A Resilient Server with a Self-Repair Network Model on the Virtualization Environment

（仮想化環境のための自己修復ネットワークモデルを用いたレジリエントサーバに関する研究）

January 2018

Doctor Of Engineering

Idris Winarno

イドリス　ウイナルノ

Toyohashi University Of Technology

Abstract

The performance of an information system depends on the reliability of the data center infrastructure. A computer which has a high specification and runs a service, which can be accessed by multiple users over a network, is called a server. The server is one of the most critical parts of a data center, hence can negatively affect the overall performance of an information system in cases of system failure (e.g. hang faulty, denial of service attack, and malware). Since servers play a critical role in data processing and data transmission for serving many clients, failures in servers cause not only performance degradation of the server itself but also threats to the entire computer connected to the servers. Resilient servers that can self-recognize failures, self-repair failures and self-replace failed parts are required when computer systems and networks become huge-scale as witnessed by data centers and the cloud computing. Recently, virtualization has become a popular method to develop servers for data center infrastructure. Thus, we require a model that can deal with virtual servers as a repair unit. There are two types of virtualization technology that we can use to build the resilient server: virtualization machine monitor (VMM) and container (also called as Linux Container or LXC). Container offers fast boot and efficient resource usage to deploy a server on the virtual environment. However, VMM provides higher diversity of guest operating system than container.

In this study, we introduce a new model of the resilient server by implementing a self-repair network (SRN) model in the virtualization environment to address failures that occur during the operation of the server. The SRN model offers a general framework of self-action models for the server to self-recognize, self-repair, and self-replace its own failed parts (learned from immune system). In this study we use the following four models of SRN: (i) self-repair, (ii) mutual-repair, (iii) mixed-repair, and (iv) switching-repair. The SRN model will be applied to the script that we called as SRN manager. This script has the main responsibility to monitor and respond to the failures. We define eight types of resilient server using the combination of three parameters including application (service), guest operating system, and host operating system (virtualization engine).

The simulations show that virtualization technology is able to build a resilient server to recover the failure in the limited scenario. The first experiment shows that container outperforms the VMM since container almost ten times faster and more efficient of memory resources compared to the VMM. However, a container has lower diversity than VMM due to container uses a shared kernel. The second experiment explains that a resilient server with a homogeneous environment offers ease of replacement of the failed parts, but this resilient server has less resilience than those with heterogeneous environment since the entire server in the homogenous environment has the same vulnerability. The last experiment describes that we use multiple virtualization engines to increase the diversity of the resilient server. Moreover, the performance of our proposed method has low performance losses and high service availability.

This page intentionally left blank

Table of Contents

This page intentionally left blank

List of Figures

List of Tables

This page intentionally left blank

List of Algorithms

This page intentionally left blank

Nomenclature

DC = data center
DoS = denial of service
DNS = domain name system
LXC = linux container
OS = operating system
SRN = self-repair network
VM = virtual machine
VMI = virtual machine introspection
VMM = virtual machine monitor
RS = Resilient server

This page intentionally left blank

1 Introduction

1.1. Background

Nowadays, the Internet has become an essential part of human life. The Internet is not only just for news or entertainment, but it also offers a significant amount of useful and practical content for users. More and more people rely on the Internet to get information to satisfy their needs in daily life. They can obtain information relating to commerce, education, news, games, and a myriad of other topics via the Internet. Hence, many common daily activities rely on the information obtained from the Internet.

The server is one of the essential parts of the Internet since servers are responsible for providing services for Internet users. A server is a computer of high specification (i.e. processor, memory, storage, etc.) that runs services or applications accessed by multiple users over a network [1]. This machine plays an important role, making it imperative to preserve the operational existence of this machine. The Domain Name System (DNS), Web server, Email, and other services are usually realized as a server. Therefore, system administrators, who are responsible for maintaining these servers, have to monitor and recover them when failure occurs.

In order to minimize the failure of a server, in traditional operations, the system administrator regularly makes routine checks or maintenance to the server. In the perspective of server (system) maintenance, the hardest part is evaluating (e.g. improving, repairing and securing) the services. The lifecycle of maintenance system is shown in Figure 1. There are four stages of the maintenance system as follows [2]:

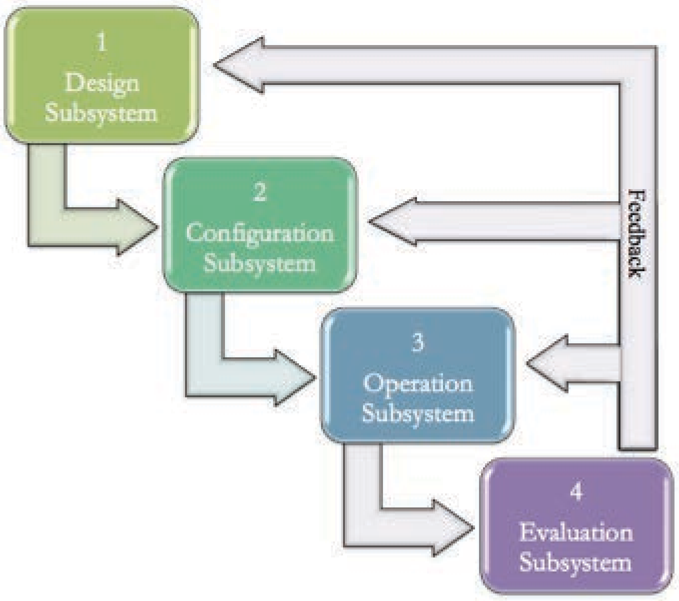


Figure 1. Maintenance system lifecycle.

1. Design Subsystem: In this stage, a system administrator needs to clarify the operation needs and establish the subsystem.
2. Configuration Subsystem: When the subsystem has been successfully installed, the next step is configuring the subsystem based on the requirement or design.
3. Operation Subsystem: After the configuration stage has finished, then the system administrator runs the subsystem to measure the performance of the whole system.
4. Evaluation Subsystem: The final stage is used to evaluate and give a feedback based on operational records and system failure events.

Meanwhile, server technologies have shifted from traditional to an era of virtual architecture (Figure 2). According to [3], they reported that the trend of server virtualization adoption worldwide would continue to grow in the future. Virtual machine (VM) technology is an implementation of virtual architecture. VM offers many benefits including CPU, memory, and I/O virtualization [4]. VM technologies offer features designed to maintain resources more easily and efficiently than traditional technologies where a single physical computer runs a single operating system. However, by using virtualization technology, the system administrator can create multiple virtual servers on a single physical computer. This means that a single physical computer can run multiple operating systems simultaneously. Here are the several advantages of VM [5] compared to traditional architecture:

1. Energy efficient: VM is able to decrease the total power consumption of computer server since we can create multiple computer servers in a single physical computer server.
2. Reduced data center footprint: Since we can reduce the number of physical computers, then there are fewer servers, less networking devices, and a smaller number of racks.
3. Faster server provisioning: We can clone (copy) the existing virtual machine in only a few minutes. Compared to traditional server, where hours are needed to setup.
4. Reduced hardware vendor lock-in: VM is middleware that provides emulation of the hardware, this feature offers flexibility to employ various types and brands of computer hardware.
5. Increased uptime: Most of traditional computer servers take longer to start. Using VM, in less than a minute the server will be ready to operate.
6. Improved disaster recovery: One of the capabilities of the VM is moving from one physical server to the other. This capability helps to increase the availability of the server.
7. Isolated applications: Server virtualization provides application isolation and removes application compatibility issues.

8. Extending the life of older applications: VM offers hardware emulation providing compatibility with older operating systems so that the older applications are able to run.
9. Helps move things to the cloud: Since the server is built using VM technology, then we can move our server to the cloud easily.

There two types of VM: (1) Virtual Machine Monitor (VMM) and (2) Container. Both will be discussed in more detail in Section 3.2 and Section 3.3 respectively.

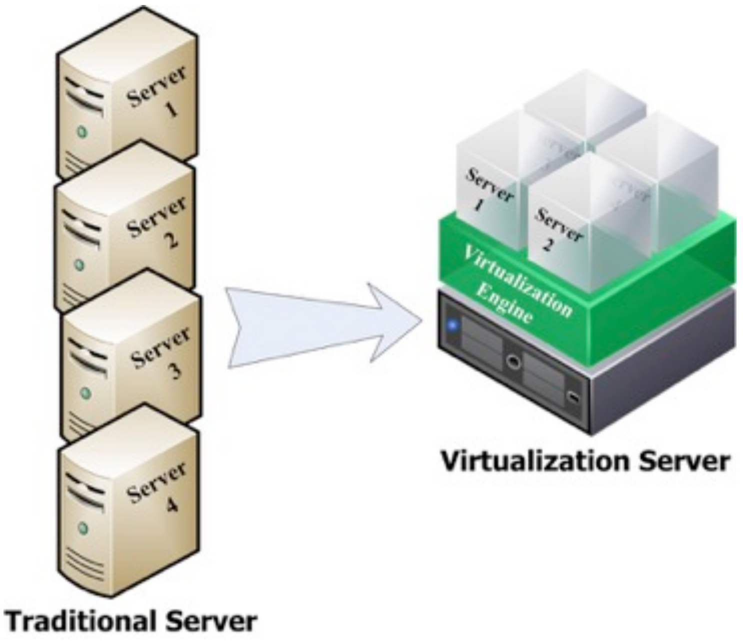


Figure 2. Comparison between traditional and virtualization server architectures.

Nobody can guarantee that a server will always continue to serve information all the time without failure. Since the server runs important services, we should endeavour to protect the server against threats in order to keep it running and serving information to users. With numerous events of server failure damaging large portions of business and the rising risks of information system vulnerability it has become a major concern. A few examples: in October 2016, a distributed denial of service (DDoS) attack targeting DNS service provider (Dyn server) affected a significant portion of Internet access and left many high-profile internet services unreachable for several hours [6]; in September 2016, the information management system of Yorkshire hospital failed and caused hundreds of operations and appointments to be postponed [7]; and Misco news reported that Internet users suffered financial losses due to a malware attack [8].

There are many scenarios of failure that could affect a server's functionality. Further, faults are classified as transient, intermittent, and permanent. Transient faults mean that the fault only occurs once and then disappears. For example, a bee flying through the beam of infrared (IR) transmitter may cause several bits on some network lost. Transient faults typically occur very infrequently, and in most systems, only a few retries are necessary to restore the system. An intermittent fault appears for a while then disappears, this situation occurs repeatedly at intervals. For example, a loose contact on a connector sometimes causes an intermittent fault. These faults are often difficult to identify and repair. Therefore, automatic logging is one way to track the problem. A permanent fault is a failure that only can be recovered when the faulty component is replaced with a healthy component. For example, part of the computer (e.g. processor) crashes. Detail of this failure is described on Table 1.

Table 1. Classification of computer failure

Type	Software	Hardware
Transient	• Invalid input	• A bee flying through the beam of infrared (IR)
Intermittent	• Unhandled exception • Insufficient resources	• Connector and wire bond • Anomaly of ICs behavior • Interference
Permanent	• Hang • Denial of service, DDoS • Malware • Zero-day vulnerability	• Power source • Disk • Mainboard • Other internal components

When we choose to use a virtualization model for our server, it does not mean that our server will be spared from potential failure. Migrating our server model from traditional to virtualization creates a new problem. When a failure occurs on the server, the simplest way to overcome this condition is by resetting it. However, if the server has a complex failure, then a reinstallation of the operating system and all necessary applications may need to be undertaken by the system administrator. In this work, we focus on the permanent failure that occurs in the software area, including hang, DoS and malware. These three failures need to be solved since the number of project bugs [9], which indicates hang, DoS attack [10] and malware [11] are always found every year. In addition, there are several types of DoS: (i) exhausting network resources such as network bandwidth (also known as the DDoS attack), (ii) exploiting a vulnerability, and (iii) exhausting internal server resource (e.g., memory). DDoS attack is more severe than DoS. However, our work only focuses on the resilient server (not the resilient network) that is why we only focus on the DoS attack. In order to detect the failure or anomaly of the system, several tools such as *Tripwire* [12], *ClamAV* [13] and *Chkrootkit* [14] may be used as a detector.

To address the problems that have been described earlier, we were motivated to build a resilient server by combining virtualization technology and a self-repair network (SRN) model [15], which is derived primarily from the concept of an immunity-based system [16]. More specifically, the definition of resilient server is a computer server system that is able to recover from an abnormal (unhealthy) state and return to a normal (healthy) state, and function as it was prior to disruption by observing its own behavior without human intervention (Figure 3). Further, the SRN model is such a model, and we can use the results of the SRN to improve the resilience of the server. The SRN offers a general framework of self-action models for the server to self-recognize, self-repair, and self-replace its own failed parts.

First, we have to design the resilient server types and then start to build the simulation of the resilient server using VMM and container technology. We place a homogenous type of the virtual machine on both the VMM and container as a server. Later, we create a number of limited scenarios of failures that could possibly occur on the server during operation including hang, DoS and Malware. A simple script connects to the detector to identify and respond to the failure.

Second, we implement an SRN model to the resilient server based on the resilient server types that have been designed. There are four models of SRN model including: (i) *self-repair*, (ii) *mutual repair*, (iii) *mixed-repair*, and (iv) *switching-repair*. All these models will be used to build a heterogeneous virtual machine to increase the diversity of the resilient server. Furthermore, we will compare the resilience of the heterogeneous resilient server to the homogeneous resilient server.

Finally, to increase the diversity of the resilient server, we use multiple virtualization engines (VMM and Container) and distribute the SRN manager to all the physical computer servers. The SRN manager works based on the SRN model to recover the failure that occurs during server operation. Further, we will evaluate the performance of the resilient server with an SRN model using benchmark applications.

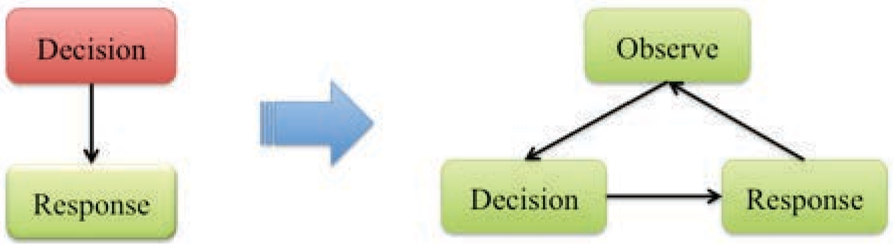


Figure 3. Comparison between conventional and resilient system.
The red color indicates operation by human, the green color operation by machine

1.2. Scope and Objectives

The objective of our research is building a resilient server that can recover from a failure during operation based on a virtualization and SRN model. We define the types of the resilient server by combining three parameters including: (i) application (service), (ii) guest operating system (OS), and (iii) virtualization engines. We limit our research within the scope of server failure and virtualization engines with some assumption as follows:

1. Failure of the server involving:
    a. Hang
    b. Denial of service (DoS)
    c. Malware
2. Virtualization engine:
    a. XEN and KVM as VMM technology.
    b. Linux Container (LXC) as container technology.

1.3. Thesis Achievements

We have successfully built several types of resilient server by combining an SRN model and virtualization technology. We created the resilient server specification using three parameters (layers) of the virtualization including service, guest OS, and virtualization engines. The resilient server is able to recover from generic failures (hang, DoS attack and malware) where other works only focus on specific problems (e.g. only DoS). Furthermore, both the homogeneous and heterogeneous resilient servers are able to recover from failures automatically without human intervention. We also improve the SRN manager by distributing it to all physical servers so that they are able to remedy failures not only in the guest OS but also in the host OS (virtualization engines) so that the downtime of the services can be reduced.

1.4. Thesis Organization

This thesis is structured as follows. First, we discuss the state of the art regarding data center evolution, fault-tolerant systems and existing studies of resilient computing. Next, we introduce our resilient server types and create the simulation of the resilient server using two types of virtualization engine technology, i.e. VMM and container. Afterwards, we explain the scenario of failures that we use to demonstrate the resilience of the server. Then, we explain the SRN model on the resilient server and build homogeneous and heterogeneous resilient servers. Furthermore, we improve our resilient server by involving multiple virtualization engines and distribute the SRN manager that operates to monitor and respond to failures across all the physical servers. Finally, we evaluate the performance of our resilient server and at the end we describe our conclusions and future works.

This page intentionally left blank

2 Related Work

2.1. Data Center Evolution

Collections of computer servers are known as Data Centers (DC). To develop a DC, there are high costs as the DC always has complex requirements for physical space, electricity, a cooling system, security, networking and other essential elements [17]. We need to provide a large area or building for the placement of physical server machines, redundancy safeguards for electricity and networking, and we must secure the area with a high level of security.

However, the DC has been evolving due to the evolution of computers, which are continuously getting faster, smaller, and more efficient [18]. The first models of DC utilized a mainframe server that offered centralization and was in wide use during the 1990s. The second model of DC, known as distributed technology, is where client-server and distributed computation mechanisms are applied. The most recent DC model that we use today is known as virtualization technology where a single physical computer machine hosts multiple virtual computer servers. Figure 4 illustrates data center evolution technology from DC version 1.0 to 3.0.

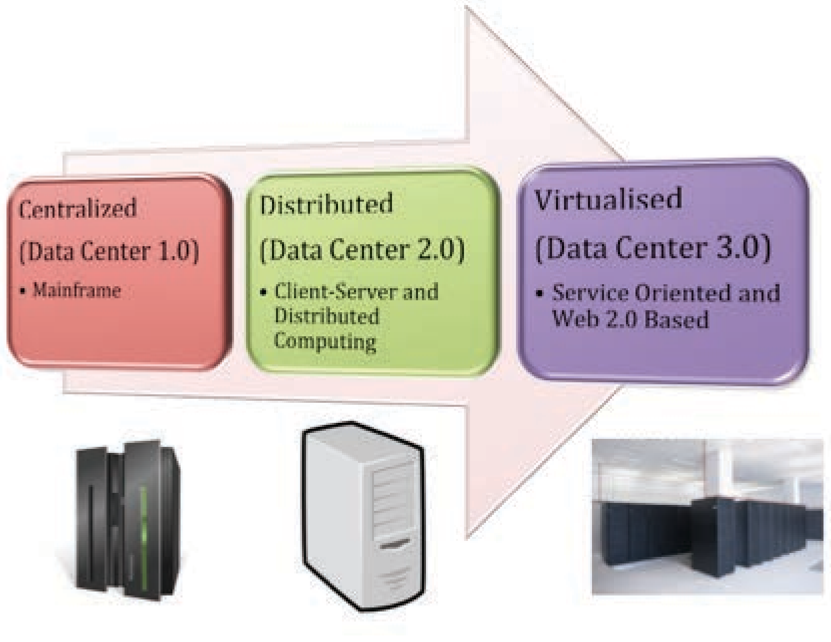


Figure 4. The data center evolution

DC is the core of the information system. The computer server inside the DC will serve all information to the users. Therefore, we need to provide resiliency for the server in the DC. The resiliency of a server is the ability of it to recover quickly and continue operating and serve information even when a failure has occurred. Since DC version 3.0 is applied, then we focus on preserving their existence based on virtualization technology.

2.2. Fault-Tolerant System

There are several ways to make our server work even when there is a system failure. One of the ways is implementing a fault-tolerant system. This system is one of the models that is usually used in a common DC to prevent system stoppages from failures [19] [20] and redundancy is its primary element to achieve a fault-tolerant system [21]. A fault tolerant system is strongly related to dependability and according to [22], dependability covers a number of useful requirements for distributed systems including:

1. Availability: the ability of the systems to provide an idle (available) component to replace the faulty component when the failure occurs.
2. Reliability: the system can work without any problem in intervals of time and a high-reliable system has a long period of time to operate without any problem.
3. Safety: the system provides a safety control when a failure occurs, for example when a computer detects a virus then the network connection to the computer is disconnected automatically to prevent the spreading of the virus to others.
4. Maintainability: how easily a failed component can be repaired. When component fails then it needs to be replaced as soon as possible to continue operation.

A honeypot using N-version programming is one of the implementations of a fault-tolerant system where it runs multiple OSes and web services. This project tried to solve the following problems [23]:

1. Unvalidated input.
2. Broken access control.
3. Broken authentication and session management.
4. Cross Site Scripting (XSS) flaws.
5. Buffer overflows.
6. Injection flaws.
7. Improper error handling.
8. Insecure storage.
9. Denial of service

10. Insecure configuration management.

Further, this project only runs in single virtualization engines called VMware. 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 [24]. This combination can be achieved when there are more than one physical server machine as primary and secondary machines (Figure 5). Another project is called Remus [25]. This project also utilizes a VM to increase the availability and dependability of a server. Remus provides completely transparent recovery from fail-stop failures of a single physical host. Meanwhile, [26] also provides a way to increase the availability of the VM by designing an automatic backup system.

In contrast to the resilient server, the fault-tolerant system focuses on "passive action" where the system keeps functioning without significant changes. Meanwhile, the resilient server focuses on "active action" whereby the system adapts to environmental changes by altering the fundamental structures to preserve their function.

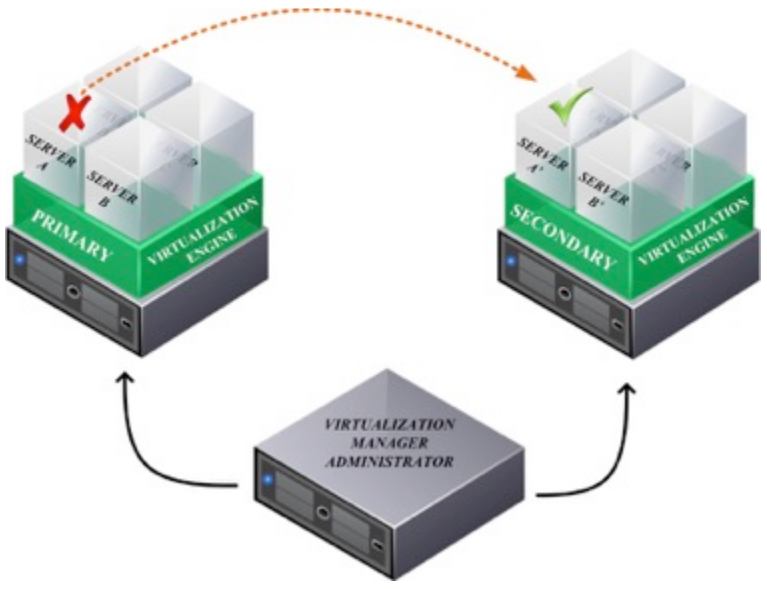


Figure 5. Combination of fault-tolerant system and virtualization.

2.3. Virtual Machine Introspection (VMI)

In virtualization technology, Virtual Machine Introspection (VMI) can be used to realize resilient computing. VMI is a technique to monitor virtual machines (VM) state. The concept of VMI was first introduced in 2003 by Garfinkel and Rosenblum [27]. Moreover, Hebbal et al. [28] classify VMI into three categories: (i) in-VMI, (ii) out-of-VMI delivered and (iii) out-of-VMI derived. However, let simplify the VMI types into two categories: in-VMI and out-VMI. In-VMI is a technique to monitor the guest OS from the inside and reports its activities to take a response. Conversely, out-VMI monitors the guest OS by observing the guest OS activities from outside of the VM. Although VMI research tends to focus on out-VMI due to some weaknesses of host-based monitoring [28], it turns out that in-VMI can bolster intrusion detection [29]. In addition, the VMI technique does not cover the recovery process of the VM since it is only focuses to detect or monitor the condition of the VM.

Table 2. Existing studies of resilient computing

Existing study	Area	Mechanism	Recovery Technique				
			1	2	3	4	5
Glass et al, 2008 [30]	Embedded System	Net. of coordinating nodes	•	•			
Akoglu et al, 2009 [31]		Coordinating master	•	•			
Herder et al, 2006 [32]	Operating System	Reincarnation	•		•		
Pu et al, 1996 [33]		Diversification	•		•		
Shapiro, 2005 [34]		Dedicated OS-services	•				
Adve et al, 2002 [35]	Cross/multi-layer	Resource coordination		•	•		
Yuan et al, 2006 [36]				•	•		
Kant et al, 2004 [37]		Net. management system		•	•		
Corsava et al, 2003 [38]	Agent-based	Agent collaboration		•	•		
Tesauro et al, 2004 [39]		Replaceable agents		•	•		
Fuad et al, 2006 [40]	Legacy appl. and AOP	Checkpoint setting				•	
Haydarlou et al, 2005 [41]		Joint points and advices				•	
Dabrowski et al, 2002 [42]	Discovery System	Shared redundant location inf.			•		
Albrecht et al, 2008 [43]				•			
Subramanian et al, 2008 [44]	Web service and QoS-based	BPEL compensation handler				•	
Baresi et al, 2007 [45]		Supervision framework				•	
Moo-Mena et al, 2008 [46]		QoS and SLA violations		•		•	
Halima et al, 2008 [47]				•		•	
Moser et al, 2008 [48]		QoS and high availability		•		•	•
Sano et al, 2015 [49]	Virtualization-based	QoS and high availability		•		•	
Nagy et al, 2006 [23]		N-version programming			•		
Dunlap et al, 2002 [50]		Logging and replaying transaction		•			
Cully et al, 2008 [25]		Asynchronous replication	•	•			
Winarno et al, 2017		Healthy level of the nodes	•	•	•		

2.4. The Existing System of Resilient Computing

There are many studies that are related to resilient computing. Table 2 shows the existing study of resilient computing which are classified into area, mechanism, and recovery techniques. There are five categories of recovery technique, including (1) replacement, (2) relocation, (3) persistence, (4) redirection, and (5) balancing. Here are several studies of the resilient computing:

2.3.1 Web Service and QoS Based

One of the most common services usually accessed on the Internet is a web service. The biggest challenge to this service is to guarantee a certain quality of service (QoS) towards the expectations of the users. To address this problem, several methods have been proposed. Moo-Mena et al. [46] considered non-intrusive communication flow QoS monitoring for service degradation detection. To recover the service from degradation, they try to redirect the old service to the new service. Halima et al. [47] presented automatic repair for middleware called QoS-Oriented Self-Healing (QOSH) that enhances simple object access protocol (SOAP) messages with QoS metadata to monitor QoS degradation.

2.3.2 Operating System Based

There are two types of fault that possibly occur on an operating system (OS); soft faults and hard faults. Soft faults are focuses on application crashes that can be recovered. While hard faults are focuses on failures caused by hardware failure or faulty drivers often resulting in blocking I/O exceptions. To recover from a hard fault the only way is by a complete system restart. Herder et al. [32], proposed a reincarnation server to solve the failure that occurs on the OS. Similar to the fault-tolerant system, the reincarnation server provides two servers that work as parent and child server. When the parent has a problem, then the child takes its position. Meanwhile, Shapiro [34] proposed dedicated OS-services to anticipate major degradation on OS. To detect OS degradation, log monitoring and log examination are used to diagnose the failure.

2.3.3 Virtualization Based

There are many studies about resilient computing by utilizing virtualization technology. First, the project known as ReVirt [50]. This project enables intrusion analysis through VM logging and replay, which concluded that ReVirt could determine and fix the damage the intruder inflicted by replying the execution before, during and after the intruder compromises the system. ReVirt used a VM with the VMM technology called User Mode Linux (UMLinux) [51] and claimed the time overhead of logging is small. The time overhead was tested using the following applications:

- POV-Ray
- Kernel-build
- NFS kernel-build
- SPECweb99
- Daily use

Since ReVirt is able to replay instruction-by-instruction sequences, it can show an arbitrary observation in detail about what has happened on the system.

Second, Chelladhurai et al. tried to secure the VM from DoS attack where they use Docker as a container [52]. They proposed a solution methodology for Docker against DoS attack. The solution includes: (i) assign memory limit, (ii) assign memory reservation, and (iii) provide default memory value. Detail of the proposed solution methodology is shown in Figure 6. They conducted three experiments for testing their proposed solution methodology. Based on their experiments, they claimed that the test results show that the security and safety of Docker containers can be substantially improved against DoS attacks.

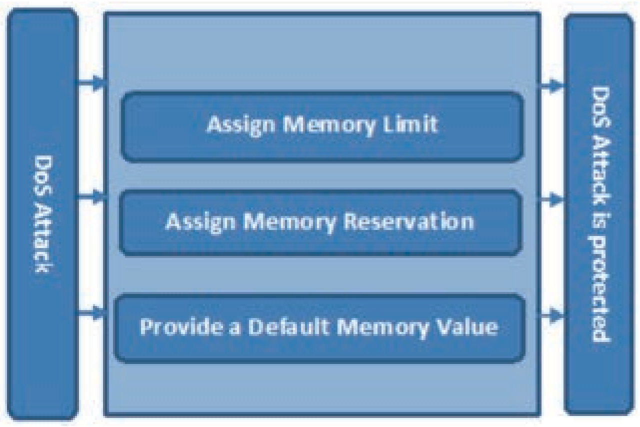


Figure 6. Proposed solution methodology for Docker Container DoS attack [52].

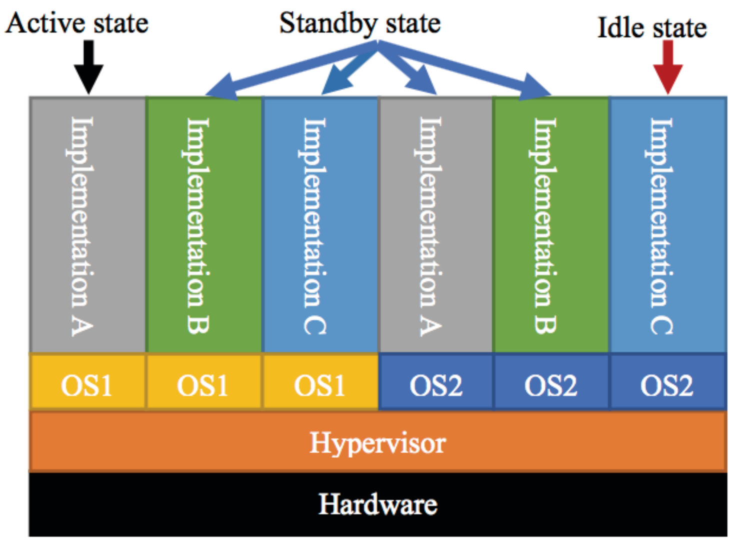


Figure 7. Cyber attack-resilient server design [53].

Third, a cyber attack-resilient server inspired by biological diversity [53]. The concept of this project is based on an artificial immune system where not all the species or individuals are infected by the same infection agent which plays important roles in species survival and adaptability. This project is an implementation of their previous study where they designed an artificial immune server against cyber-attacks

[54]. This project was also inspired from the concept of the diversity in the operating system that was introduced in the Tempo-C specializer of the Synthetix Project [33]. However, Temp-C specializer is focused on the “micro diversity” of kernel components in an operating system, while their project is focused on the “macro diversity” of operating systems and server implementations. Figure 7 shows the proposed design of the cyber-attack-resilient server. There are six VM’s that run the same service that being a DNS service. They explained that the DNS service is vital for the sustainability of other services. Therefore, this service should always be protected. On Windows OS, they equipped with SecondDEP [55] to detect and prevent a zero-day attack if that attack uses a shellcode. Meanwhile, on Linux OS, they equipped AppArmor that restricts the capabilities of programs such as an I/O access.

Different from the cyber-attack-resilient servers that have been described earlier, our study focuses on the resilient server by increasing the diversity of the node where we build and evaluate the implementation of an SRN model that runs on a homogeneous and heterogeneous environment to overcome failures (i.e. hang) that occur during server operation.

This page intentionally left blank

3 Simulating Resilient Server Using Virtualization Technologies

In this section, we will simulate a resilient server using virtualization technologies. When we decide to use a virtualization technology 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, such as [56], [57] and [58]. We simulate a resilient server using virtualization technologies including VMM and container. As VMM, we use XEN and Docker as container. In the simulation, we try to solve several failures that usually occur on the server during their operations.

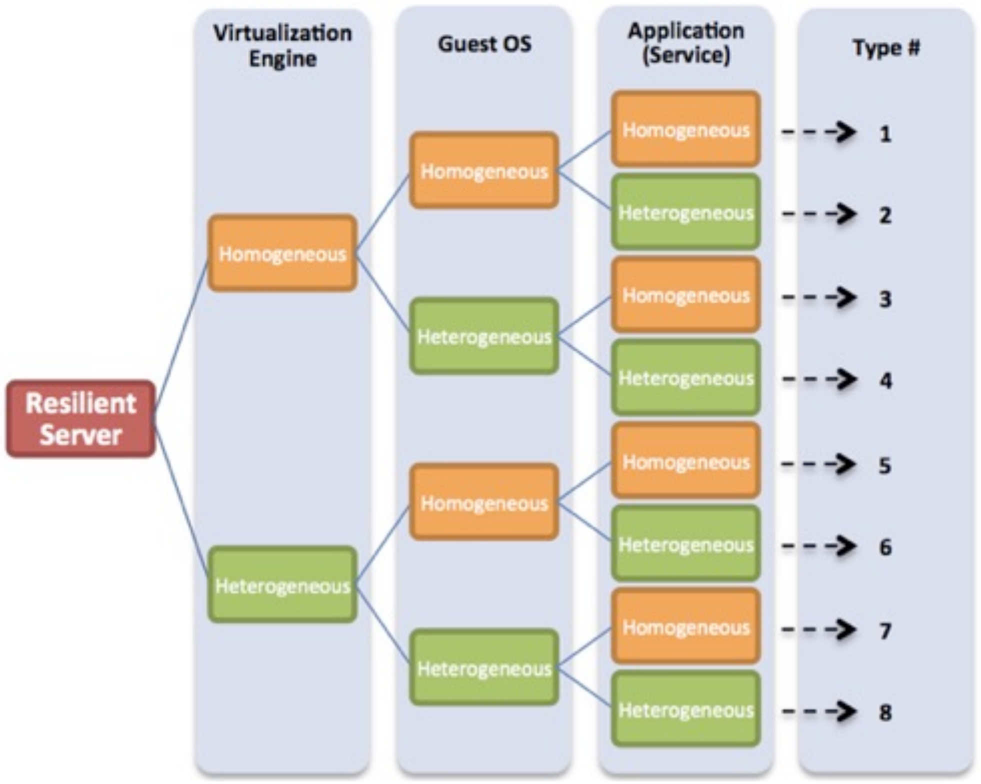


Figure 8. Types of resilient server with three parameter combination.

3.1. Types of Resilient Server

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 in order to increase the diversity of the server. In the biological concept, diversity is the key point of species survival and adaptability where not all species or individuals are infected with the same infectious agent which means that the higher diversity delivers high resiliency of the server. Therefore, 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.

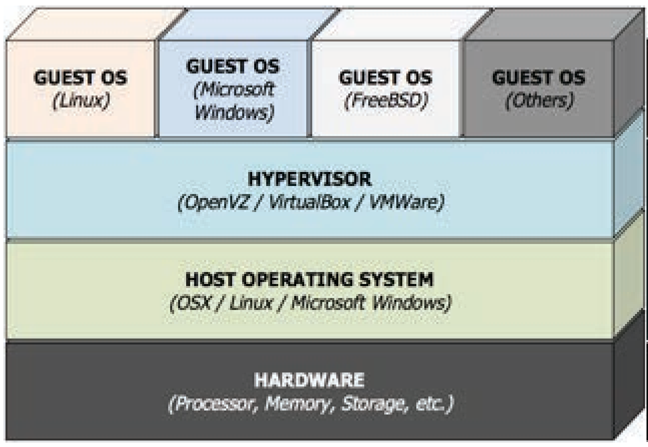
As shown in Figure 8, the combination of each parameter on type #1 to #8 shows that diversity becomes high. For example, type #1 has the same specification (homogeneous) on each parameter so that we can call type #1 as a homogeneous resilient server. Meanwhile, types #2 to #8 have different specifications on each parameter (heterogeneous) so that we can call them heterogeneous resilient servers. We will discuss type #2 to #6 on Section 4 and type #8 on Section 5 in detail. Since type #8 of the resilient server shows that this type is the most heterogeneous combination compared to the other types, consequently, this type is the most resilience. This section only focuses on resilient server type #1. Moreover, simulation design and implementation of resilient server type #1 will be explained further.

3.2. Virtual Machine Monitor

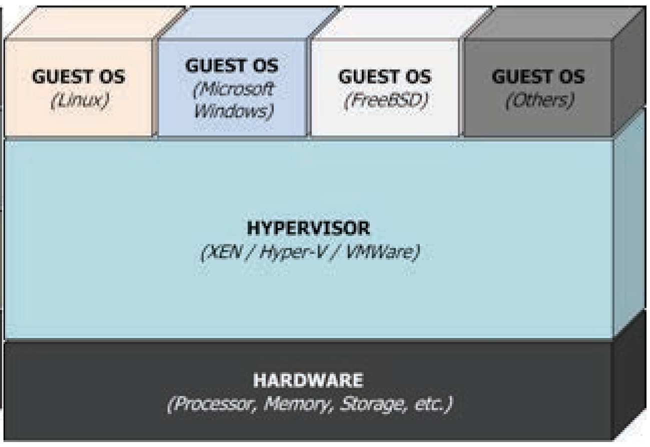
There are two types of VMM (also known as a hypervisor) that can be used for simulations or even used for an operational purpose. The first type is hosted VMM, and the second type is native (bare-metal) VMM. Figure 9 illustrates the differences in the two types of VMM. Table 3 shows an example of the application of the VMM. We used both types of the VMM. We simulate the Native VMM under running hosted VMM using a single computer. In the simulations, we use XEN 4.1.4 as native VMM and VMware Fusion 7.0.0 as hosted VMM.

Table 3. Example of VMM Applications

Type	Application
Native	VMware ESX/ESXi
	KVM
	XEN
	Hyper-V
Hosted	VMware Player
	OpenVZ
	VirtualBox



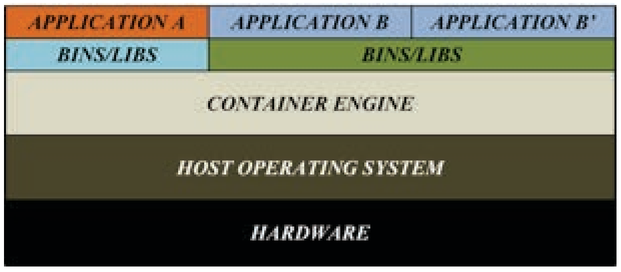
GUEST OS (Linux)
GUEST OS (Microsoft Windows)
GUEST OS (FreeBSD)
GUEST OS (Others)
HYPERVISOR (OpenVZ / VirtualBox / VMWare)
HOST OPERATING SYSTEM (OSX / Linux / Microsoft Windows)
HARDWARE (Processor, Memory, Storage, etc.)

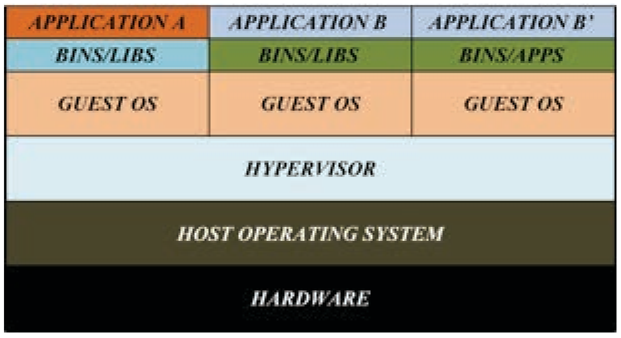


GUEST OS (Linux)
GUEST OS (Microsoft Windows)
GUEST OS (FreeBSD)
GUEST OS (Others)
HYPERVISOR (XEN / Hyper-V / VMWare)
HARDWARE (Processor, Memory, Storage, etc.)

(a) (b)

Figure 9. Comparison between (a) hosted and (b) native VMM.





(a) (b)

Figure 10. Comparison between (a) container and (b) hosted VMM schemes.

3.3. Container

Container (also called a Linux Container) technology has become popular over the last two years [59]. Linux Container (LXC) offers several features and advantages over a VMM in which LXC claims to be faster and more efficient in its use of resources [60] [61] [62]. It uses a different virtualization technique than that of VMM. In particular, the LXC does not simulate the hardware environment. However, the LXC isolates its own domain to execute particular guest programs and acts as if the program is running on a separate system. Figure 10 shows the comparison between the LXC and hosted VMM schemes. OpenVZ, Docker and Wardern are the variants of the LXC. However, LXC has several weaknesses due to:

- Running only on a Linux operating system.
- Can only virtualize applications that run on the Linux OS.

- Uses a shared kernel
- Is vulnerable to the containment environment.

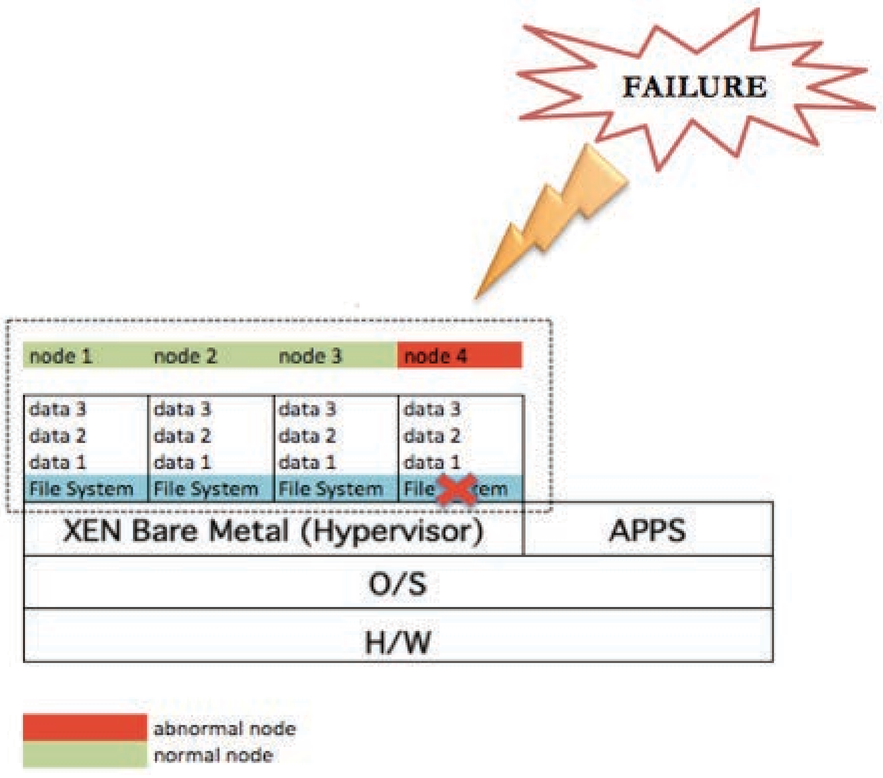


Figure 11. Logical design using VMM.

3.4. System Design and Implementation

3.4.1. Logical Design

We use the logical design to simulate the resilient server using both VMM and container technology. On VMM, each node has four partitions that are stored in separate files (virtual disk). There are 4 VM: node 1 through node 4. We assume that node 1 through node 3 are normal nodes, and the node 4 is an abnormal node (Figure 11). The logical design simply aims at easier recovery of the file system or data of the node when an abnormal condition (an anomaly) is detected. Each node has to install an application (detector) to check the condition of the node itself periodically (Figure 12). There are three detectors that run on each node. When the detector detects an anomaly, it instructs the node to start the recover. However, if the problem still remains, the node has to get the copy of the file system or data from the other nodes.

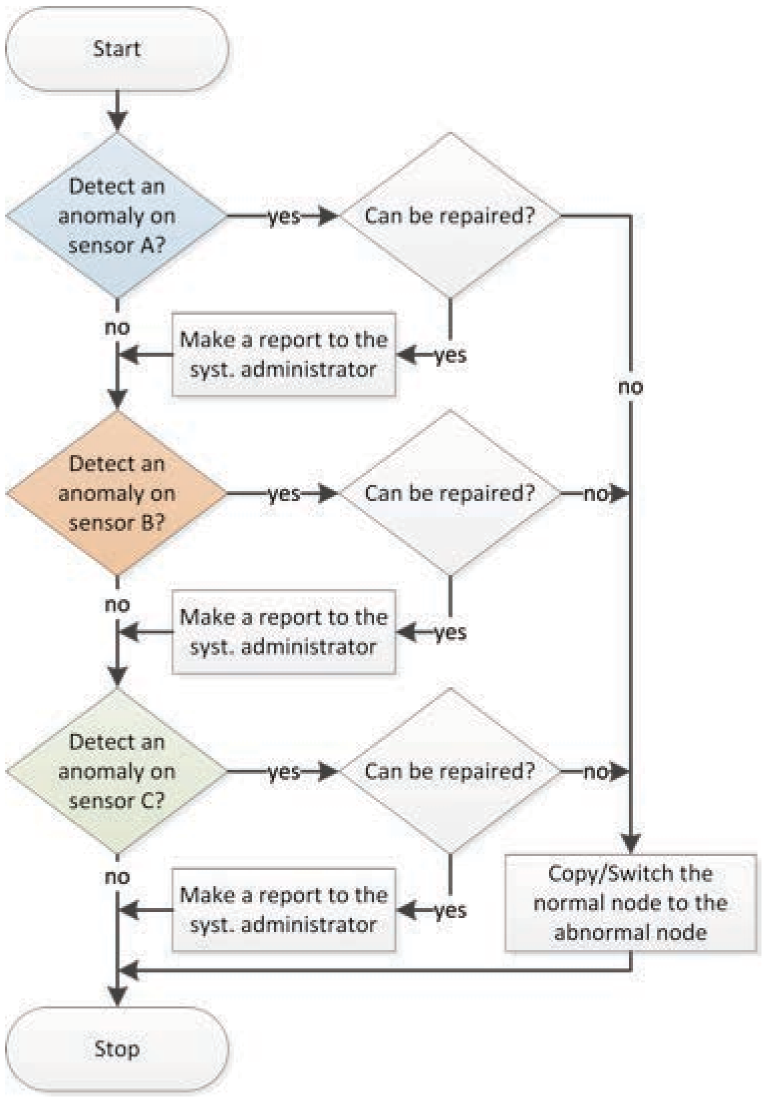

Start
Detect an anomaly on sensor A?
yes
Can be repaired?
no
Make a report to the syst. administrator
yes
no
Detect an anomaly on sensor B?
yes
Can be repaired?
no
no
Make a report to the syst. administrator
yes
Detect an anomaly on sensor C?
yes
Can be repaired?
no
no
Make a report to the syst. administrator
yes
Copy/Switch the normal node to the abnormal node
Stop

Figure 12. Flowchart of a node to check the system periodically.

Meanwhile, on container, we design the simulation environment using Debian GNU/Linux 7 OS and Docker 1.6.2 as LXC. Figure 13 shows the logical design of the simulation. We create a new image (called Debian-ws) that consists of Debian GNU/Linux 7 OS and several additional applications. The additional applications are *Tripwire* (as a detector), Apache2 (as a web server) and a script to be run periodically to detect and to respond the abnormal condition. Docker generates five containers: Debian0 to Debian4.

3.4.2. Scenarios

To simulate the failures that occurs on the node attacks (Figure 11 and Figure 13), we use the following three scenarios:

- Hang scenario: node having faulty (hang),

- Denial of Service (DoS) scenario: node being attacked by DoS,
- Malware scenario: node being contaminated by a virus where contaminated nodes may infect other nodes.

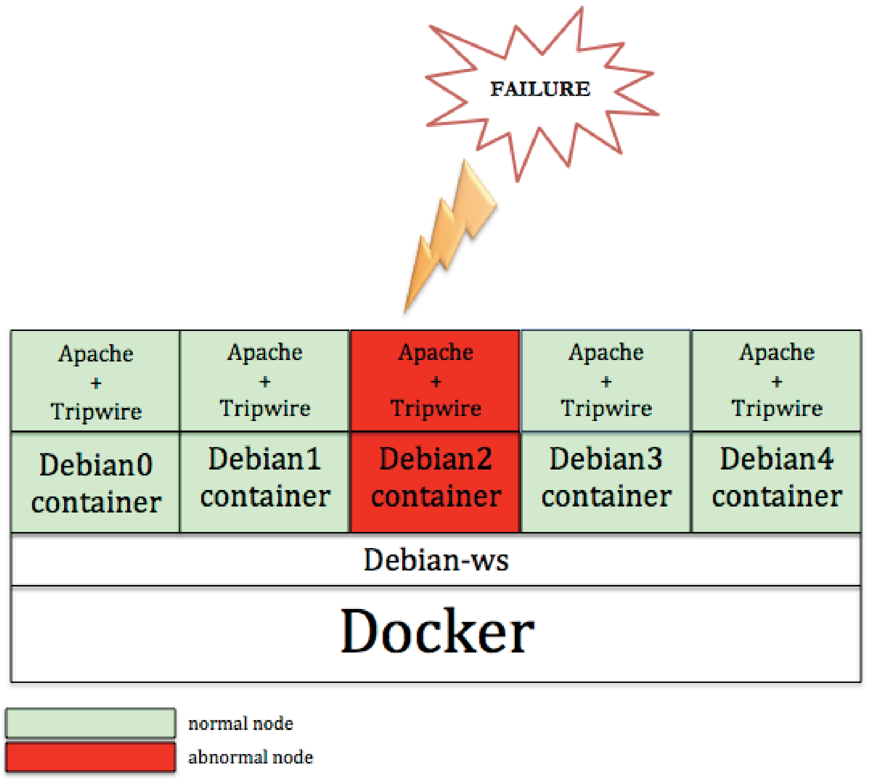
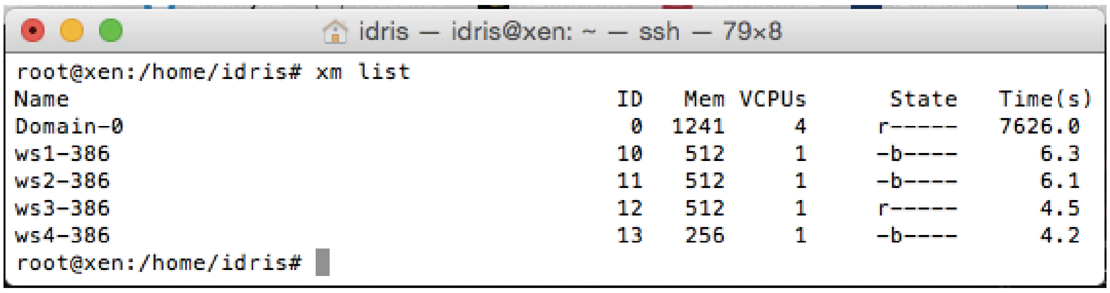


Figure 13. Logical design using container.

For the first scenario: a node hang up, there are many possibilities that can cause the hang problem on the system [63]. It is hard for the node to detect and solve the hang scenario. When hang happens to the server then all the services will be totally stopped. An example of a software bug that causes hangs in the application or OS. In the worse situation, server no longer responds the local or remote instruction. Then, system administrators have no choice except resetting the server manually and it takes time to reset the machine, especially when the server use traditional server. The virtualization engine has been involved to solve this problem. A simple method to detect the hang problem from the network is by pinging the node. When the node is not responding to the ping, the virtualization engine has to restart the node and report to the system administrator.

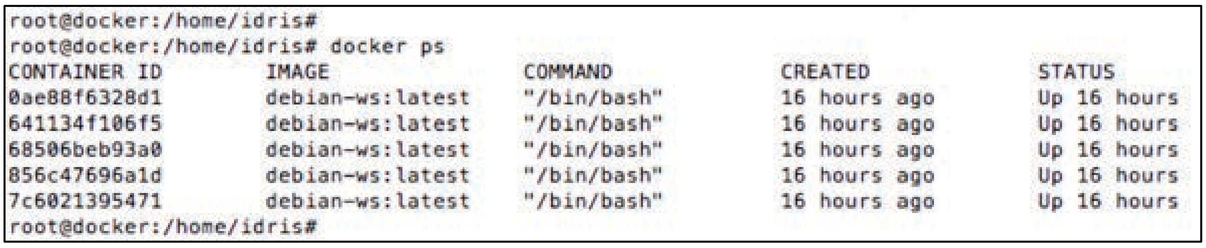
In the second scenario: we use only DoS scenario to focus on the specific services (e.g., web server), which have the vulnerability. In the DoS scenario, we assume the attacker sends the DoS packet to a node. The virtualization engine has to respond by checking the running services and add the IP address of the attacker to the firewall rule. If the services did not respond to the packet send by the virtualization engine, the node has to restart the services by itself. If the problem still occurs, the virtualization engine has to prepare to clean the node by copying from the normal node.

In the last scenario, we assume that a virus infects a node and spreads through the network. Each node has to check the integrity of the system whether the system changed or not. If the virus successfully changed the system and the change is detected by a detector, the node has to copy the file system from the normal node. If the virus caused the overall system down, the hypervisor has to make a decision to isolate or disconnect the contaminated nodes from the network (when the contaminated nodes are identified).



```
root@xen:/home/idris# xm list
Name                                        ID   Mem VCPUs      State   Time(s)
Domain-0                                     0  1241     4     r-----    7626.0
ws1-386                                     10   512     1     -b----       6.3
ws2-386                                     11   512     1     -b----       6.1
ws3-386                                     12   512     1     r-----       4.5
ws4-386                                     13   256     1     -b----       4.2
root@xen:/home/idris#
```

Figure 14. VMM implementation with 4 nodes.



```
root@docker:/home/idris#
root@docker:/home/idris# docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS
0ae88f6328d1        debian-ws:latest    "/bin/bash"         16 hours ago        Up 16 hours
641134f106f5        debian-ws:latest    "/bin/bash"         16 hours ago        Up 16 hours
68506beb93a0        debian-ws:latest    "/bin/bash"         16 hours ago        Up 16 hours
856c47696a1d        debian-ws:latest    "/bin/bash"         16 hours ago        Up 16 hours
7c6021395471        debian-ws:latest    "/bin/bash"         16 hours ago        Up 16 hours
root@docker:/home/idris#
```

Figure 15. Container implementation with 5 nodes.

3.4.3. Implementation

We use XEN version 4.1 as the VMM and Docker version 1.6.2 as container under Debian 7 GNU/Linux. We created four nodes as guest OS with the x86 platform for VMM (Figure 14) and five nodes as container OS for Docker (Figure 15). The virtualization engine has important roles of monitoring all of the nodes and responding if there is an abnormal condition. Hence, the virtualization engine must be capable of checking all of the three scenarios in section 3.4.2.

Moreover, we made the structure of all the nodes identical. The identical nodes allow the inter-node repair (mutual-repair) to be realized by simply copying the normal component of the node to overwrite the component of the abnormal node. To monitor the system, each node must run a program in the background periodically based on the flowchart in Figure 12.

3.5. Simulation Results

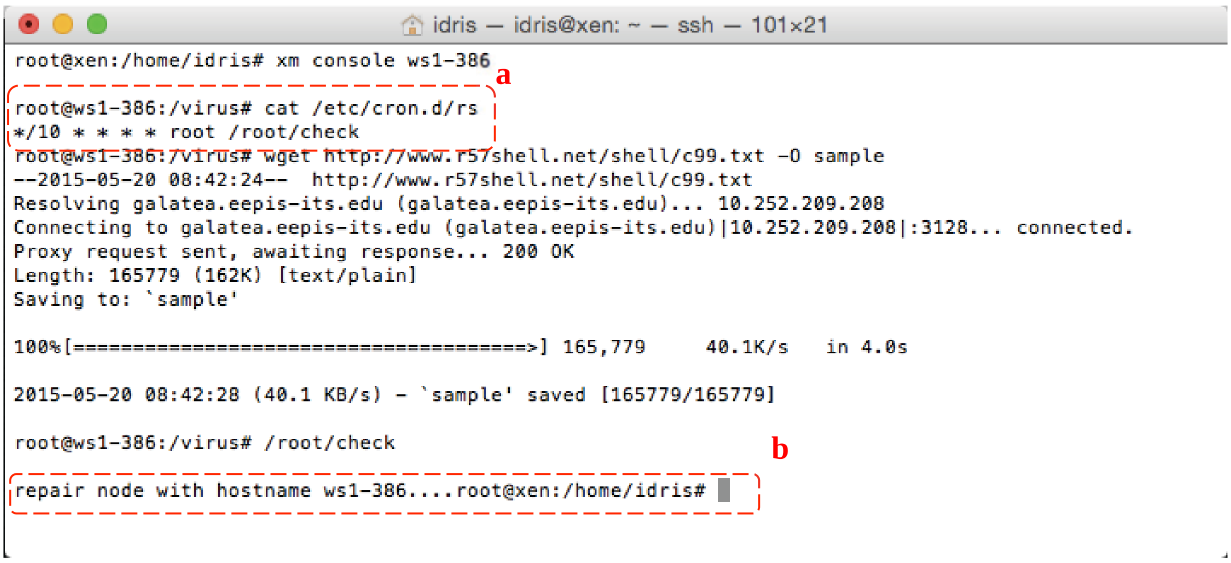
There are a lot more scenarios that can cause malfunction in servers, however, this preliminary simulations focus on the three scenarios: Hang (faulty), Denial of Service (DoS) and Malware.

1. Hang scenario

Through the XEN console we can manage to monitor each node and to simulate the hang condition. Since the nodes are under control of the Linux OS, the OS allows us to simulate the hang scenario by simply sending a halt signal to the OS. When the halt signal is received, however, the node will shut down the system. Thus, we must use another way to simulate the hang scenario: by using XEN command that is "`xm pause <node name>`" command. We run the command for the node 4 and the hypervisor responded by restarting the node 4.

2. DoS scenario

We used *slowloris* script [64] to simulate the DoS scenario. Slowloris is a script that creates many connections to the service. In this scenario, we simulated an attack on the HTTP services (Apache web server). When the script of slowloris is executed to attack the node, the HTTP service ceased responding to the client requests. The virtualization engine checks the services periodically and reacts to the attack by adding the attacker IP address on the firewall rules. With this simulation, DoS attacks demonstrated to hamper a certain services. Further, the other (identical) nodes also have the same vulnerability, suggesting a limitation of homogeneous nodes and a necessity of heterogeneous nodes as well. There are also possibilities to simulate the Distributed DoS (DDoS) attack through this simulation assuming homogeneous nodes involving a framework such as Metasploit framework. Also, heterogeneous nodes must be considered to overcome this scenario since each node can have a distinct OS and service in the heterogeneous environment.



```
root@xen:/home/idris# xm console ws1-386

root@ws1-386:/virus# cat /etc/cron.d/rs
*/10 * * * * root /root/check
root@ws1-386:/virus# wget http://www.r57shell.net/shell/c99.txt -O sample
--2015-05-20 08:42:24--  http://www.r57shell.net/shell/c99.txt
Resolving galatea.eepis-its.edu (galatea.eepis-its.edu)... 10.252.209.208
Connecting to galatea.eepis-its.edu (galatea.eepis-its.edu)|10.252.209.208|:3128... connected.
Proxy request sent, awaiting response... 200 OK
Length: 165779 (162K) [text/plain]
Saving to: `sample'

100%[======================================>] 165,779     40.1K/s   in 4.0s

2015-05-20 08:42:28 (40.1 KB/s) - `sample' saved [165779/165779]

root@ws1-386:/virus# /root/check

repair node with hostname ws1-386....root@xen:/home/idris# 
```

Figure 16. (a) Application run periodically using cron. (b) virus is detected and server (node) is being repair.

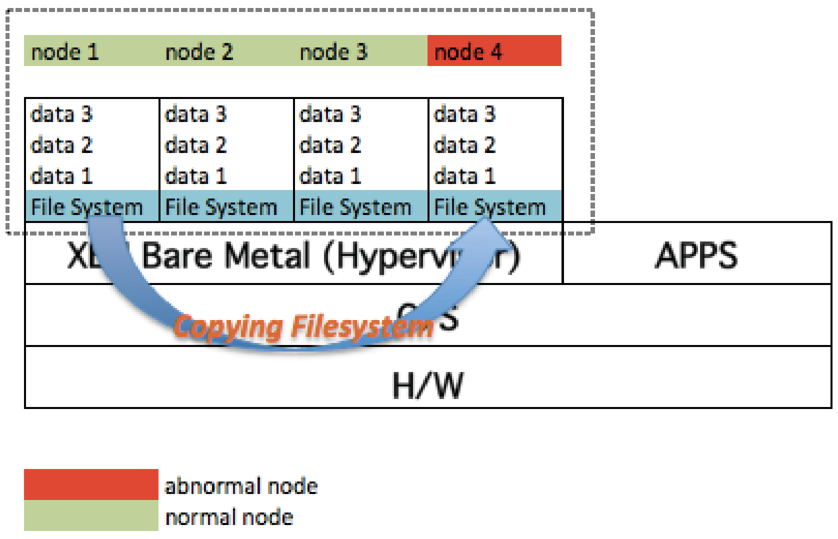


Figure 17. Copying file system from normal node to abnormal node.

3. Malware scenario

In this scenario, we used *ClamAV* and *Tripwire* as detectors to identify malicious codes. We created a shell application to execute *ClamAV* and *Tripwire* to check the node condition. The shell code will execute automatically and periodically by using cron (job scheduler) (Figure 16a). When the malicious code (virus) detected in a node, the node will try to recover or repair the infected files or systems (Figure 16b). If the infection still remains, the node asks to copy the file system from the uninfected node (normal node) (Figure 17). As far as the limited condition in the specific scenario is concerned, the simulation demonstrated to solve the scenario when the nodes are identical (homogeneous node).

4. Execution time

The execution time is starting to count when the broken node is asked for repairing by the healthy node. The counting will be stopped when the broken node has been repaired. Table 4 shows that the execution time of repaired node using VMM is 20.75 seconds on average. Moreover,
Table 5 shows that the execution time of repaired node using container is 2.0 seconds on average. The comparison graph of them is shown in Figure 18.

Table 4. Execution time of the repaired node using VMM.

Hostname	Start	Stop	Duration	Origin
Ws1	1439682933	1439682953	20 sec	Ws4
Ws2	1439682731	1439682752	21 sec	Ws1
Ws3	1439682771	1439682793	22 sec	Ws1
Ws4	1439682821	1439682841	20 sec	Ws2

Table 5. Execution time of the repaired node using container.

Hostname	Start	Stop	Duration	Image
Debian0	1439684766	1439684768	2 sec	Debian-ws
Debian1	1439684769	1439684771	2 sec	Debian-ws
Debian2	1439684773	1439684775	2 sec	Debian-ws
Debian3	1439684791	1439684793	2 sec	Debian-ws
Debian4	1439684795	1439684797	2 sec	Debian-ws

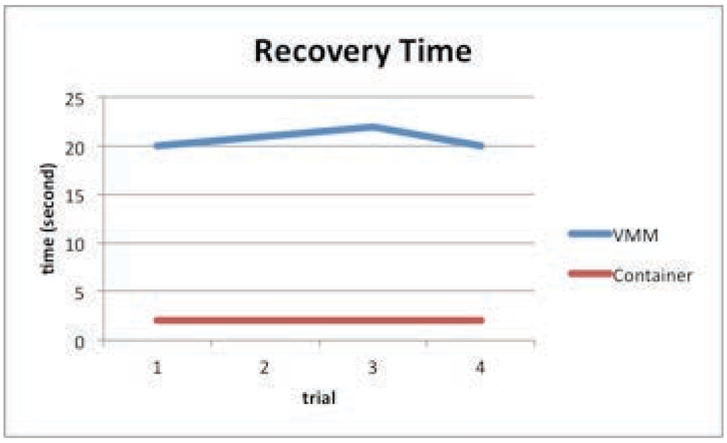


Figure 18. Comparison graph of recovery (repair) time between VMM and container

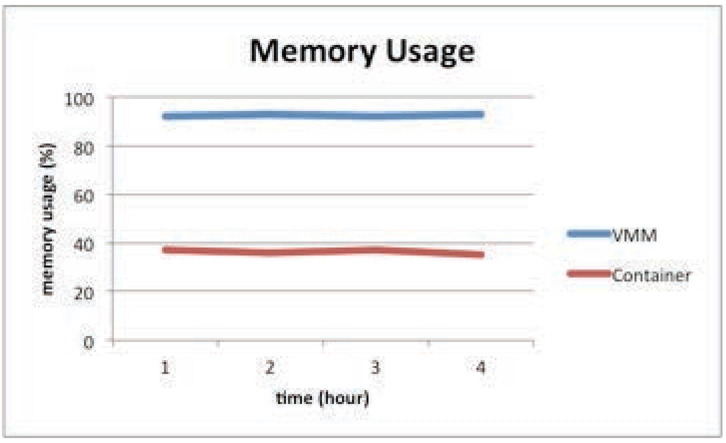
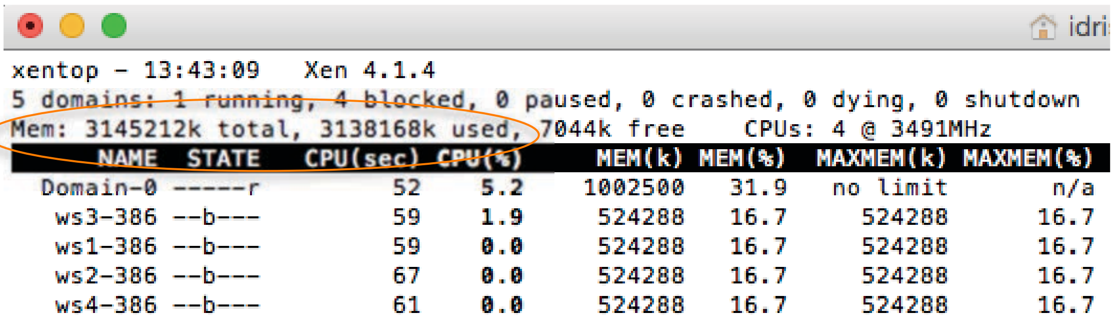
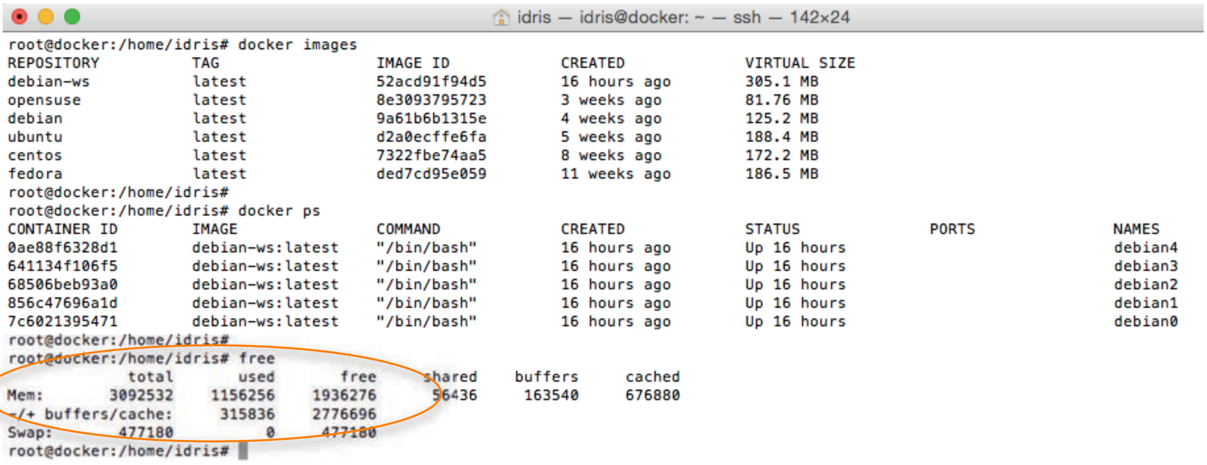


Figure 19. Comparison graph of memory usage between VMM and container.



```
xentop - 13:43:09   Xen 4.1.4
5 domains: 1 running, 4 blocked, 0 paused, 0 crashed, 0 dying, 0 shutdown
Mem: 3145212k total, 3138168k used, 7044k free    CPUs: 4 @ 3491MHz
      NAME  STATE   CPU(sec) CPU(%)     MEM(k) MEM(%)  MAXMEM(k) MAXMEM(%)
  Domain-0 -----r         52    5.2    1002500   31.9   no limit       n/a
   ws3-386 --b---         59    1.9     524288   16.7     524288      16.7
   ws1-386 --b---         59    0.0     524288   16.7     524288      16.7
   ws2-386 --b---         67    0.0     524288   16.7     524288      16.7
   ws4-386 --b---         61    0.0     524288   16.7     524288      16.7
```

Figure 20. Memory usage of XEN (VMM) with four guest OSs



```
root@docker:/home/idris# docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             VIRTUAL SIZE
debian-ws           latest              52acd91f94d5        16 hours ago        305.1 MB
opensuse            latest              8e3093795723        3 weeks ago         81.76 MB
debian              latest              9a61b6b1315e        4 weeks ago         125.2 MB
ubuntu              latest              d2a0ecffe6fa        5 weeks ago         188.4 MB
centos              latest              7322fbe74aa5        8 weeks ago         172.2 MB
fedora              latest              ded7cd95e059        11 weeks ago        186.5 MB
root@docker:/home/idris#
root@docker:/home/idris# docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS               NAMES
0ae88f6328d1        debian-ws:latest    "/bin/bash"         16 hours ago        Up 16 hours                             debian4
641134f106f5        debian-ws:latest    "/bin/bash"         16 hours ago        Up 16 hours                             debian3
68506beb93a0        debian-ws:latest    "/bin/bash"         16 hours ago        Up 16 hours                             debian2
856c47696a1d        debian-ws:latest    "/bin/bash"         16 hours ago        Up 16 hours                             debian1
7c6021395471        debian-ws:latest    "/bin/bash"         16 hours ago        Up 16 hours                             debian0
root@docker:/home/idris#
root@docker:/home/idris# free
             total       used       free     shared    buffers     cached
Mem:       3092532    1156256    1936276      56436     163540     676880
-/+ buffers/cache:     315836    2776696
Swap:       477180          0     477180
root@docker:/home/idris#
```

Figure 21. Memory usage of Docker with five containers.

5. Memory usage

We allocate 3GB physical memory for each simulation for VMM and container. Figure 20 shows that the VMM was running four guest OSs (ws1-386 to ws4-386) and consumed more than 90% of physical memory, while container was running five containers (Debian0 to Debian4) and consumed only approximately 37.4% of physical memory (Figure 21). From the simulation results (the execution time and the memory usage), container performance is faster and more efficient of resource (memory) rather than VMM. This is because container which shares a kernel. The shared kernel on the container could reduce the startup time. Nevertheless, VMM offers more security since it needs a modified kernel [60]. The comparison graph of memory usage is shown in Figure 19.

As far as the limited simulations are concerned, the repair process and the copy process that described in Figure 17 are the model of SRN, which involved to the virtualization technique. However, homogeneous node may not be a sound assumption. For example, in the malware scenario, all the nodes are equally vulnerable to the same viruses. Moreover, it would be difficult to avoid the same infection even after the infected node is cleaned up. To overcome this problem, we can use the heterogeneous nodes (each node has a distinct operating system or even distinct services). The heterogeneous node could be implemented by installing different distribution operating system (e.g., CentOS and Debian GNU/Linux) and services (e.g., Apache and Nginx). By implementing the heterogeneous node, it is more difficult for malicious codes to exploit the system, for they have to find the vulnerability each of the heterogeneous nodes.

3.6. Concluding Remarks

Simulations revealed that the self-repair network with homogeneous nodes can be realized by involving the virtualization techniques where the self-repair is executed by copying the content of the homogeneous node. Simulations also suggested that the servers with the homogeneous environment can be made resilient against failures in limited scenarios. Although the self-repair network can deal with a specific type of failures to a limited scale of failures, the self-repair network alone cannot even recognize a large-scale failures, and recovery is also limited if we restrict ourselves to mutual repairing between homogeneous nodes. We also suggest that the diversity created by heterogeneous nodes involving a self-reconfiguration model that allows a system to replace failed nodes with heterogeneous nodes and to protect against ever growing threats involving diversity (learned from the immune system) can be a solution to keep resilience against unknown type of failures.

We also conclude that Docker, which is implementing container technology, outperforms XEN, which is implementing VMM technology. Container takes 2 seconds to recover the abnormal node while VMM takes 20.75 seconds longer. VMM consumes much memory than container where more than 90% is used by VMM while container only use approximately 37.4%. However, we have to consider that one of the weaknesses of the container is only running in the Linux environment. Furthermore, to protect against growing threats, we have to involve the diversity by creating heterogeneous nodes and involving a self-reconfiguration model.

4 A Resilient Server With A Self-Repair Network Model

After simulating a resilient server using virtualization technology in Section 3, we continue to build a resilient server by involving the SRN model. We enhanced our work by increasing the diversity of the service and guest OS where we created a heterogeneous of service and guest OS. In other words, we build a resilient server type #4. As a result of the work, we compare the resilient server with a homogeneous and heterogeneous environment.

4.1. Self-Repair Network Model

The resilient server that we previously simulated on Section 3 is equipped with a simple script. In this section, we will enhance the script that we called as an SRN Manager and it will implement 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:

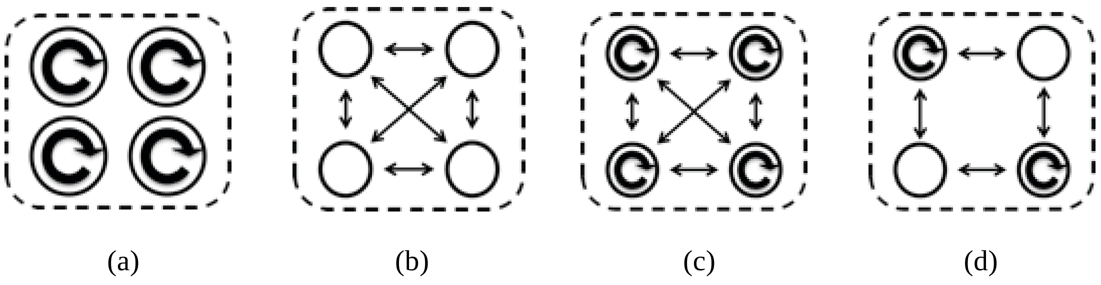


Figure 22. (a) Self-repair, (b) Mutual-repair, (c) Mixed-repair, (d) Switching-repair. An arrow indicates the direction from a repairing node to a node being repaired.

4.1.1. Self-repair model

The first model is *self-repair* model. This model has an ability to repair a failure that occurs on the nodes by itself (Figure 22a). 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 [65]. 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.

4.1.2. Mutual-repair model

The second model of SRN is a *mutual-repair* model that has an ability to repair the other nodes (Figure 22b). 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 [15].

4.1.3. Mixed-repair model

This model contains a combination of two basic models of the SRN including the *self-repair* and *mutual-repair* model (Figure 22c). In other words, a *mixed-repair* model that 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 [66] [67]. 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*.

4.1.4. 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 (Figure 22d). In 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.

Table 6. Partition table of each node.

Disk partition	Mount point	Size (GB)	Disk filename	Usage
1	/	1	data1.img	file system
2	/var	0.5	data2.img	variable data
3	/usr	0.5	data3.img	usr-land programs and data
4	/home	0.5	data4.img	home directory

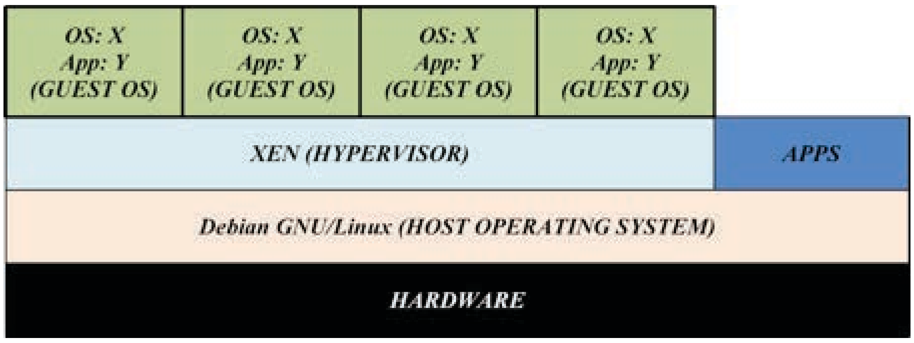


Figure 23. Logical design of resilient server with a homogeneous environment.

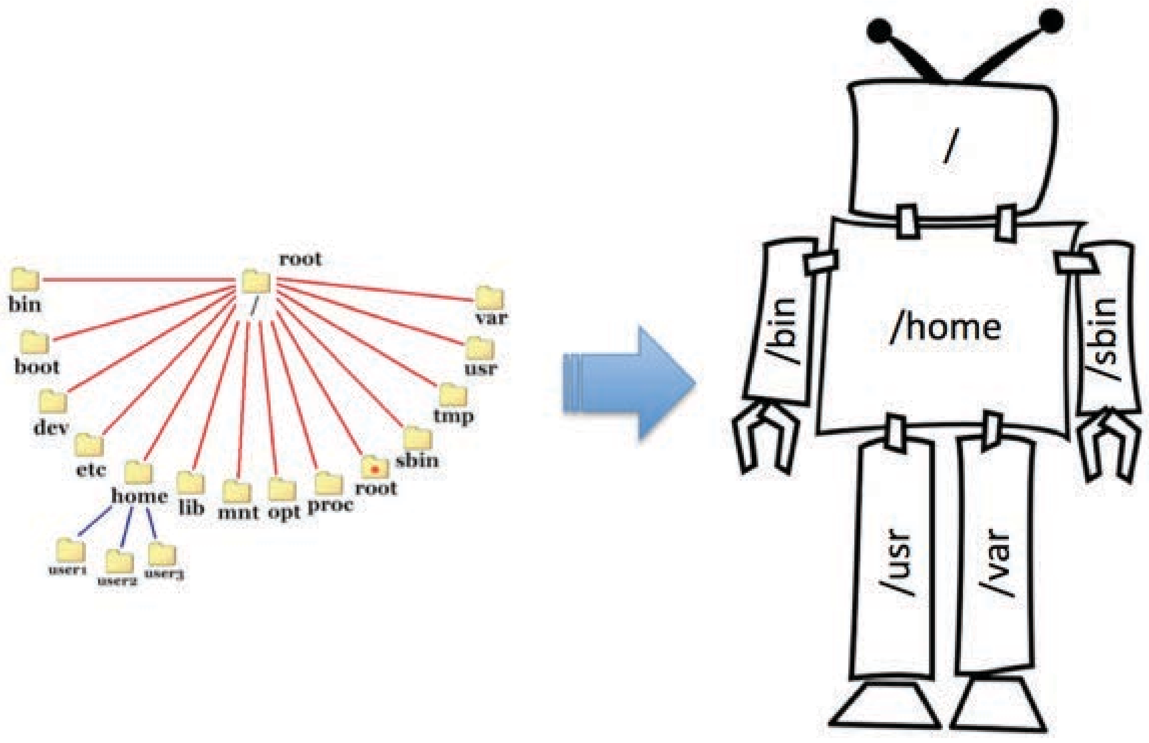


Figure 24. An illustration of file hierarchy system of Linux OS to the humanoid robot.

4.2. Resilient Server with a Homogeneous Environment

The definition of the resilient server with a homogeneous environment means that we are running the server on the virtualization with the same (homogeneous) application types, the guest operating system (OS), and virtualization engine (i.e. VMM and container). In other words, we implement the resilient server type #1 that already discussed in Section 3.1. The logical design of this type can be shown in detail in Figure 23. This type offers flexibility to copy their faulty component from the other node compared to other types.

As the illustration of how the resilient server with a homogenous environment has the possibility to change or replace its broken part from the normal node, Figure 24 shows the file hierarchy system of Linux OS where it has a few directory such as *bin*, *boot*, *dev* and others. Assume that these directories can be placed into the humanoid robot. Further, when one of humanoid robot parts has a problem, then we can easily replace it with a new part as long as the part is compatible with the existing humanoid robot.

To realize the illustration of Figure 24 to the resilient server, we have to create the servers (nodes) with same specifications such as memory and storage disk capacity since we are focusing on a homogeneous environment. We define each node by giving them storage that contains separate disk partitions as shown in Figure 25. Further, this figure also shows an abnormal node and normal node where an abnormal node has an infected file in one or more of the disk partitions. Moreover, we can store the disk partition as a file (Table 6) so that we can easily copy the component from one node to another when the failure occurs.

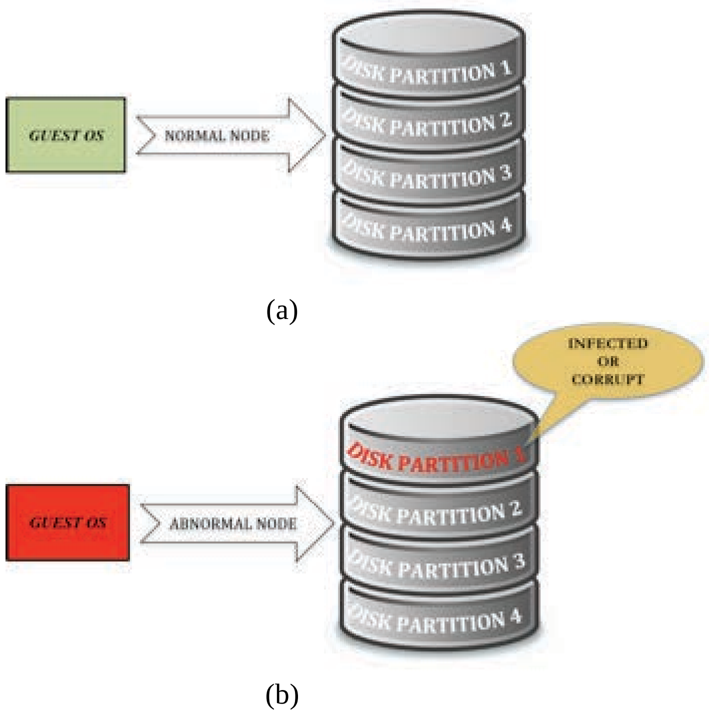


Figure 25. Disk partition design of resilient server with a homogeneous environment: (a) normal node, (b) abnormal node.

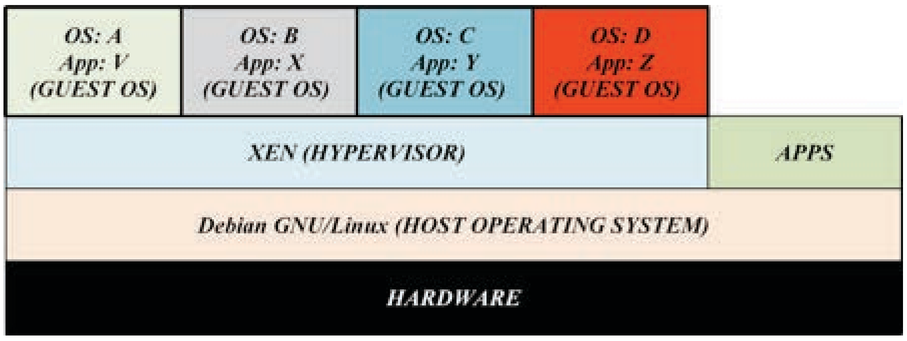


Figure 26. Logical design of resilient server with a heterogeneous environment.

4.3. Resilient Server with a Heterogeneous Environment

In contrast to the resilient server with a homogeneous environment, in the resilient server with a heterogeneous environment, we have to make the nodes different to each other. In this design, we distinguish each node by the operating system and the application running specific services (e.g. web server or DNS server) as described by the resilient server type #4. For example, node X is running a Linux operating system and an Apache web server. Meanwhile, node Y runs a Windows operating system and an IIS web server. The logical design of this resilient server is shown in Figure 26. Further, the resilient server type #8 will discuss in detail in Section 5.

The resilient server with a heterogeneous environment is more complex than the resilient server with a homogeneous resilient server. We cannot change or copy the broken component from the other nodes since they have a different OS and application. Therefore, we cannot apply *mutual-repair* to the resilient server with a heterogeneous environment.

The biggest challenge of the resilient server with a heterogeneous environment is how to synchronize the information among the nodes. The synchronization process is very vital to continue the services to the user with consistent information. Before the normal nodes become abnormal due to a component failure, the information should be delivering to the replacement node. The problem is how the OS and application on the replacement node able to accommodate the information from the abnormal node so that the replacement node has consistent information as the abnormal node.

4.4. Healthy Level of The Nodes

We also called the server that runs on the virtualization environment as a node. This node will implement SRN model to address the failure that already explains in the previous section (Section 4.1). In order to implement the SRN model, each node has to know its condition (healthy level) or others before the repairing action is decided. The healthy level value is between 0 and 1. More higher the value then the node is more healthy. Therefore, each node has to measure their healthy level by using the detector (Figure 12) that installed to detect the failure. When the detector detects abnormal condition then the healthy level will be decreased and vice versa. Further, we have to define the threshold value of the healthy level. The threshold is used to determine when the repairing process is started.

Algorithm 1. Pseudo-code implementation of the script for hang scenario

```
R ← 1
threshold ← 0.5
repeat
    if receiveIcmpRequest = FALSE then
        if R < threshold then
            resetTheHangNode()
        end if
        if R > 0 then
            R ← R - 0.1
        end if
    else if R < 1 then
        R ← R + 0.1
    end if
until scriptIsStopped
```

4.5. System Design and Implementation

To simulate the failures that occur on the virtualization with the resilient server with a homogeneous and a heterogeneous environment, we define the three scenarios similar to Section 3.4.2 as follows:

a) Hang scenario: a node partially or totally stops functioning

b) Denial of Service (DoS) scenario: a node being attacked by a DoS

c) Malware scenario: a node being infected by a virus where the contaminated node may infect other nodes

Therefore, we need to install an application as SRN manager that operates as a detector. The detector will detect and recognize an abnormal condition present on the nodes. These detectors are inspired by an immunity-based system [68] when the healthy level (*R*) values of the detector is 0 to 1. ICMP, TCP and UDP ping are examples of the network transport that we can use to create a script to detect the existence of the nodes. Further, in addition to installing the detector on the host OS or virtualization engine, we have to install the detector on each of the nodes (e.g. anti-virus and *Tripwire*). When the detectors identify an abnormal activity, the virtualization engine and the nodes respond to the detector alert by initiating the SRN model (e.g. *self-repair* or *mutual-repair*) which is followed by sending a report to the system administrator. The detectors are executed by the script that runs the hang, DoS and malware scenarios (Algorithm 1, Algorithm 2 and Algorithm 3, respectively). We need to evaluate the node condition (*N*) as to whether the node is getting faulty (value=1) or not (value=0) by checking the healthy level ($0 \le R \le 1$) with the threshold as shown in Equation 1. In this simulation we assume that the healthy level threshold value is 0.5. The assumption of the healthy level threshold value refers to the middle value where if the value is too high then the node is easily determined as the faulty node and vice versa.

4.5.1. Hang scenario

There are many potential activities that could trigger a hang condition. However, for this simulation we created a script that runs as a detector to check the node condition as to whether the node is in normal condition or in abnormal (hang) condition by utilizing an ICMP ping. This script is executed by the host OS or virtualization engine as a *daemon*. Algorithm 1 shows the algorithm of the script and the flowchart of this algorithm is shown at Appendix 1. To simulate the hang scenario on the nodes, we use the `halt` command to the node to create a hang condition. When the script did not (value=0) receive the ICMP ping (*I*) request from the node within a particular period, it makes the healthy level (*R*) value decrease 0.1 and vice versa (Equation 2). The host OS or virtualization engine will assume that the nodes are in the halt condition when the *R* value is lower than the threshold and the virtualization engine resets the node by restarting the node (intra-repair). After the resetting node becomes normal, the healthy level of the node will increase along with the resetting node sending the ping request to the virtualization engine.

$$N = \begin{cases} 0, & R > 0.5 \\ 1, & R \le 0.5 \end{cases} \tag{1}$$

$$R = \begin{cases} R - 0.1, & I = 0 \\ R + 0.1, & I = 1 \end{cases} \tag{2}$$

$$R = \begin{cases} R - 0.1, & I_c \ge 300 \\ R + 0.1, & I_c < 300 \end{cases} \tag{3}$$

Algorithm 2. Pseudo-code implementation of the script for DoS scenario

```
$R \leftarrow 1$
$threshold \leftarrow 0.5$
repeat
    if $Nc \geq Ic$ then
        if $R < threshold$ then
            addTheSuspectedIpToFirewall()
        else
            removeTheSuspectedIpFromFirewall()
        end if
        if $R > 0$ then
            $R \leftarrow R - 0.1$
        end if
    else if $R < 1$ then
        $R \leftarrow R + 0.1$
    end if
until scriptIsStopped
```

4.5.2. DoS scenario

DoS is one of the various techniques often used by attackers to interfere with specific services (e.g. web server). It is difficult to overcome this attack, since DoS works by flooding traffic directly to the service and causing abnormal operation. We used the *slowloris* [64] script to simulate a DoS attack to the web server. In this scenario, we create a script by utilizing the `netstat` command to check the number of connections on a particular service and whether the connections are over the limit or not (Algorithm 2 and Appendix 2). When the script detects the number of connections (N_c) from an IP address that is accessing particular services is higher than the threshold of the maximum number of connections (I_c) within a certain period, it makes the healthy level (R) value decrease 0.1. However, if the N_c from the same IP address is detected lower than the threshold of I_c on the next check, it will make the R value of the IP address increase 0.1 (Equation 3). When the R value of the suspected IP address is lower than the threshold, the script will add the IP address to the firewall list using the `iptables` command (intra-repair) and vice versa.

Algorithm 3. Pseudo-code implementation of the script for malware scenario

```
R ← 1
threshold ← 0.5
repeat
    if detectMalware = TRUE then
        cleanTheMalware()
        Rp ← partitionDiskInfection()
        R ← R - Rp
        if R < threshold then
            isFound ← findNormalNodeWithDifferentApplication()
            if isFound = TRUE then
                switchTheAbnormalNodeWithStandbyNode()
            else
                isolateTheNode()
            end if
        end if
    else
        if R < 1 then
            R ← R + 0.1
        end if
    end if
until scriptIsStopped
```

4.5.3. Malware Scenario

Since we separate the file system and other paths (Table 6) into the different partition disk (Figure 25), we could repair the contaminated partition disk in two ways: (1) recovering by antivirus, and (2) switching with the normal node. We utilized anti-virus and *Tripwire* to detect the abnormal activity that was caused by the viruses. When the antivirus or *Tripwire* detects the virus, then the antivirus tries to remove the virus and inform the script that is running on the virtualization engine.

The script will then update the healthy level (R) value of the node while the antivirus is trying to remove the virus. We assume the healthy level ($0 \le R \le 1$) of the node based on the partition disk of the nodes (R_p) (Table 7). The nodes are healthy (normal) when the R value is 1 and vice versa. After the R values have been updated by the script, then the script checks whether the antivirus has succeeded or not. If the R value is lower than the threshold, the script continues to search the normal node (R=1). If there are no normal nodes then the script will isolate (turn off) the infected node in order to reduce the risk of the virus spreading to the other nodes. However, if the script found the normal node with a different application, then the abnormal node will be switched with the standby node (Algorithm 3 and Appendix 3).

Table 7. The healthy level value of the partition disk.

Disk Partition ID#	**Mount point**	**Healthy level of disk (*Rp*)**
1	/	0.2
2	/var	0.2
3	/usr	0.2
4	/home	0.4

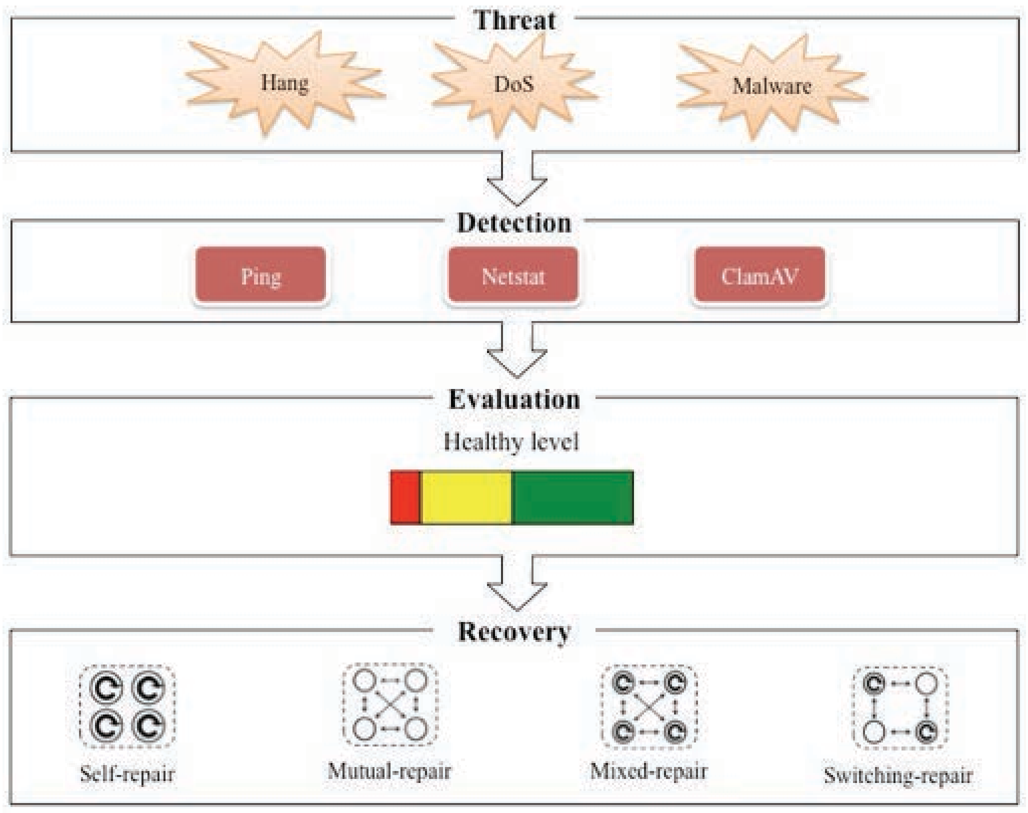


Figure 27. Implementation of the resilient server with an SRN model.

4.5.4. General Implementation

The description of general implementation of the resilient server with an SRN model is shown in Figure 27. The implementation starts with failure (e.g., hang, DoS, and malware) to the server then the detector will identify the failure. For example, ping will be used to identify the hang failure, netstat will be used to detect the DoS attack and ClamAV will be used to detect the malware. The result from the detector will affect the healthy level of the node as described in Algorithm 1, Algorithm 2 and Algorithm 3. When the healthy level is higher than the threshold then the repairing process will be applied to the failed node. For example, on hang condition where the application (service) cannot respond user request due to software bug (e.g., CVE-2014-0098 [69]) then *the self-repair* is chosen to reset the application.

4.6. Experimental Results

Using the scenarios that we defined, we simulated them on the homogeneous and heterogeneous environments as follows:

4.6.1. Hang Simulation

In either the resilient server with a homogenous or heterogeneous environment, when the node does not send the ping request on the particular node to the virtualization engine then the script will decrease the healthy level value of the node (*R*) and vice versa. Table 8 and Table 9 show the test case result from the hang scenario. Figure 28 and Figure 29 shows the detail of the experiment result of hang failure simulation. Some nodes may not send the ping request to the virtualization engine due to heavy load processes occurring inside the node for a certain period (Table 8, node id #2 and #4). However when the nodes return to a normal state then the *R* value will increase on the next check. All of the nodes in Table 8 show that the *R* value is lower than the threshold and consequently this condition will trigger the virtualization engine to reset the node.

Table 8. Test cases of hang scenario (step 7) on the resilient server with a homogeneous environment.

Node ID #	**OS**	**IP Address**	**Healthy level**	**State**	**Action**	**SRN Model**
1	M	192.168.0.1	0.4	X	Reset	*self-repair*
2	M	192.168.0.2	0.4	X	Reset	*self-repair*
3	M	192.168.0.3	0.4	X	Reset	*self-repair*
4	M	192.168.0.4	0.4	X	Reset	*self-repair*

Table 9. Test cases of hang scenario (step 7) on the resilient server with a heterogeneous environment.

Node ID #	**OS**	**IP Address**	**Healthy level**	**State**	**Action**	**SRN Model**
1	D	192.168.1.1	1.0	O	-	-
2	R	192.168.1.2	0.9	O	-	-
3	M	192.168.1.3	0.4	X	Reset	*self-repair*
4	S	192.168.1.4	0.8	O	-	-

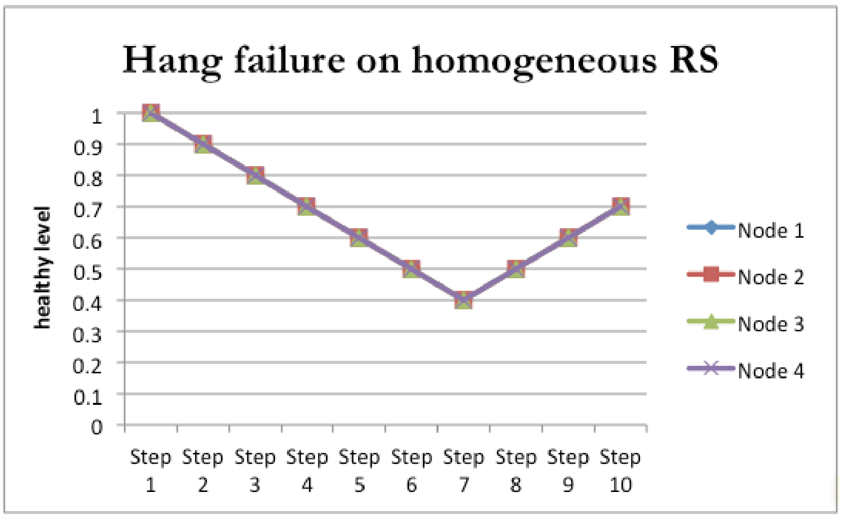


Figure 28. Result of hang failure on the resilient server with a homogeneous environment.

When the node changes from an abnormal (X) to a normal (O) state after the reset process then the *R* value will increase on the next check. Since the resilient server with a homogeneous environment has the same structure (OS), the probabilities of the hang faulty recovery are the same (Table 8). On the other hand, the resilient server with a heterogeneous environment has different probabilities for a hang faulty recovery (Table 9).

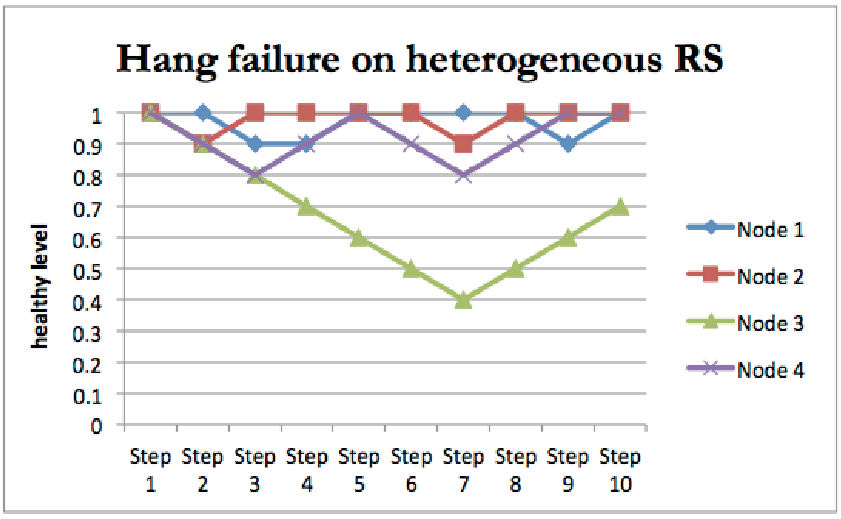


Figure 29. Result of hang failure on the resilient server with a heterogeneous environment.

4.6.2. DoS Simulation

In order to simulate a DoS attack, we use *slowhttptest* as an attacking tool to the web server. The simulations are run on both the resilient server with homogenous and heterogeneous environments. The first action of this simulation is to define the target IP address of the web server. When the *slowhttptest* is executed, it creates multiple concurrent connections to the web server and makes the web server busy or operationally hampered. The nodes that are being attacked are running a script (Algorithm 2) that utilizes the `netstat` command to detect unusual connections coming from a particular IP address. Table 10 and Table 11 shows that both of the resilient servers with homogeneous and heterogeneous environments have the different condition. Only one node becomes abnormal node (X) when the Denial of Service (DoS) attacks the node with heterogeneous environment. Figure 30 and Figure 31 shows the detail of the experiment result of DoS attack simulation. The script will add the IP address that is suspected of being an attacker to the firewall list by utilizing the `iptables` command if the healthy level value is lower than the threshold.

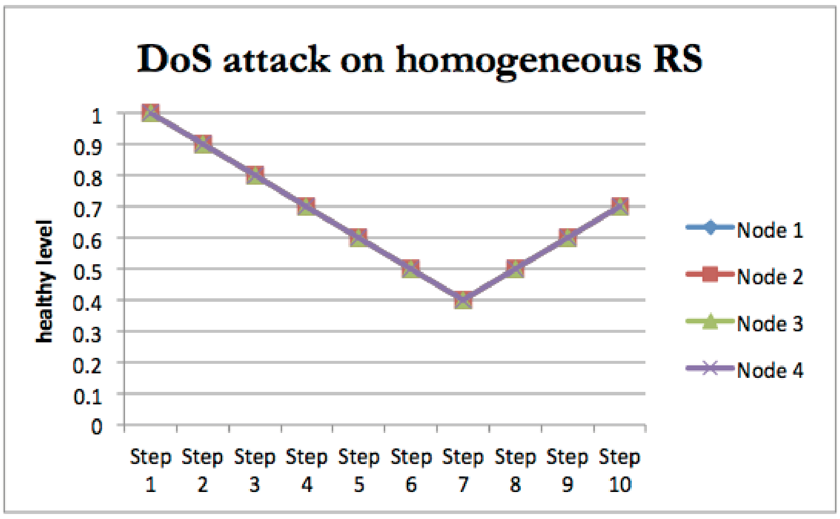


Figure 30. Result of DoS scenario on the resilient server with a homogeneous environment.

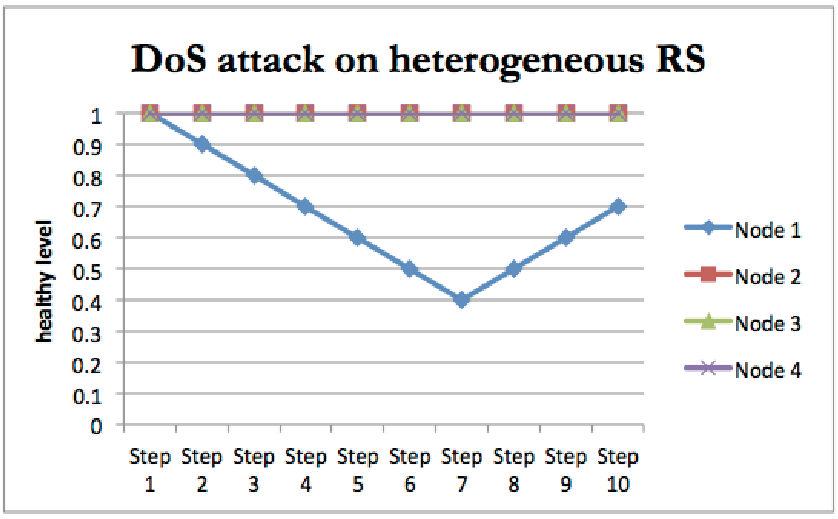


Figure 31. Result of DoS scenario on the resilient server with a heterogeneous environment.

Table 10. Test cases of DoS scenario (step 7) on the resilient server with a homogeneous environment.

Node ID #	**Service**	**Suspected IP Address**	**Healthy level**	**State**	**Action**	**SRN Model**
1	A	192.168.0.1	0.4	X	Block	*mixed-repair*
2	A	192.168.0.2	0.4	X	Block	*mixed-repair*
3	A	192.168.0.3	0.4	X	Block	*mixed-repair*
4	A	192.168.0.4	0.4	X	Block	*mixed-repair*

Table 11. Test cases of DoS scenario (step 7) on the resilient server with a heterogeneous environment.

Node ID #	**Service**	**Suspected IP Address**	**Healthy level**	**State**	**Action**	**SRN Model**
1	T	192.168.1.1	0.4	X	Block	*mixed-repair*
2	N	192.168.1.2	1.0	O	Block	*mixed-repair*
3	A	192.168.1.3	1.0	O	Block	*mixed-repair*
4	L	192.168.1.4	1.0	O	Block	*mixed-repair*

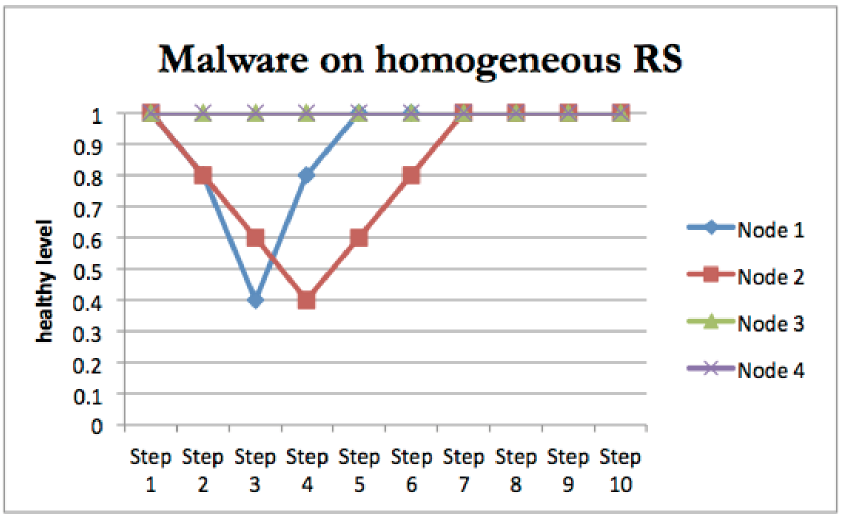


Figure 32. Result of malware scenario on the resilient server with a homogeneous environment.

4.6.3. Malware Simulation

In this simulation we create a script by involving antivirus and Tripwire. The script collects information from the antivirus or Tripwire whether the virus or anomaly inside the node is found (X) or not (O). In the resilient server with a homogeneous environment, the probabilities of the virus infection recovery on each node are the same. Meanwhile, the resilient server with heterogeneous environment has different probabilities for virus infection recovery. Table 12 and Table 13 show the test case results of the virus infection in the resilient server with homogeneous and heterogeneous environments. Figure 32 and Figure 33 shows the detail of the experiment result of malware simulation. There are two actions that can be executed by the script on each node: (1) Turn off or isolate the infected node, and (2) switch the infected node with the standby node.

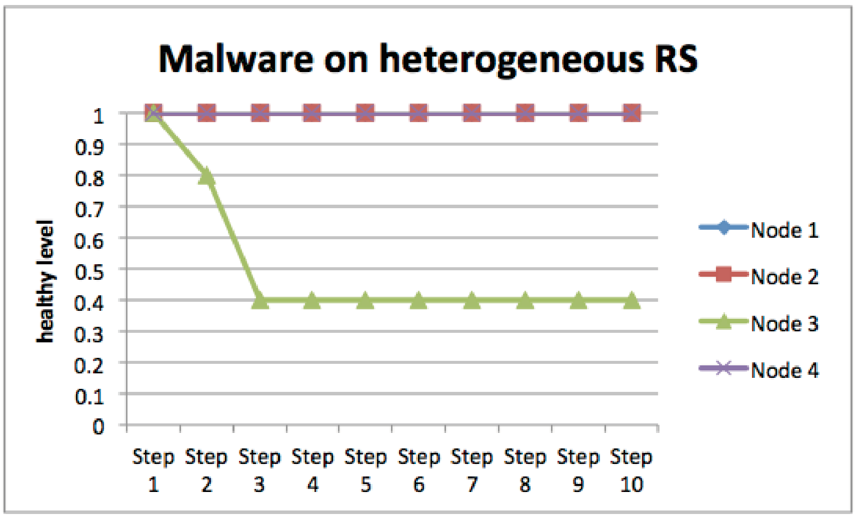


Figure 33. Result of malware scenario on the resilient server with a heterogeneous environment.

Table 12 shows that the script on node id #1 detected the virus and that the healthy level value was less than the threshold and the script responded by repairing the infected node. Although node id #2 was infected but the healthy level of node id #2 is higher than the threshold so that this node was not repaired. Table 13 shows that node id #3 has the same condition with node id #1 on Table 12. However, since the nodes are running in the resilient server with a heterogeneous environment then the script responds by switching the infected node with the standby node (state=S). While node id #4 replaces the position of node id #3, we could repair the infected node. This repair makes the application that runs vital services able to provide the service continuously and could reduce the chances of virus infection to the entire system (data center).

Table 12. Test cases of malware scenario (step 3) on the resilient server with a homogeneous environment.

Node ID #	**OS**	**IP Address**	**Infection Disk Partition ID#**	**Healthy level**	**State**	**Action**	**SRN Model**
1	M	192.168.0.1	1, 4	0.4	X	Copy	*mutual-repair*
2	M	192.168.0.2	1, 3	0.6	O	-	-
3	M	192.168.1.1	-	1	O	-	-
4	M	192.168.1.2	-	1	O	-	-

Table 13. Test cases of malware scenario (step 3) on the resilient server with a heterogeneous environment.

Node ID #	**OS**	**IP Address**	**Infection Disk Partition ID#**	**Healthy level**	**State**	**Action**	**SRN Model**
1	D	192.168.2.1	-	1	O	-	-
2	R	192.168.2.2	-	1	O	-	-
3	M	192.168.2.3	1, 4	0.4	X	Switch	*switching-repair*
4	S	192.168.2.4	-	1	S	-	-

Container (LXC) technology performs faster and more efficiently uses resources (e.g. memory consume) than Virtual Machine Monitor (VMM) technology. LXC performs 10 times faster than VMM (Section 3). However the VMM has many more types of guest operating systems than the LXC. For that reason VMM has a higher diversity than LXC to create a heterogeneous environment. By increasing the diversity of the node we can reduce the risk of fault to the server.

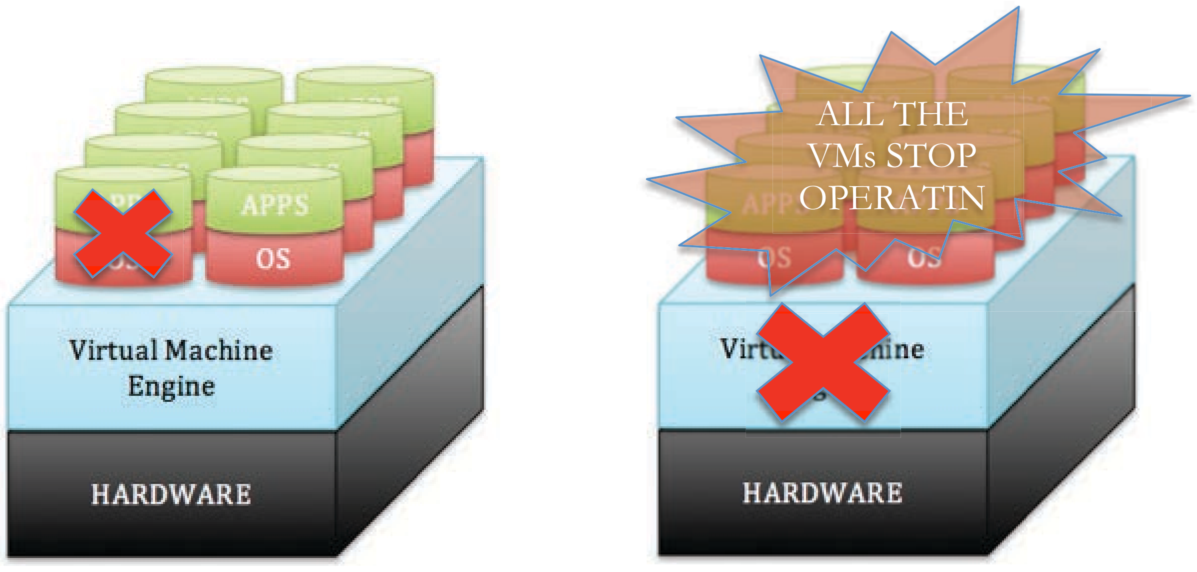
4.7. Concluding Remarks

The combination of virtualization technology and a self-repair network (SRN) model may be used to make servers resilient against failures that occur during operation. Simulations of resilient server using virtualization with a homogenous and a heterogeneous environment have been conducted. These simulations show that the server is able to recover from a failure in limited scenarios. Although Container technology offers higher performance than the Virtual Machine Monitor (VMM), VMM technology offers higher diversity. Similarly to the immune system, the diversity of nodes in the heterogeneous environment may lessen the vulnerability of the server since the virus in one infected node is unable to spread to other nodes.

5 Increasing The Diversity of Resilient Server Using Multiple Virtualization Engines and Distributed SRN Manager

We have built the resilient server with a heterogeneous resilient server that able to rescue from failures in Section 4. However, this resilient server 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 [70], CVE-2014-8866 [71]). Figure 34 shows the comparison of failures that occurs on the virtual machine and the virtualization engine. When the failure occurs on the virtual machine, it will not affect to the other virtual machine. However, if the failure occurs on the virtualization engine, then the entire virtual machine under the same virtualization engine will stop operating.

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. We equipped the virtualization engine with an application that operates with a role to monitor and respond to failures called the SRN manager. Further, the SRN manager implements the SRN model that works to monitor and respond to all of the virtualization engines. The SRN manager can be placed in the centralized or decentralized (distributed) position. If the SRN manager is placed in centralized position then its mean that we use a single 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 [72].



(a) (b)

Figure 34. Comparison of failures that occurs on (a) the virtual machine and (b) the virtualization engine.

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 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.

Algorithm 4. 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
```

5.1. 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 4 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 4). 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 (line 7 and 8 of Algorithm 4). Further, when the primary SRN manager that has the highest priority number is working, it will trigger the other scripts (Algorithm 5, Algorithm 6 and Algorithm 7) to activate its function to monitor and respond to failures occurring on the server (line 6 of Algorithm 4).

Algorithm 5. Pseudo-code implementation of the script for hang scenario with the distributed SRN manager

```
1   R ← 1
2   threshold ← 0.5
3   repeat
4       activeSrnm ← checkTheSrnmStatus()
5       if receiveRespondRequest = FALSE and activeSrnm = TRUE then
6           if R < threshold then
7               resetTheHangNode()
8               makeAReportToTheAdministrator()
9           end if
10          if R > 0 then
11              R ← R - 0.1
12          end if
13      else
14          if R < 1 then
15              R ← R + 0.1
16          end if
17      end if
18  until scriptIsStopped
```

We create several scenarios of failure of the server that are similar to those used in our previous section (Section 4.5), 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 (Algorithm 1 to Algorithm 3). The modification aims to adapt to the new design of the SRN manager as shown in line 4 and 5 of Algorithm 5, Algorithm 6, and Algorithm 7. We also need to consider implementation of Algorithm 5, Algorithm 6, and Algorithm 7, since we use multiple virtualization engines. For example, in Algorithm 5 (hang scenario), when the healthy level (*R*) of the node is lower than the threshold (line 6 of Algorithm 5) then the node will be defined as the hang node. The SRN manager will respond to this failure by resetting the hang node (line 7 of Algorithm 5). Since we use two types of virtualization engine, then each of virtualization engines has different command to reset the virtual machine. For example, 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 6. Pseudo-code implementation of the script for DoS with the distributed SRN manager

```
1  $R$ ← 1
2  $threshold$ ← 0.5
3  repeat
4      $activeSrnm$ ← checkTheSrnmStatus()
5      if $N_c \geq I_c$ and $activeSrnm$ = TRUE then
6          if $R < threshold$ then
7              addTheSuspectedIpToFirewall()
8          else
9              removeTheSuspectedIpFromFirewall()
10         end if
11         if $R > 0$ then
12             $R$ ← $R$ - 0.1
13         end if
14     else
15         if $R < 1$ then
16             $R$ ← $R$ + 0.1
17         end if
18     end if
19 until scriptIsStopped
```

Algorithm 7. Pseudo-code implementation of the script for Malware scenario with the distributed SRN manager

```
R ← 1
threshold ← 0.5
repeat
  activeSrnm ← checkTheSrnmStatus()
  if detectMalware = TRUE and activeSrnm = TRUE then
    cleanTheMalware()
    R ← R – 0.1
    if R < threshold  then
      isFound ← findNormalNodeWithSameAppsButDifferentOs()
      if isFound = TRUE then
        switchTheAbnormalNodeWithStandbyNode()
        makeAReportToTheAdministrator()
      else
        isolateTheNode()
        makeAReportToTheAdministrator()
      end if
    end if
  else
    if R < 1 then
      R ← R + 0.1
    end if
  end if
until scriptIsStopped
```

The same condition also occurs to DoS attack scenario as shown on Algorithm 6 at line 7 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 healthy level is lower than the threshold (line 6 of Algorithm 6). The healthy level of the node will decrease if the number of existing connections (N_c) is higher than the allowed maximum number of connections (I_c) as shown in Algorithm 6 at line 12. 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).

Meanwhile, Algorithm 7 is used to respond the malware that compromises the server. When the malware is detected, then the server will try to clean up the malware by itself (*self-repair*) and the *R* value of the server will be decreased (line 6 and 7 of Algorithm 7). When the next inspection the malware remain exist and the *R* value lower than the threshold value, then the server will find the replacement (*switching-repair*) as shown on line 9 and 11 of Algorithm 7.

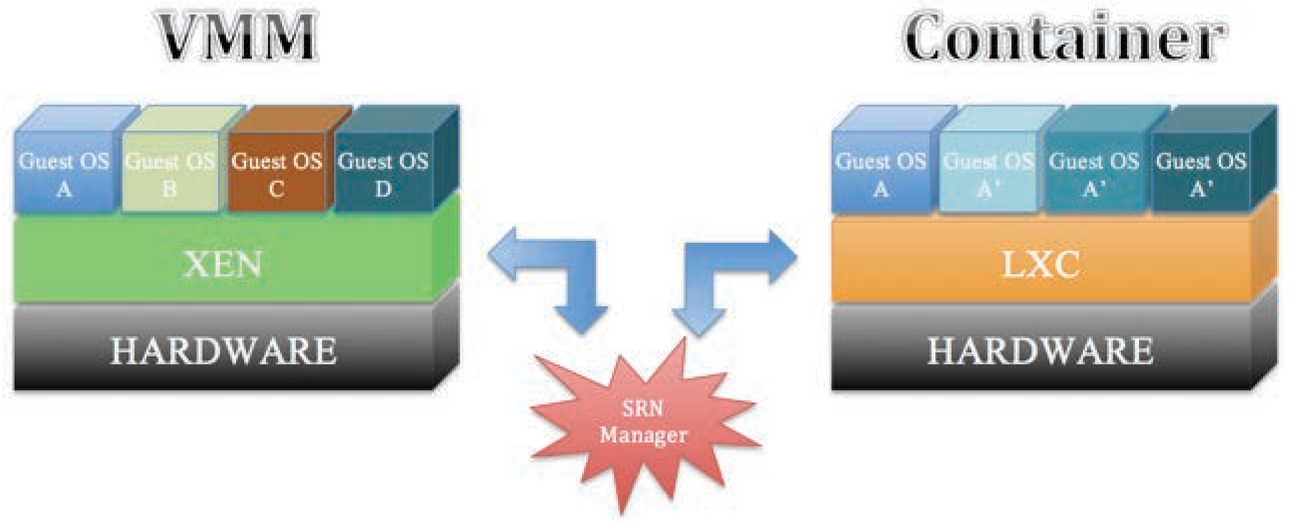


Figure 35. Resilient server with two different virtualization engines.

5.2. Multiple Virtualization Engines

The use of multiple virtualization engines is intended to increase the diversity of the server. This concept is inspired by biological diversity where the same infectious agent will not infect individuals [49]. Further, this concept was first introduced in Tempo-C specializer of the Synthetix project [33]. To actualize type #8 of the resilient server, we need to use a distinct virtualization engine on each of the physical server machines. Virtualization engine is an environment where the VM can be run. If the virtualization engine fails, then all the VM running on it will stop functioning. Therefore, we have to preserve the existence of the virtualization engine by providing an alternative VM technology to prevent failure on the virtualization engines.

There are two VM technologies that we use in this work, VMM and container (Figure 35). The combination of them will give the advantage of increasing diversity at the virtualization engines level. Since the VMM is used for server operation, accordingly XEN as native VMM are chosen. We choose XEN as native VMM since it is freeware and successful (i.e., Citrix Suites). Further, XEN provides flexibility and ease of integration [73] with other applications and programming libraries to support the SRN Manager. The other virtualization engines that we use as container is LXC. We use LXC since it has simple design, which makes the LXC have less vulnerability. Further, LXC provides a facility to configure the network for the VM [74] without involving the other application (e.g., `iptables` command).

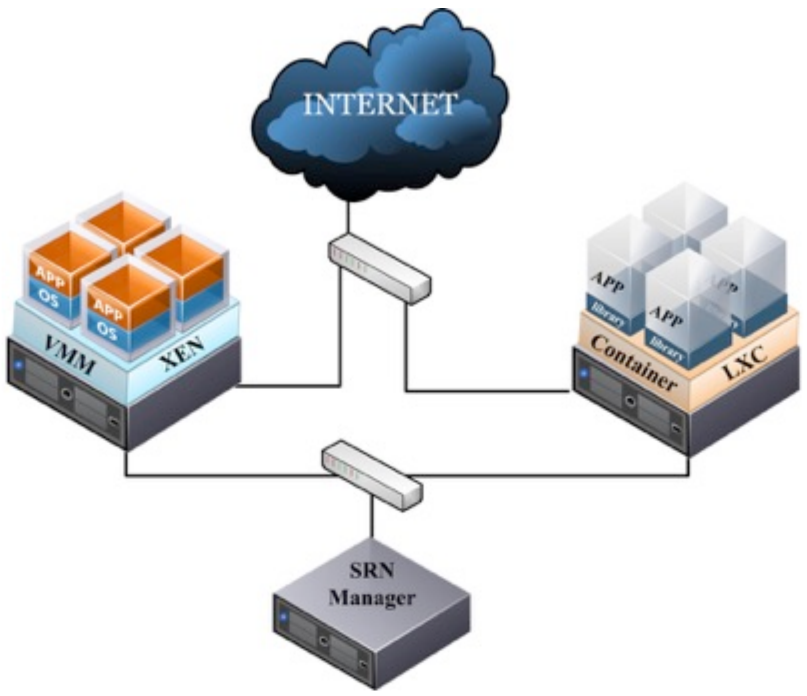


Figure 36. Centralized SRN manager on type #8 of the resilient server.

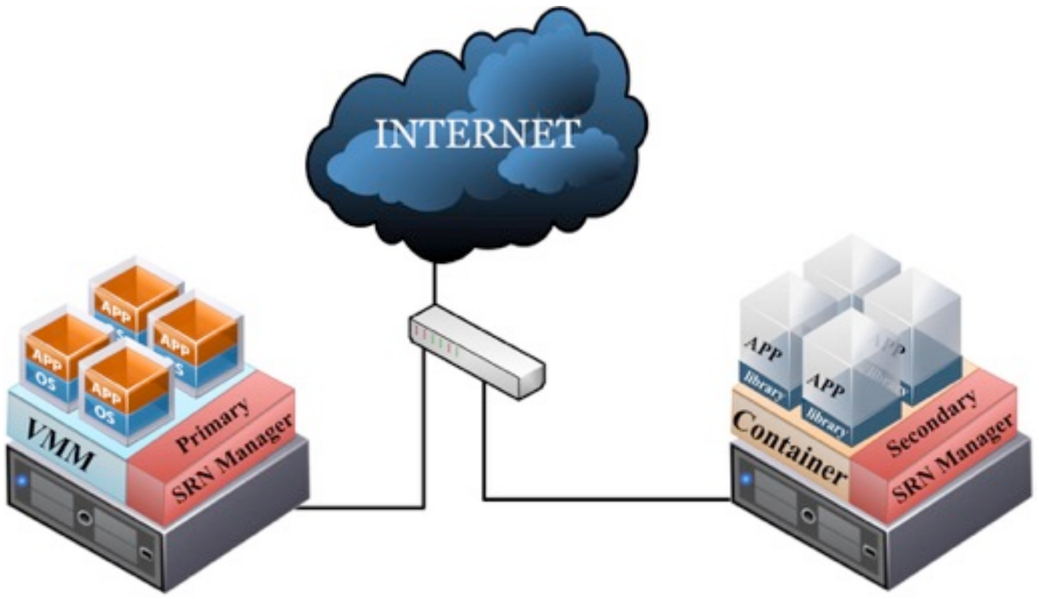


Figure 37. Distributed SRN manager on type #8 of the resilient server.

5.3. System Design and Implementation

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:

5.3.1. Centralized SRN Manager

Figure 36 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 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.

Table 14. Technical specification of the physical server machine [72].

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

5.3.2. 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 37. 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 refers to our previous work [72] and is shown in Table 14. 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. 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.

5.4. Experimental Results

We test our design (type #8 of the resilient server) shown in Figure 37 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 (Section 4.5). The server failure includes hang, DoS, and malware. The experimental results of the failure scenarios are shown as follows.

5.4.1. 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 the all the nodes under VMM (XEN) stop running as shown in Table 15 and Figure 38. When the SRN manager on VMM is 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 16 and Figure 39. 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).

Table 15. Experimental result of the hang failure (step 7) with the distributed SRN manager

Virt. Engine	Node ID #	Guest OS	Guest OS Healthy level	Guest OS State	SRN Man. State	SRN Man. Respond	SRN Model
VMM	1	Debian	0.4	X	X	*Migrate*	*switching-repair*
	2	Ubuntu	0.4	X		*Migrate*	*switching-repair*
	3	Fedora	0.4	X		*Migrate*	*switching-repair*
	4	Win. 7	0.4	X		*Migrate*	*switching-repair*
Container	5	CentOS	1	O	O	-	-
	6	Debian	1	O		-	-
	7	Fedora	1	O		-	-
	8	Ubuntu	1	O		-	-

Table 16. Experimental result of the hang failure (step 7) with the centralized SRN manager

Virt. Engine	Node ID #	Guest OS	Guest OS Healthy level	Guest OS State	SRN Man. State	SRN Man. Respond	SRN Model
VMM	1	Debian	0.4	X	X	*(unable to respond)*	-
	2	Ubuntu	0.4	X		*(unable to respond)*	-
	3	Fedora	0.4	X		*(unable to respond)*	-
	4	Win. 7	0.4	X		*(unable to respond)*	-
Container	5	CentOS	1	O		*(unable to respond)*	-
	6	Debian	1	O		*(unable to respond)*	-
	7	Fedora	1	O		*(unable to respond)*	-
	8	Ubuntu	1	O		*(unable to respond)*	-

5.4.2. 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*, *slowloris*, and *httperf*. When an attacker 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 17 and Figure 40 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 18 and Figure 41 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).

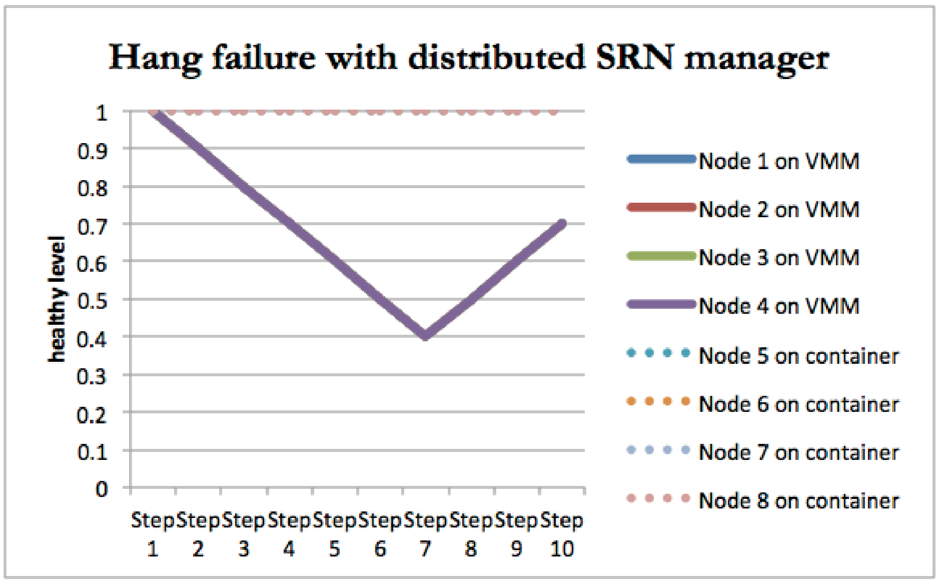


Figure 38. Result of hang failure on the resilient server with distributed SRN manager.

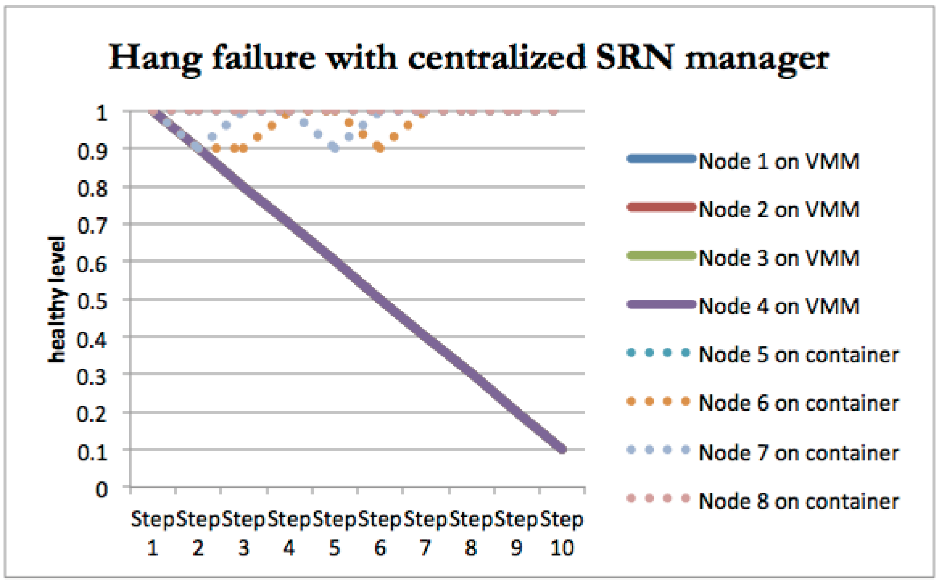


Figure 39. Result of hang failure on the resilient server with centralized SRN manager.

Table 17. Experimental result of the DoS attack failure (step 7) with the distributed SRN manager

Virt. Engine	**Node ID #**	**Guest OS**	**Guest OS Healthy level**	**Guest OS State**	**SRN Man. State**	**SRN Man. Respond**	**SRN Model**
VMM	1	Debian	0.4	X	X	*Block*	*mixed-repair (self-repair)*
	2	Ubuntu	0.9	O		*Block*	*mixed-repair (mutual-repair)*
	3	Fedora	1	O		*Block*	*mixed-repair (mutual-repair)*
	4	Win. 7	1	O		*Block*	*mixed-repair (mutual-repair)*
Container	5	CentOS	1	O	O	*Block*	*mixed-repair (mutual-repair)*
	6	Debian	1	O		*Block*	*mixed-repair (mutual-repair)*
	7	Fedora	1	O		*Block*	*mixed-repair (mutual-repair)*
	8	Ubuntu	1	O		*Block*	*mixed-repair (mutual-repair)*

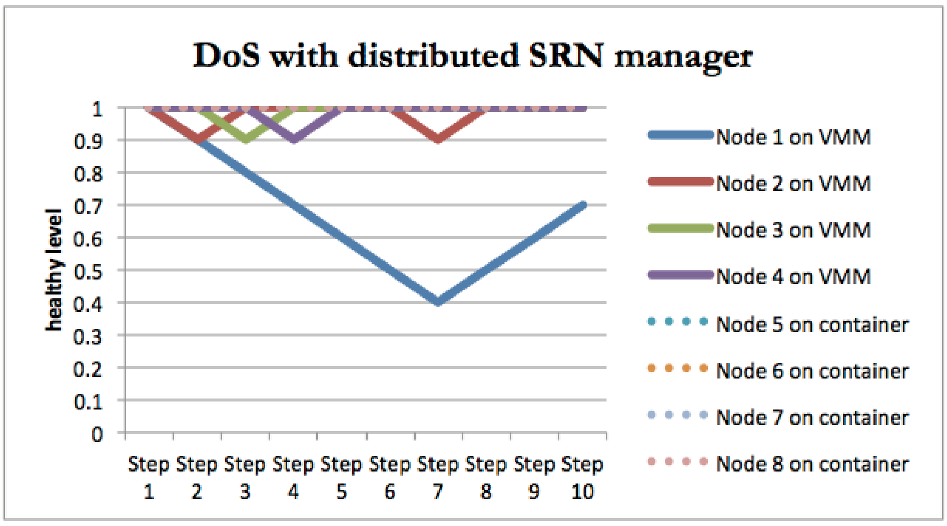


Figure 40. Result of DoS attack on the resilient server with distributed SRN manager.

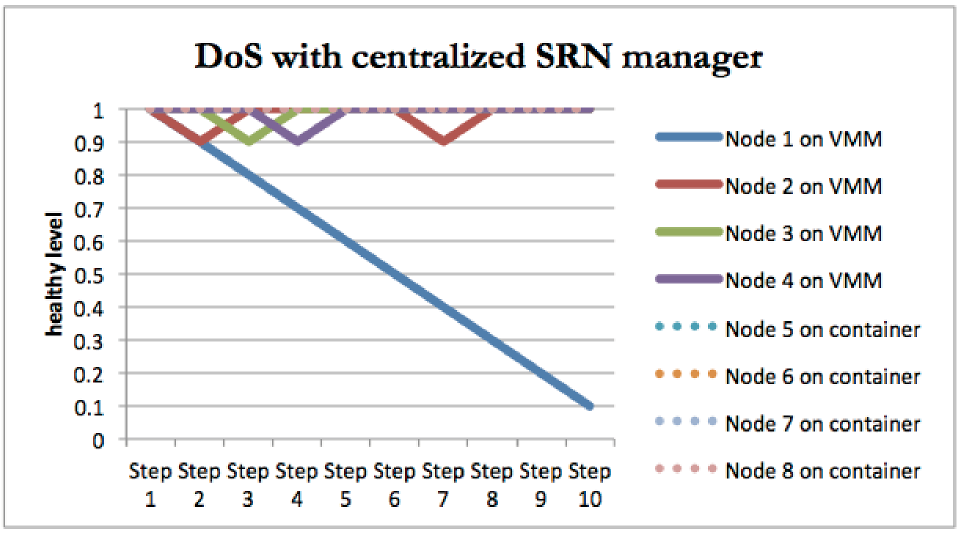


Figure 41. Result of DoS attack on the resilient server with centralized SRN manager.

Table 18. Experimental result of the DoS attack failure (step 7) with the centralized SRN manager

Virt. Engine	**Node ID #**	**Guest OS**	**Guest OS Healthy level**	**Guest OS State**	**SRN Man. State**	**SRN Man. Respond**	**SRN Model**
VMM	1	Debian	0.4	X	X	*(unable to respond)*	-
	2	Ubuntu	0.9	O		*(unable to respond)*	-
	3	Fedora	1	O		*(unable to respond)*	-
	4	Win. 7	1	O		*(unable to respond)*	-
Container	5	CentOS	1	O		*(unable to respond)*	-
	6	Debian	1	O		*(unable to respond)*	-
	7	Fedora	1	O		*(unable to respond)*	-
	8	Ubuntu	1	O		*(unable to respond)*	-

5.4.3. 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* [75]) then the SRN manager responds by cleaning the malware as shown by node id #1 in Table 19 and Figure 42. 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 19.

The worst case happens when the malware attacks the resilient server with the centralized SRN manager as shown in Table 20 and Figure 43. 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 19. Experimental result of the Malware failure (step 3) with the distributed SRN manager

Virt. Engine	**Node ID #**	**Guest OS**	**Guest OS Healthy level**	**Guest OS State**	**SRN Man. State**	**SRN Man. Respond**	**SRN Model**
VMM	1	Debian	0.4	X	X	*Clean*	*self-repair*
	2	Ubuntu	0.4	X		*Migrate*	*switching-repair*
	3	Fedora	1	O		-	-
	4	Win. 7	1	O		-	-
Container	5	CentOS	1	O	O	-	-
	6	Debian	1	O		-	-
	7	Fedora	1	O		-	-
	8	Ubuntu	1	O		-	-

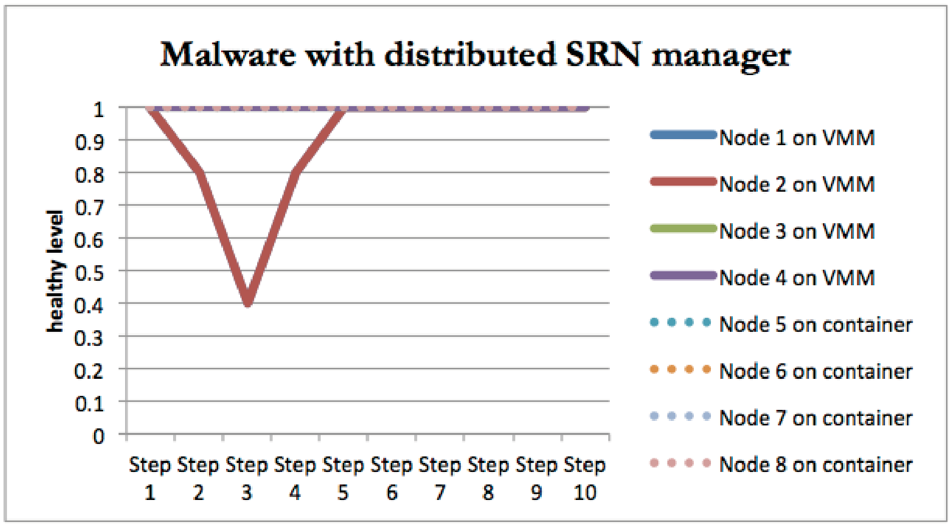


Figure 42. Result of malware scenario on the resilient server with distributed SRN manager.

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

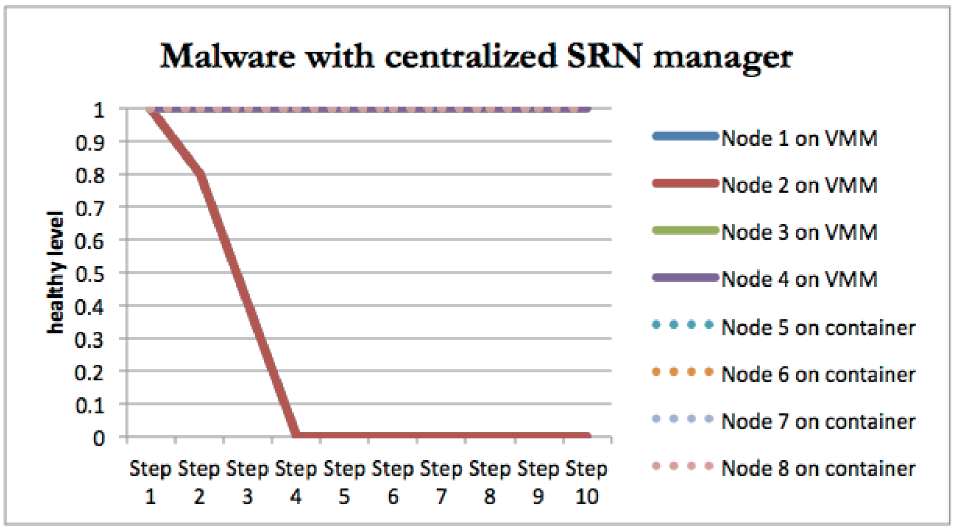


Figure 43. Result of malware scenario on the resilient server with centralized SRN manager.

Table 20. Experimental result of the Malware failure (step 4) with the centralized SRN manager

Virt. Engine	**Node ID #**	**Guest OS**	**Guest OS Healthy level**	**Guest OS State**	**SRN Man. State**	**SRN Man. Respond**	**SRN Model**
VMM	1	Debian	0.4	X	X	*(unable to respond)*	-
	2	Ubuntu	0.4	X		*(unable to respond)*	-
	3	Fedora	1	O		*(unable to respond)*	-
	4	Win. 7	1	O		*(unable to respond)*	-
Container	5	CentOS	1	O		*(unable to respond)*	-
	6	Debian	1	O		*(unable to respond)*	-
	7	Fedora	1	O		*(unable to respond)*	-
	8	Ubuntu	1	O		*(unable to respond)*	-

Based on the experimental result of resilient server type #8 and compared to the other types of resilient server, 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 [72]. Further, this design is also inspired by the diversity of operating systems [33] and an immunity-based system [68]. Table 21 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.

Table 21. Comparison of resilient server types

Resilient Server / **Failure**	**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*

5.5. Concluding Remarks

There were three parameters, including service (application), guest OS, and virtualization engine that we used in this work to design a type #8 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 [72] 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.

6 Quantitative Comparison and Performance Evaluation of Resilient Server with a Self-Repair Network Model

In this chapter, we will compare and evaluate our resilient server with other existing study using a benchmark application. The benchmark application called as UnixBench [76]. We will use this application to measure the performance losses of the server when we applied our proposed method to the server. The second benchmark application is *slowhttptest* [77]. This application is a highly configurable tool that simulates some application layer DoS attacks by prolonging HTTP connections in different ways (e.g., slow header). We will use this application to evaluate the performance of the homogeneous and heterogeneous resilient server.

6.1. Quantitative Comparison

In order to measure quantitative data, we built the resilient server with an SRN model using two types of hypervisor (KVM-based SRN and XEN-based SRN). These two hypervisors will run on different physical servers with the same specification of hardware (8 core Intel Xeon CPU and 8 GiB of RAM). In this section, we use a resilient server with totally homogeneous type (type #1). Therefore, each hypervisor host two VM with 2 vCPU and 1 GB of RAM running Debian GNU/Linux 5 and run Apache2 web server. The comparison of the performance overhead includes the individual and combined scenario of failures. The baseline is the execution time when running the workloads in each VM without the SRN manager enabled.

Overall, on average KVM-based SRN has the lowest performance losses compared to the XEN-based SRN. It is shown by the performance losses value in which the KVM-based SRN produced 0.55%, 0.6% and 0.91% performance losses for hang, DoS, and malware respectively (Table 22). While the XEN-based SRN produced the performance losses about 0.9% for hang, 2.2% for DoS and 1.29% for malware. Therefore, the combined failure for KVM-based is also lower than XEN-based SRN that is 2.37% and 3.97% respectively.

We also compare our resilient server with those in other studies that have similar scenarios of failure. Pham et al. built a reliable and secure (RnS) monitoring system for a virtual machine using out-VMI technique on the KVM hypervisor [78]. They monitor the activity of the VM using the HyperTap framework to detect hang, rootkit (malware), and privilege escalation attack. Moreover, the scenario of failures is a bit different in that the RnS detects a privilege escalation, while our resilient server detects DoS. However, we try to compare the hang, malware, and combined failure. Based on the comparison results, the KVM-based SRN still has the lowest performance losses where it shows that the performance losses of RnS in average are 1.52% for hang, 1.72% for malware and 4.3% for combined failure (Table 22). Detail of the comparison of combined failure is shown in Figure 44.

Table 22. Performance losses on the hang, DoS, malware, and combined failure.

Scenario of failure	KVM-based SRN	XEN-based SRN	RnS
Hang	0.55%	0.90%	1.52%
DoS	1.60%	2.20%	N/A
Malware	0.91%	1.29%	1.72%
Combined failure	2.73%	3.97%	4.30%

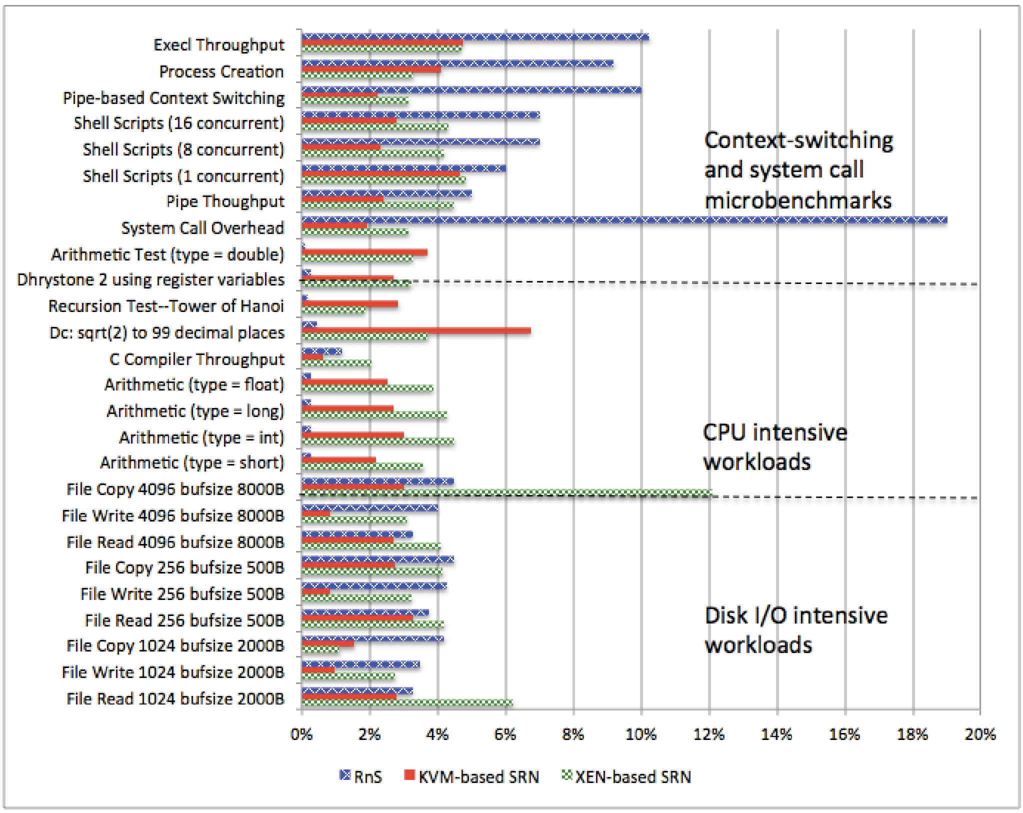


Figure 44. Comparison of performance losses for the combined failure between RnS, KVM-based SRN and XEN-based SRN.

Although the comparison results showed that the KVM-based SRN outperforms the RS, the RnS in which represents the out-VMI, offers flexibility to monitor the VM without interfering or modifying the VM. Meanwhile, in-VMI implementation is less easy because we need to modify the VM to install the detector. However, we can easily choose the detector for the in-VMI so that we can get the monitoring result in detail.

6.2. Performance Evaluation

As we have explained in Section 3.1, the key point of the resilient server is based on a biological concept being the diversity of the nodes. To evaluate the performance of the resilient server, we will focus on a specific service called the web server. We will use three types of web servers: (i) Apache 2.2.9, (ii) Nginx 0.6.32, and (iii) Lighttpd 1.4.19 which are run in the Debian GNU/Linux 5. In addition, we run those three web servers with default settings. Thus, we will create three VMs for each of the web servers (Figure 45). Since we use more than one web server then we need a load balancer so that the users or attacker can access the entire web servers as a single node. To build the load balancer for the web server, we use ipvsadm 1.24, ldirectord 2.1.3 and hearbeat 1.4.19.

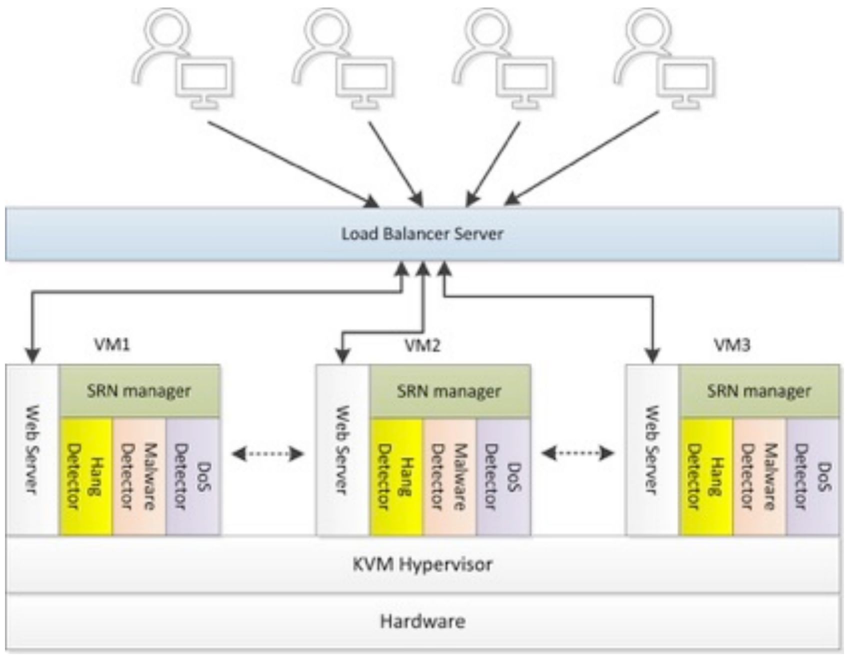
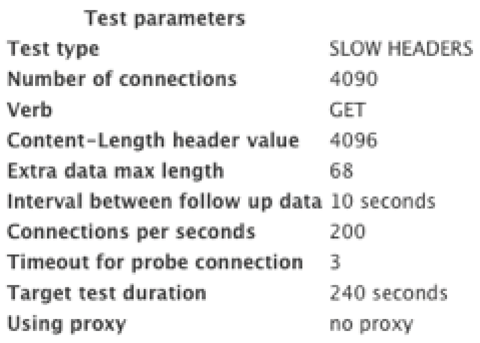


Figure 45. Three web servers with load balancer.

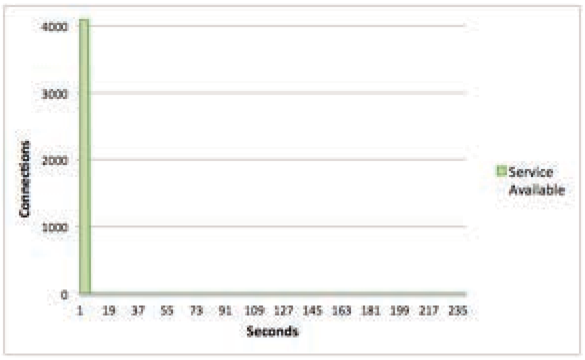
In order to evaluate the performance of the server, we use *slowhttptest* to measure the service availability. We will simulate *slowloris* (slow headers) [79] attack to this server. Therefore, we use several parameters on the *slowhttptest* as shown in Figure 46. As a result, when we implement a homogeneous web server to all servers it shows that the availability of the service is lower compared to the server which implements a heterogeneous web server. When the SRN manager is enabled, the homogeneous Apache2 web server (Figure 47a) has the lowest service availability (2,46%). While the other homogeneous web servers, Lighttpd (Figure 47b) and Nginx (Figure 47c), have better performance (70.12% and 88.35% respectively). The heterogeneous web server (Figure 47d) has the highest service availability (90.12%). Further, when the SRN manager is disabled the service availability is decreasing. The homogeneous Apache2 (Figure 48a) has 2,46% service availability while Lighttpd (Figure 48b) and Nginx (Figure 48c) has 51.6% and 60.8%

respectively. Moreover, the heterogeneous web server (Figure 48d) has 73.7% service availability. The results indicated that heterogeneous web server and SRN manager could maintain the service availability.

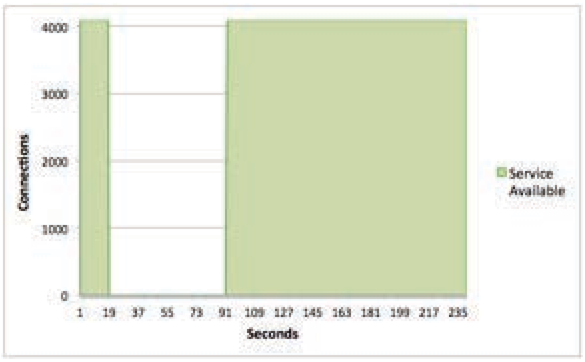


Test parameters	
Test type	SLOW HEADERS
Number of connections	4090
Verb	GET
Content-Length header value	4096
Extra data max length	68
Interval between follow up data	10 seconds
Connections per seconds	200
Timeout for probe connection	3
Target test duration	240 seconds
Using proxy	no proxy

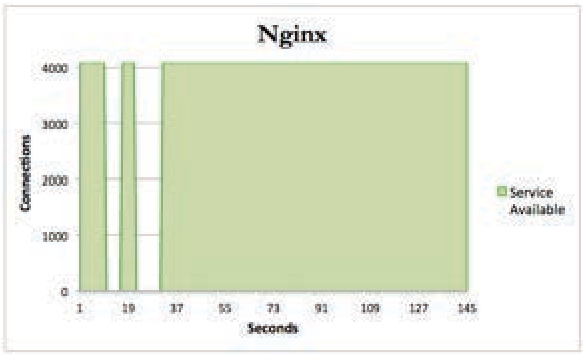
Figure 46. Test parameters of *slowhttptest* to measure the performance of the server.



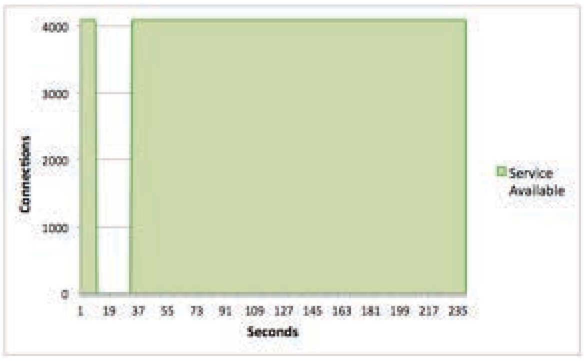
(a)



(b)

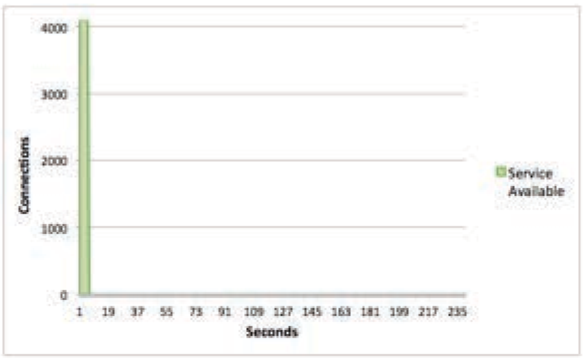


(c)

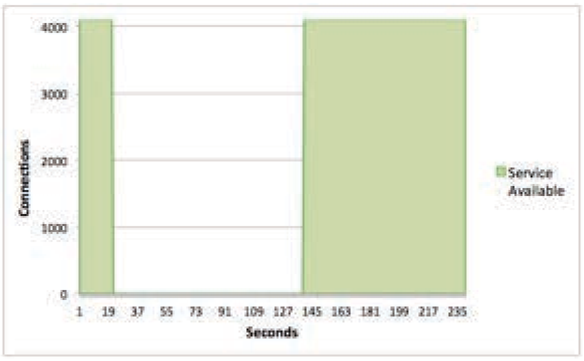


(d)

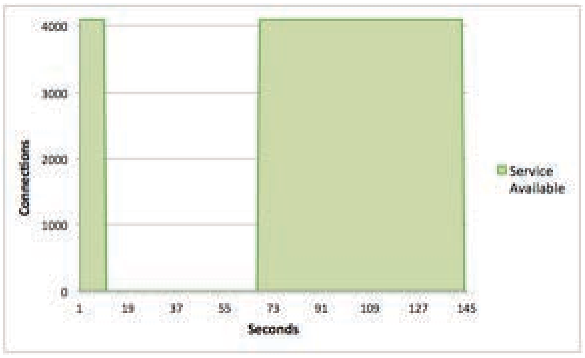
Figure 47. Performance of the homogeneous web server ((a) Apache2, (b) Lighttpd, and (c) Nginx) and (d) heterogeneous web server with SRN enabled.



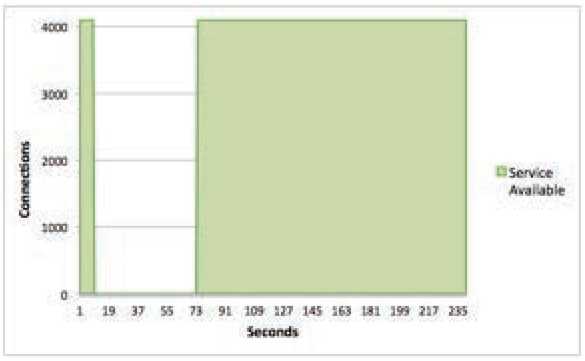
(a)



(b)



(c)



(d)

Figure 48. Performance of the homogeneous web server ((a) Apache2, (b) Lighttpd, and (c) Nginx) and (d) heterogeneous web server with SRN disabled.

6.3. Concluding Remarks

The combination of virtualization technology and a self-repair network (SRN) model may be used to build servers resilient against failures that occur during operation. This chapter presents a quantitative comparison of the resilient server using different types of virtualization engine (hypervisor) and virtual machine introspection (VMI) technique. KVM and XEN are used as the hypervisor, while in-VMI and out-VMI are used as the VMI technique. Based on the experiment results, the KVM-based SRN (in-VMI) outperform the RnS (out-VMI). However, out-VMI offers flexibility to monitor the VM so that we can create a new VM without modifying it. Further, the experimental evaluation shows that node diversity can improve the resilience of the server against unknown types of failures.

This page intentionally left blank

7 Summary of Contribution

This thesis offers the following contributions:

- *Recovery of server function.* Our work covers more generic problems such as hang, DoS, and malware compared with existing work (Section 7.1).
- *Detection of server failure*. Since we use virtualization technology, it will help the server to recover from failures automatically without human support (Section 7.2).
- *Diversification of implementation*. We have built a heterogeneous resilient server in order to decrease the vulnerability of the server (Section 7.3).

7.1. Recovery of Server Function

There are several studies that are related to the resilient server (Table 2). Since our work focuses on the virtualization-based area, then we will discuss it in more detail in this area. First, Sano et al. [53] [49] developed a resilient server that focuses on recovering the domain name system (DNS) service from a DoS attack. Like our work, their concept of the resilient server uses multiple virtualization engines and referred to as hybrid virtualization. However, their work only focuses on a single problem (DoS attack), which attacks the DNS service. In contrast, our work does not only focus on a single problem, but aims to enhance the resilience of the server by solving multiple problems (hang, DoS, and malware).

Second, Nagy et al. [23] built a server, which runs all virtual machines in parallel that run different web server implementations on various operating systems. They use N-version programming to detect a zero-day exploit. Further, Dunlap et al. [50] built a ReVirt project to help the system administrator of the server to evaluate an attack incident by replaying the attack event easily. Similar to Nagy et al. [23] study, this project is designed only to evaluate the server from failure, and the system administrator does the evaluation process manually. However, our work is not designed to detect an attack (e.g., zero-day attack), but to automatically recover from multiple failures that possibly occur during server operation.

Third, Cully et al. [25] proposed the Remus project to increase the availability of the server by asynchronous virtual machine replication. They designed a resilient server consisting of two servers as an active server and backup server. This work is similar to the fault-tolerant system where if the active server encounters a failure, then the backup server will replace its position. Different to this project, our work implements an SRN model to address the failures so that the server is able to recover from the failure not only by replacing the faulty server (node) but also through *self-repair* and *mutual-repair*.

Finally, Pham et al. [78] introduced reliable and secure (RnS) monitoring of VM using hardware architectural invariants (HyperTap framework). RnS has a slightly similar scenario of failures to our work that being hang, malware (rootkit) and escalation privilege. However, they use out-VMI as the monitoring system while we use in-VMI. Moreover, they do not discuss how to recover from a failure after it has been detected.

7.2. Detection of Server Failure

The system administrator of a computer server cannot predict when a failure will happen, and know what causes the failure. Computer failures can occur in various ways. As we have explained in Chapter 1, there are classifications of computer failure (Table 1). When computer failure occurs, we need to repair it as soon as possible and automatically so that the service (e.g., web server) that runs on the server can be accessed again. We use in-VMI to detect the failure on the VM since we use an SRN model where this model is described as a network that consists of nodes which have the capabilities of repairing other nodes and being repaired by other nodes.

In this study, we introduced the healthy level of the nodes (Section 4.4). The healthy level is used to know the condition of the nodes and to decide whether the node needs to be repaired or not. The healthy level value of the node will decrease if the abnormal condition is detected and vice versa. We use a detector to identify the failure (e.g., ping to detect hang). According to the classification of computer failure in Table 1, the failure may recover without repairing process (e.g., transient failure). That's why we also use the healthy level to avoid the SRN manager concluding all the detected failures as a permanent failure.

Although VMI research tends to focus on out-VMI due to some weaknesses of host-based monitoring [28], it turns out that in-VMI can bolster intrusion detection [29]. The experiment results show that the KVM-based SRN (in-VMI) outperforms the RnS (out-VMI). However, out-VMI offers flexibility to monitor the VM so that we can create a new VM without modifying the VM.

7.3. Diversification of Implementation

We have undertaken an experimental evaluation. Our experiment focused on the comparison of the resilient server with homogeneous and heterogeneous resilient servers. The experiment reveals that a homogeneous resilient server is easy to recover from failure, especially when the malware attacks the server. The resilient server with the homogeneous environment easily repairs (replaces) the abnormal node (Figure 32). However, we have to consider that the nodes in this resilient server have the same vulnerability. Meanwhile, the resilient server with the heterogeneous environment has more resilience than in the homogeneous environment (Figure 47) where the heterogeneous environment provides the highest availability of the service.

8 Conclusion and Future Work

8.1. Conclusion

We have successfully built the simulation of the resilient server using virtualization technology. We define eight types of resilient server using virtualization technology by combining three parameters of server component (service, guest OS, and virtualization engine). In this work, we use two types of virtualization technology, which is used in this study. The first type is VMM and the second type is container. We use XEN as VMM and Docker (LXC) as container. To simulate the resilient server with virtualization technology, we create three scenarios of failure that possibly occurs during server operation including hang, DoS, and malware. Based on the simulation results, the resilient server with virtualization technology demonstrated how a server could keep the resilience and protect against server operation malfunctions without any human support. Further, the simulation also revealed that container performance outperforms the VMM. It is shown by the execution time of container, which is ten times faster than VMM. Meanwhile, the memory usage of container is more efficient than VMM where container consumes 37.4%, while VMM consume more than 90% of memory resource. Although, the container is faster and more efficient, but VMM offers higher diversity that can be a solution to keep more resilience (learned from the immune system).

The resilient server can be built in the homogeneous (type #1) and heterogeneous (type #2 to #8) environment. We compare the performance both of resilient server by involving the SRN model. There four model of SRN that we used in the experiment. The SRN model will be applied to solve the failure that occurs on the server in the homogeneous and heterogeneous resilient server. The experiment results show that the homogeneous resilient server is easier to replace or copy to the faulty part from the normal node (server). Although the resilient server with homogenous environment offers ease of part replacement, this resilient server has less resilience than heterogeneous environment since the entire server in the homogenous environment has the same vulnerability. Furthermore, the biggest challenge on the resilient server with heterogeneous resilient server is synchronization of data or information among the servers.

Finally, in order to increase the resilience of the server, we need to enhance the diversity of the server by building the resilient server type #8 where all of the parameters have a different combination. In this type of resilient server, we have to consider the importance of the SRN manager that operates to monitor and respond the failures that occur on the server. The SRN manager can be placed in the centralized or decentralized position. The experiment results explain the performance of distributed SRN manager on the resilient server surpasses the centralized SRN manager. When the SRN manager is placed in the distributed position, it can rescue the server from failures that occur on the guest OS and host OS (virtualization engine) since more than one SRN manager is available to protect the server. However, distributed SRN manager causes higher cost compared to centralized SRN manager. Also, the quantitative comparison reveals that our proposed method has low performance loss and the diversity of the nodes can improve the resilience of the server against unknown type of failures.

8.2. Future Work

In the future, 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. Further, SDN concept offers a higher level of server resilience not only in respect to the host but also from the perspective of the entire network. When the server can organize the network using the SDN concept, then it can avoid particular failure, e.g., DoS attack [80]. Figure 49 shows a simple simulation of resilient server by involving SDN concept without SRN model. Moreover, we also have to consider physical failure due to natural events (e.g., earthquakes, floods, etc.) 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.

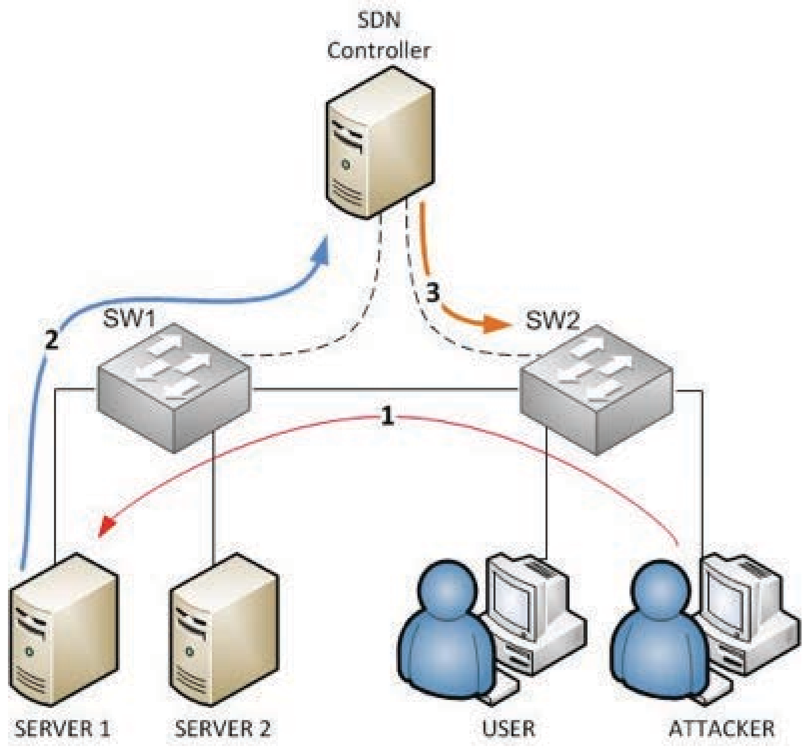


Figure 49. Simulation of resilient server with SDN concept.

References

[1] D. Hemmendinger, A. Ralston, D. Reilly, and S. Maffeis, "Client/Server Term Definition," in *Encyclopedia of Computer Science*.: 1998 International Thomson Computer Publishing, 1998.

[2] IEC 60300-3-10. (2001) International Standard IEC 60300–3–10: Dependability Management - Part 3–10: Application guide-Maintainability.

[3] Spiceworks Inc. (2016, Aug.) Server virtualization and OS trends. [Online]. https://community.spiceworks.com/networking/articles/2462-server-virtualization-and-os-trends

[4] M. Rosenblum and T. Garfinkel, "Virtual machine monitors: current technology and future trends," *Computer*, vol. 38, no. 5, pp. 39-47, May 2005.

[5] David Marshall. (2011, Nov.) InfoWorld. [Online]. http://www.infoworld.com/article/2621446/server-virtualization/server-virtualization-top-10-benefits-of-server-virtualization.html

[6] S. Mansfield-Devine, "DDoS goes mainstream: how headline-grabbing attacks could make this threat an organisation's biggest nightmare," *Network Security 2016*, vol. 11, pp. 7-13, 2016.

[7] Karl Flinders. (2017, Jan.) ComputerWeekly. [Online]. http://www.computerweekly.com/news/450411930/Report-blames-human-and-legacy-system-failure-for-Yorkshire-hospital-problems

[8] Joanne Herman. (2013, Sep.) Misco News. [Online]. http://www.misco.co.uk/blog/news/01293/36-percent-of-internet-users-suffer-financial-losses-due-to-malware-attacks

[9] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engles, "An empirical study of operating systems errors," in *ACM SIGOPS Operating Systems Review*, vol. 35, 2001, pp. 73-88.

[10] D. Moore, C. Shannon, D.J. Brawn, and Stefan, S. Heoffrey M., "Inferring Internet Denial-of-Service Activity," *ACM Transaction on Computer Systems*, vol. 24, no. 2, pp. 115-139, 2006.

[11] (2016) 2016 Internet Security Threat Report. [Online]. https://www.symantec.com/content/dam/symantec/docs/reports/istr-21-2016-en.pdf

[12] Tripwire. [Online]. http://www.tripwire.com/

[13] ClamAV. [Online]. https://www.clamav.net/

[14] Chkrootkit. [Online]. http://www.chkrootkit.org

[15] Yoshiteru Ishida, *Self-Repair Networks: A Mechanism Design*.: Springer, 2015.

[16] Niels Kaj. Jerne, "The immune system," *Scientific American*, vol. 229, no. 1, pp. 52-60, 1973.

[17] D. Patel Chandrakant and J. S. Amip, *Cost model for planning, development and operation of a data center*.: Hewlett-Packard (HP), HP Laboratories, 2005.

[18] Ian Bond. (2008, Apr.) Cisco. [Online]. http://www.cisco.com/c/dam/global/da_dk/assets/docs/presentations/DCWorkshopCopenhagen0408.pdf

[19] Rodrigo S. Couto, Miguel Elias M. Campista, and Luis Henrique M. K. Costa, "A reliability analysis of datacenter topologies.," in *2012 IEEE Global Communications Conference (GLOBECOM)*, Anaheim, CA, 2012, pp. 1890-1895.

[20] C. Guo et al., "Dcell: a scalable and fault-tolerant network structure for data centers," in *the ACM SIGCOMM 2008 conference on Data communication*, Seattle, WA, USA, pp. 75–86.

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

[22] Herman Kopetz and Paulo Verissimo, "Real Time and Dependability Concepts," in *Distributed Systems*. NY, USA: Addison-Wesley Publishing, 1993, pp. 411-446.

[23] L. Nagy, R. Ford, and W. Allen, "N-Version Programming for the Detection of Zero-day Exploits," in *IEEE Topical Conference on Cybersecurity*, 2006.

[24] D.J. Scales, M. Nelson, and G. Venkitachalam, "The design of a practical system for fault-tolerant virtual machines," *ACM SIGOPS Operating Systems Review*, vol. 44, no. 4, pp. 30-39, 2010.

[25] B. Cully et al., "Remus: High availability via asynchronous virtual machine replication," in *the 5th USENIX Symposium on Networked Systems Design and Implementation*, 2008.

[26] I. Winarno and M.N. Sani, "Automatic backup system for virtualization environment," *EMITTER International Journal of Engineering Technology*, vol. 4, no. 1, pp. 91-101, 2014.

[27] Tal Garfinkel and Mendel Rosenblum, "A virtual machine introspection based architecture for intrusion detection," in *Network and Distributed Systems Security Symposium*, 2003, pp. 191-206.

[28] Yacine Hebbal, Sylvie Laniepce, and Jean-Marc Menaud, "Virtual machine introspection: Techniques and application," in *Availability, Reliability and Security (ARES), 2015 10th International Conference*, 2015, pp. 676-685.

[29] Sijin He, Moustafa Ghanem, Li Guo, and Yike Guo, "Cloud resource monitoring for intrusion detection," in *Cloud Computing Technology and Science (CloudCom), 2013 IEEE 5th International Conference*, vol. 2, 2013, pp. 281-284.

[30] Glass M., Lukasiewycz M., Reimann F., Haubelt C., and Teich J., "Symbolic Reliability Analysis of Self-healing Networked Embedded Systems," in *the 27th international conference on computer safety, reliability, and security*, Berlin, 2008, pp. 139–152.

[31] Akoglu A., Sreeramareddy A., and Josiah J., "FPGA based distributed self healing architecture for

reusable systems," *Cluster Computing*, vol. 12, no. 3, pp. 269-284, Sep. 2009.

[32] J. N. Herder, H. Bos, B. Gras, P. Homburg, and A. S. Tanenbaum, "MINIX3: A highly reliable, self-repairing operating system," *ACM SIGOPS Operating System Reviews*, vol. 40, no. 3, pp. 80-89, 2006.

[33] C. Pu, A. Black, C. Cowan, and J.A. Walpole, "A specialization toolkit to increase the diversity in operating systems," in *Workshop Notes on Immunity-Based Systems, International Conference on Multiagen Systems*, 1996, pp. 107-117.

[34] M. W. Shapiro, "Self-healing in modern operating systems," *Queue*, vol. 2, no. 9, 2004.

[35] Adve S. et al., "The Illinois GRACE Project: global resource adaptation through cooperation," in *The workshop on self-healing, adaptive, and self-managed systems*, 2002.

[36] Yuan W., Senior N. K., Adve S.V., Jones D.L., and Kravets R.H., "GRACE-1: cross-layer adaptation for multimedia quality and battery energy," *IEEE Transactions on Mobile Computing*, vol. 5, no. 7, pp. 799-815, 2006.

[37] Kant L. and Chen W., "Service survivability in wireless networks via multi-layer self-healing," in *Wireless communications and networking conference*, vol. 4, 2005, pp. 2446-2452.

[38] Corsava S. and Getov V., "Intelligent architecture for automatic resource allocation in computer clusters," in *the 17th international symposium on parallel and distributed processing*, 2003, p. 201.1.

[39] Tesauro G. et al., "A multi-agent systems approach to autonomic computing," in *the third international joint conference on autonomous agents and multiagent systems*, 2004, pp. 464-471.

[40] Fuad M.M., Deb D., and Oudshoorn M.J., "Adding self-healing capabilities into legacy object oriented application," in *the international conference on autonomic and autonomous systems*, 2006, p. 51.

[41] Haydarlou A., Overeinder B., and Brazier F., "A self-healing approach for object-oriented applications," in *the sixteenth international workshop on database and expert systems applications*, 2005, pp. 191-195.

[42] Dabrowski C. and Mills K., "Understanding self-healing in service-discovery systems," in *the first workshop on self-healing systems*, New York, 2002, pp. 15-20.

[43] Albrecht J., Oppenheimer D., Vahdat A., and Patterson D.A., "Design and implementation trade-offs for wide-area resource discovery," *ACM Transactions on Internet Technology*, vol. 8, no. 4, pp. 1-44, 2008.

[44] Subramanian S., Thiran P., Narendra N.C., Mostefaoui G.K., and Maamar Z., "On the enhancement of BPEL engines for self-healing composite web services," in *the 2008 international symposium on applications and the internet*, Washington, DC, 2008, pp. 33-39.

[45] Baresi L. and Guinea S., "Dynamo and self-healing BPEL compositions," in *29th International conference on software engineering-companion*, 2007, pp. 69-70.

[46] F. Moo-Mena, J. Garcilazo-Ortiz, L. Basto-Diaz, F. Curi-Quintal, and F. Alonzo-Canul, "Defining a self-healing QoS-based infrastructure for web services applications," in *Computer Science and Engineering Workshop*, 2008, pp. 215-220.

[47] R. B. Halima, K. Drira, and M. Jmaiel, "A QoS-oriented reconfigureable middleware for self-healing web services," in *Web Services 2008. ICWS 08 IEEE International Conference*, 2008, pp. 104-111.

[48] Moser O., Rosenberg F., and Dustdar S., "Non-intrusive monitoring and service adaptation for Ws-BPEL," in *the 17th international conference on World Wide Web*, New York, 2008, pp. 815-824.

[49] F. Sano, T. Okamoto, I. Winarno, Y. Hata, and Y. Ishida, "A cyber attack-resilient server using hybrid virtualization," *Procedia Computer Science*, vol. 96, pp. 1627-1636, 2016.

[50] G.W. Dunlap, S.T. King, S. Cinar, M.A Basrai, and P.M Chen, "ReVirt: Enabling intrusion analysis through virtual-machine logging and replay," in *The 2002 Symposium on Operating Systems Design and Implementation (OSDI)*, 2002.

[51] K. Buchacker and V. Sieh, "Framework for testing the fault-tolerance of systems including OS and network aspects," in *the 2001 IEEE International Symposium on High Assurance Systems Engineering (HASE)*, 2001, pp. 95-105.

[52] J. Chelladhurai, P.R. Chelliah, and S.A. Kumar, "Securing docker containers from denial of service (DoS) attack," in *2016 IEEE International Conference on Service Computing (SCC)*, San Fransisco, CA, 2016, pp. 856-859.

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

[54] M. Tarao and T. Okamoto, "Toward an artificial immune server against cyber attack," in *The 21st International Symposium on Artificial Life and Robotics*, 2016, pp. 36-39.

[55] T. Okamoto, "SecondDEP: Resilient computing that prevents shellcode execution in cyber-attacks," *Procedia Computer Science*, vol. 60, pp. 691-699, 2015.

[56] S. S. Moghadam, "A survey of virtualization security," *International Journal of Scientific & Engineering Research*, vol. 4, no. 9, pp. 1533-1536, 2013.

[57] K. Hashizume, D.G. Rosado, E. F. Medina, and E.B. Fernandez, "An analysis of security issues for cloud computing," *Journal of Internet Service and Applications*, vol. 4, no. 1, p. 5, 2013.

[58] J.S. Reuben, "A survey on virtual machine security," Helsinki University of Technology, Technical report 2007.

[59] D. Merkel, "Docker: Lighweight linux containers for consistent development and deployment," *Linux Journal*, vol. 14, no. 239, 2014.

[60] D. Rajdeep, R. A. Reddy, and K. Dharmesh, "Virtualization vs Containerization to support PaaS," in

IEEE International Conference on Cloud Engineering, 2014, pp. 610-614.

[61] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, "An updated performance comparison of virtual machines and linux containers," IBM Research Report, 2014.

[62] A. M. Joy, "Performance comparison between linux containers and virtual machines," in *International Conference on Advances in Computer Engineering and Application*, 2015, pp. 342-346.

[63] D. Cotreneo, R. Nattella, and S. Russo, "Assessment and improvement of hang detection in the Linux operating system," in *The 28th IEEE International Symposium on Reliable Distributed System*, 2009, pp. 288-294.

[64] Slowloris HTTP DoS. [Online]. http://ha.ckers.org/slowloris/

[65] Y. Zhu et al., "What is system hang and how to handle it," in *The 2012 IEEE 23rd International Symposium on Software Reliability Engineering*, 2012, pp. 141–150.

[66] 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.

[67] B. Q. M. Al-Musawi, "Mitigating DoS/DDoS Attacks Using Iptables," *International Journal of Engineering & Technology, IJET-IJENS*, vol. 12, no. 3, 2012.

[68] Yoshiteru Ishida, *Immunity-Based System: A Design Perspective*.: Springer.

[69] CVE-2014-0098. [Online]. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-0098

[70] CVE-2016-1571. [Online]. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-1571

[71] CVE-2014-8866. [Online]. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-8866

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

[73] Charlie Schluting. (2009, Aug.) VMWare or XEN? Depends on Your Fluency in Linux. [Online]. http://www.enterprisenetworkingplanet.com/linux_unix/article.php/3836936/VMWare-or-Xen--Depends-on-Your-Fluency-in-Linux.htm

[74] Tobby Banerjee. (2014, Aug.) Understanding the key differences between LXC and Docker. [Online]. https://www.flockport.com/lxc-vs-docker/

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

[76] UnixBench. [Online]. https://github.com/kdlucas/byte-unixbench

[77] Slowhttptest, Application Layer DoS attack simulator. [Online]. https://github.com/shekyan/slowhttptest

[78] Cuong Pham, Zachary J. Estrada, Phuong Cao, Zbigniew Kalbarczyk, and Ravishankar K. Iyer, "Building realible and secure virtual machines using architectural invariants," *IEEE Security & Privacy*, vol. 12, no. 5, pp. 82-85, 2014.

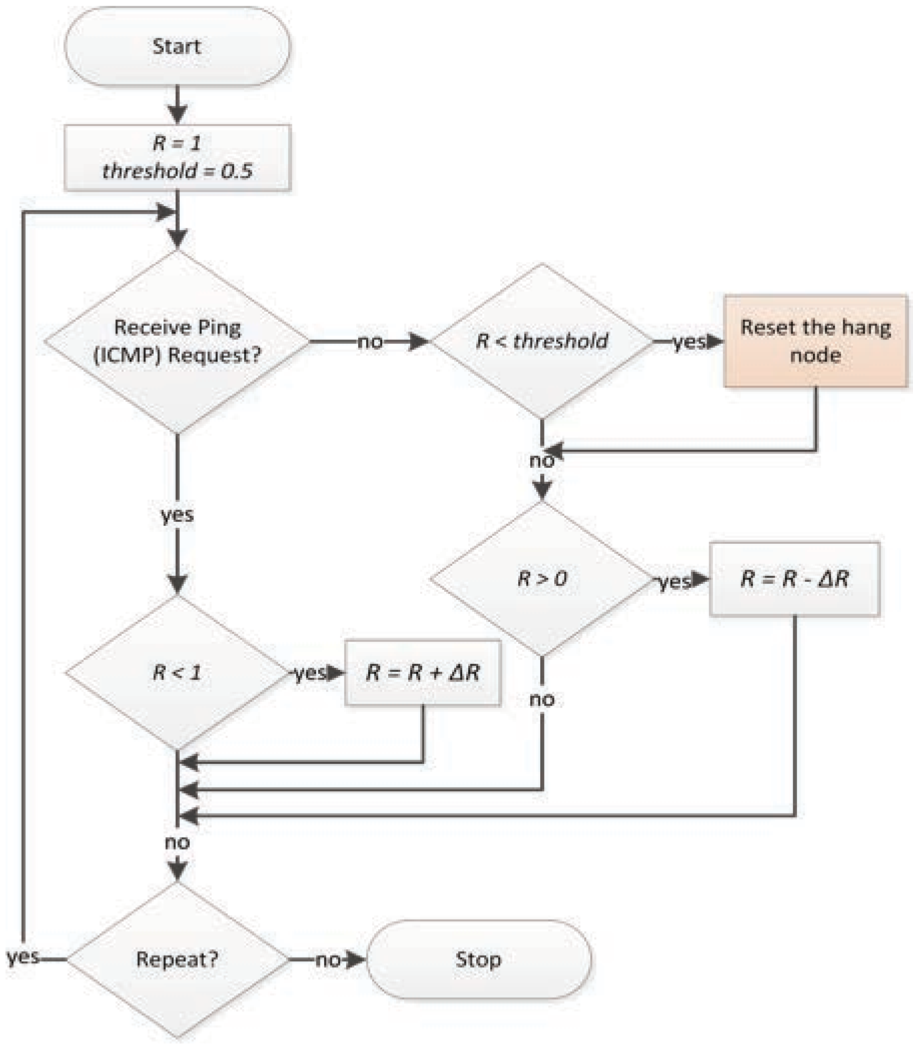
[79] CVE-2007-6750. [Online]. tp://www.cvedetails.com/cve/CVE-2007-6750/

[80] I. Winarno and Y. Ishida, "Simulating resilient server using software-defined networking," in *The 2016 International Conference on Advanced Informatics: Concepts, Theory and Application*, Penang, 2016.

[81] Fumikazu Sano, Takeshi Okamoto, Idris Winarno, Yoshikazu Hata, and Yoshiteru Ishida, "A cyber attack-resilient server inspired by biological diversity," *Artificial Life and Robotics*, vol. 21, no. 3, pp. 345-350, 2016.

Appendix 1

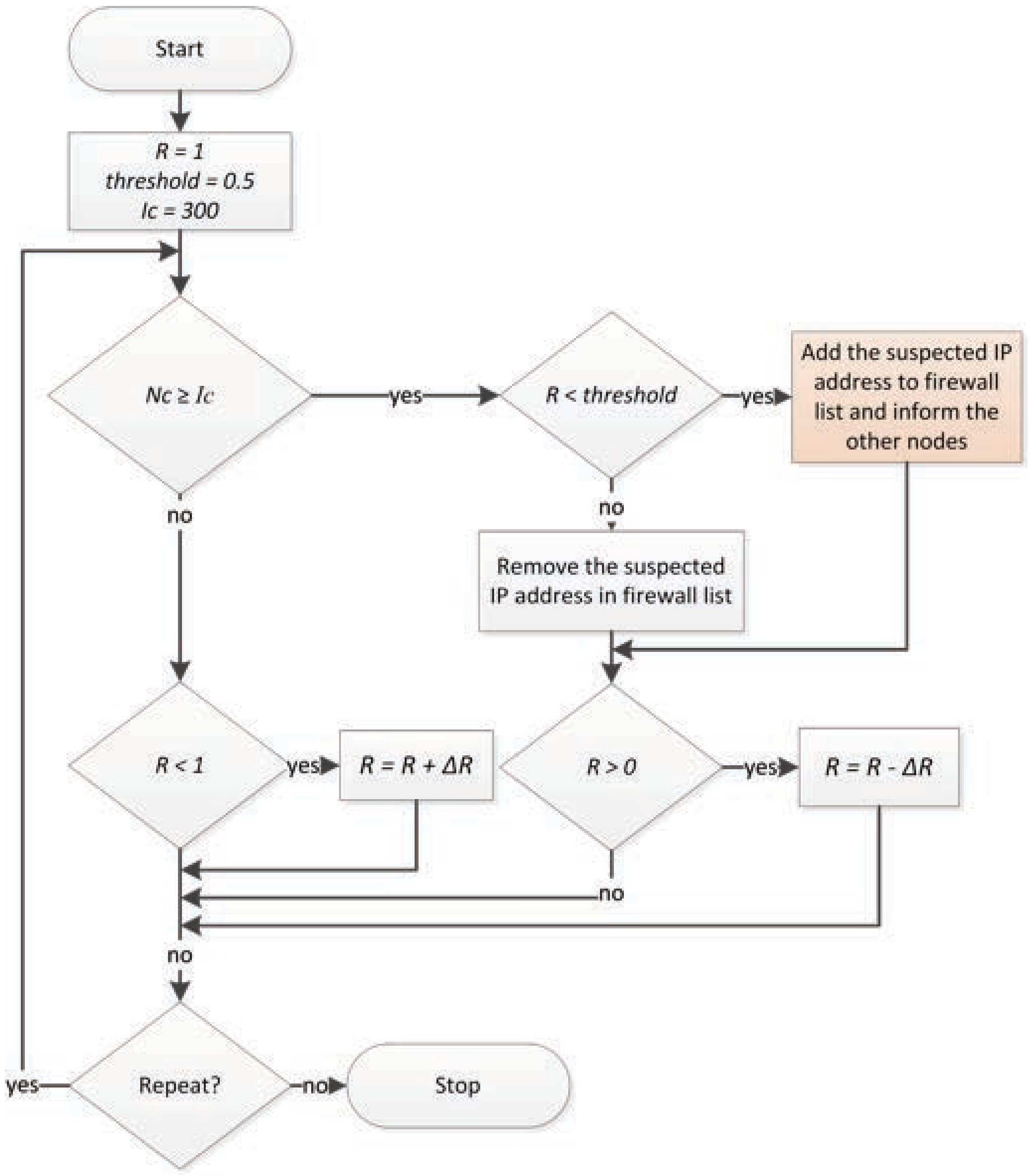
Flowchart of the script for hang scenario (Algorithm 1)



This page intentionally left blank

Appendix 2

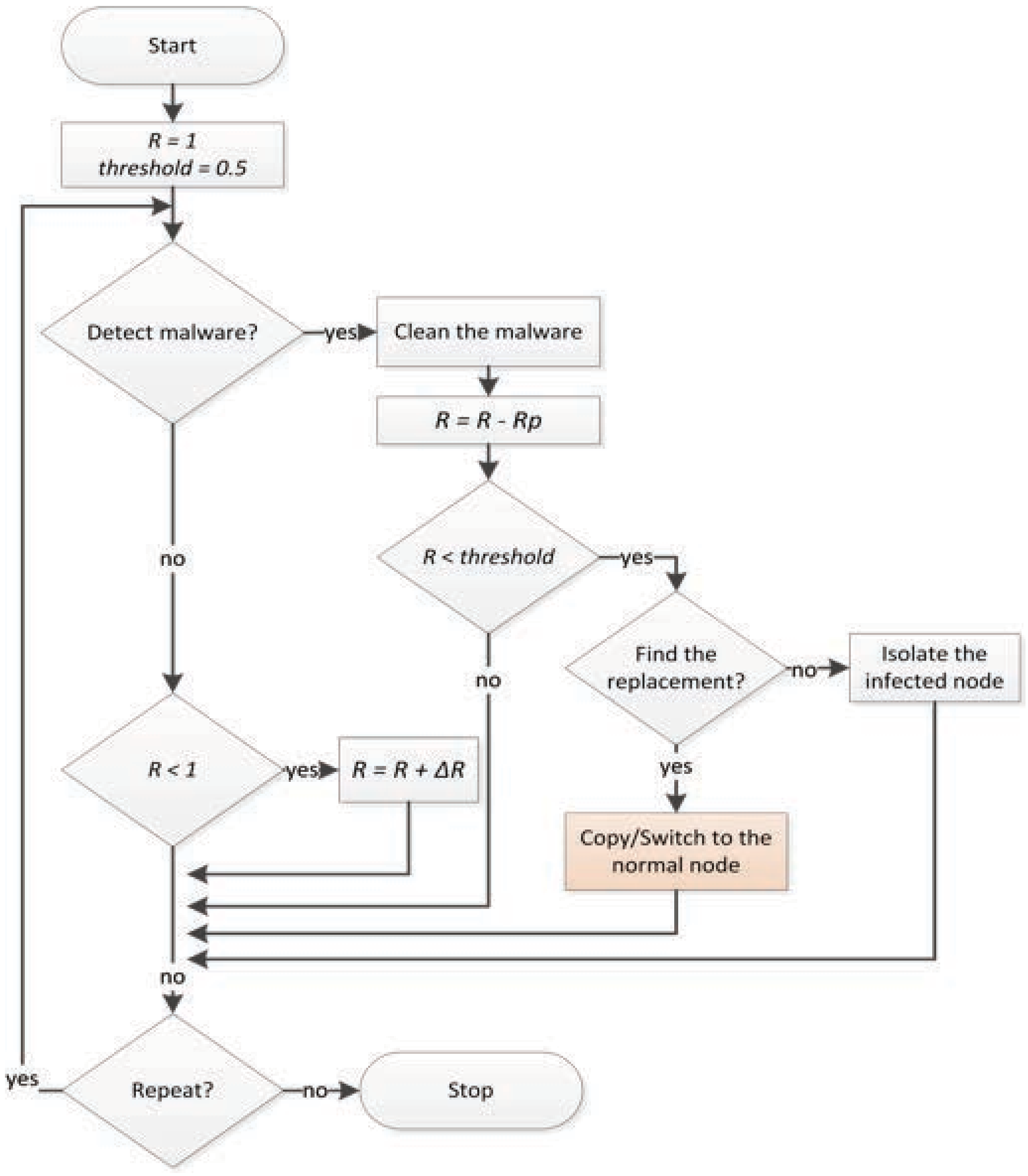
Flowchart of the script for DoS scenario (Algorithm 2)


Start
R = 1
threshold = 0.5
Ic = 300
Nc ≥ Ic
yes
R < threshold
yes
Add the suspected IP address to firewall list and inform the other nodes
no
Remove the suspected IP address in firewall list
no
R < 1
yes
R = R + ΔR
R > 0
yes
R = R - ΔR
no
no
yes
Repeat?
no
Stop

This page intentionally left blank

Appendix 3

Flowchart of the script for malware scenario (Algorithm 3)



This page intentionally left blank

List of Publications

Journal papers:

1. 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.
2. I. Winarno, T. Okamoto, Y. Hata and Y. Ishida. Distributed SRN Manager on A Resilient Server with Multiple Virtualization Engines. *International Journal of Innovative Computing, Information and Control*. vol. 13, no. 2, pp.365-379, 2017.

Conference papers:

1. I. Winarno and Y. Ishida, Simulating Resilient Server Using XEN Virtualization, *Procedia Computer Science*, vol.60, pp.1745-1752, 2015.
2. 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.
3. 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.
4. I. Winarno and Y. Ishida, Simulating Resilient Server using Software-defined Networking, *Proc. of IEEE 2016 International Conference on Advanced Informatics: Concepts, Theory and Application*, 2016.

This page intentionally left blank

Acknowledgments

All praises are to ALLAH, the most beneficent and merciful, the Lord of universe. Without His blessings, this works could not be accomplished.

I would like to express my deep appreciation to Prof. Yoshiteru Ishida for his sincere advice and passionate guidance. I have learned not only scientific knowledge, but also the method of scientific thinking, how to identify the problems and formulate the method to find the solutions.

I would like to express my gratitude to the examiner members, Prof. Shigeru Masuyama, Prof. Ren Ohmura and Prof. Toshihiro Fujito, for their valuable advice and feedback. I also would like to thank to Prof. Takeshi Okamoto and Prof. Koji Harada for giving me valuable discussions and suggestions in completion of this work. In addition, I also would like to thank to all members of System Science Laboratory for their assistance and friendship during my study especially to Hata-san.

Furthermore, I would like to thank to DIKTI (Directorate General of Higher Education, Indonesia) for providing me scholarship (contract number 242.7/E4.4/2014) during my study and Electronic Engineering Polytechnic Institute of Surabaya (EEPIS), which gives me the chance and support to study at Toyohashi University of Technology.

With my endless love, I am grateful to my lovely wife, Usmaida Agusetia. Her love, patience, encouragement, and support are undeniably. Her understanding and tolerance during accompanying me is a testament of her unyielding devotion and love. To my little boy, Rashya-kun and both of my little girls, Marsha-chan and Seisha-chan, you give me spirit and inspiration.

I extend my sincere gratitude to my parents for their endless support, supplication, sacrifice, and unconditional love. And thank to my sisters, and all my family for your encouragement, support, and supplication. And also I would like to thank to “Alioke” who has helped me a lot.

Last but not least, I would like to thank to all of EEPIS-TUT members (Zen-san, Reesa-san, Hary-san, Bima-san, Hero-san, Agus-san, and Bambang-san) and PPI (Indonesian Student Association) members for brotherhood and togetherness. I also would like to thank to Pak Udin and family for your help and kindness during my family and I stay in Toyohashi.

Toyohashi,
Idris Winarno

This page intentionally left blank