of a centralized SRN manager.

In the future, we will involve other technologies such as a software-defined networking concept to obtain a higher level of server resilience not only in respect to the host but also from the perspective of the entire network.

Acknowledgment. We would like to thank the anonymous reviewers for their valuable comments and suggestions that greatly contributed to improving the presentation. We would like to thank to DIKTI (Indonesian Government for Higher Education) for the BPPLN scholarship.

REFERENCES

[1] Cisco Systems, Cisco visual networking index: Forecast and methodology, *White Paper*, pp.2015-2020, 2016.

[2] I. Winarno and Y. Ishida, Simulating resilient server using XEN virtualization, *Procedia Computer Science*, vol.60, pp.1745-1752, 2015.

[3] I. Winarno, T. Okamoto, Y. Hata and Y. Ishida, Implementing SRN for resilient server on the virtual environment using container, *Intelligent System Research Progress Workshop*, 2015.

[4] Y. Ishida, *Self-Repair Network: A Mechanism Design*, Springer, 2015.

[5] I. Winarno, T. Okamoto, Y. Hata and Y. Ishida, A resilient server based on virtualization with a self-repair network model, *International Journal of Innovative Computing, Information and Control*, vol.12, no.4, pp.1059-1071, 2016.

[6] *CVE-2016-1571*, https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-1571.

[7] I. Winarno, T. Okamoto, Y. Hata and Y. Ishida, Increasing the diversity of resilient server using multiple virtualization engines, *Procedia Computer Science*, vol.96, pp.1701-1709, 2016.

[8] A. S. Tanenbaum and M. V. Steen, *Distributed Systems: Principles and Paradigms*, 2nd Edition, Pearson, 2007.

[9] L. Nagy, R. Ford and W. Allen, N-version programming for the detection of zero-day exploits, *Proc. of IEEE Topical Conference on Cybersecurity*, 2006.

[10] F. Sano, T. Okamoto, I. Winarno, Y. Hata and Y. Ishida, A cyber attack-resilient server inspired by biological diversity, *Journal of Artificial Life and Robotics*, vol.21, no.3, pp.345-350, 2016.

[11] *VMware vSphere*, http://www.vmware.com/products/vsphere/enhanced-app-performance.html.

[12] B. Cully, G. Lefebvre, D. Meyer, M. Feeley, N. Hutchinson and A. Warfield, Remus: High availability via asynchronous virtual machine replication, *Proc. of the 5th USENIX Symposium on Network System Design and Implementation*, pp.161-174, 2008.

[13] J. D. Pike, D. Reeves and M. Gujarathi, Virtual machine manager for managing multiple virtual machine configurations in the scalable enterprise, *U.S. Patent No. 8,127,291*, 2012.

[14] Y. Zhu, Y. Li, J. Xue, T. Tan, J. Shi, Y. Shen and C. Ma, What is system hang and how to handle it, *Proc. of the 2012 IEEE the 23rd International Symposium on Software Reliability Engineering*, pp.141-150, 2012.

[15] K. Chatterjee, Design and development of a framework to mitigate DoS/DDoS attacks using iptables firewall, *International Journal of Computer Science and Telecommunications*, vol.4, no.3, pp.67-72, 2013.

[16] B. Q. M. Al-Musawi, Mitigating DoS/DDoS attacks using iptables, *International Journal of Engineering & Technology*, vol.12, no.3, 2012.

[17] *Slowhttptest Application Layer DoS Attack Simulator*, https://code.google.com/p/slowhttptest/.

[18] *Slowloris HTTP DoS*, https://ha.ckers.org/slowloris.

[19] *The httperf HTTP Load Generator*, https://github.com/httperf/httperf.

[20] M. Cabak, V. Gazivoda and B. Krstajic, Security recommendation for an Ubuntu server-based system, *MREN Best Practice Document*, 2016.

[21] C. Pu, *A Specialization Toolkit to Increase the Diversity in Operating Systems*, Ph.D. Thesis, Portland State University, 1996.

[22] Y. Ishida, *Immunity-Based System: A Design Perspective*, Springer, 2004.

[23] I. Winarno and Y. Ishida, Simulating resilient server using software-defined networking, *Proc. of International Conference on Advanced Informatics: Concepts, Theory and Application*, 2016.