논문 21-46-09-19 The Journal of Korean Institute of Communications and Information Sciences '21-09 Vol.46 No.09
https://doi.org/10.7840/kics.2021.46.9.1497

# 컨테이너 기반 네트워크 서비스를 위한 자동화된 테스트 프레임워크

레 반끄엉•, 유 명 식°

# An Automated Testing Framework for Container-Based Network Services

Van-Cuong Le•, Myungsik Yoo°

## 요 약

NFV(Network Functions Virtualization)는 운영자에게 네트워크 서비스를 위한 매우 유연하고 확장 가능한 인프라를 제공하는 네트워크 아키텍처이다. NFV 네트워크의 네트워크 기능은 공통 서버에서 구현 및 실행된다. IT 기업들이 서비스 제품을 개발하기 위해 컨테이너형 기술을 사용하는 경향이 증가하고 있다. 따라서 컨테이너 기반 네트워크 서비스 테스트를 자동화하여 검증 속도를 높이고 네트워크 서비스 포설 비용을 절감할 수 있는 새로운 방법이 필요하다. 또한 제품이 포설된 환경에서 컨테이너 기반 네트워크 서비스의 기능 테스트 및 성능 평가를 위한 자동화된 테스트가 필요하다. 본 논문에서는 컨테이너 기반 네트워크 서비스에 대한 자동화 테스트 프레임워크를 제안한다. 제안된 프레임워크를 구현하고 테스트함으로써, 제안 프레임워크가 적합성, 상호운용성 및 성능 테스트를 포함하는 포괄적인 솔루션을 제공하는 것을 보였다.

**Key Words :** NFV, Container, Testing VNF, Kubernetes, Network Service Mesh

## ABSTRACT

Network functions virtualization (NFV) is a network architecture offering operators a highly flexible and scalable infrastructure for network services. The network functions of an NFV network are implemented and executed on common servers. There is an increasing tendency for information technology companies to use containerized technology to develop their service products. Therefore, they need a new method to automate testing container-based network services to accelerate the verification speed and reduce network service development cost. There is also a need for automatic function testing and performance evaluation of container-based network services in production environments. In this paper, we propose an automation testing framework for container-based network services. By implementing and testing the proposed framework, we show that the proposed framework provides a comprehensive solution covering conformance, interoperability, and performance testing.

※ This research was supported by the MSIT(Ministry of Science and ICT), Korea, under the ITRC(Information Technology Research Center) support program(IITP-2021-2017-0-01633) supervised by the IITP(Institute for Information & Communications Technology Planning & Evaluation).

• First Author : Department of Information Communication Convergence, Soongsil University, Seoul, South Korea, cuonglv@ssu.ac.kr, 학생회원

° Corresponding Author : School of Electronic Engineering, Soongsil University, Seoul, South Korea, myoo@ssu.ac.kr, 종신회원

논문번호 : 202106-129-D-RN, Received June 10, 2021; Revised June 17, 2021; Accepted June 17, 2021

## Ⅰ. Introduction

The European Telecommunications Standards Institute (ETSI) is helping to define the Network Function Virtualization (NFV) and related concepts to provide new approaches to adapt to increasing requirements to reduce operational and capital expenditures. Virtual network functions (VNF) in an NFV network are software that implements and facilitates network function in virtual machines or containers on top of shared servers instead of dedicated hardware[1].

For the NFV environment, VNF testing has always been considered as one of the key subjects. A single VNF has to be performed conformance testing and performance testing. Besides, a VNF communicates with other VNFs to serve its service in practice, so that the multiple VNFs also need to be performed interoperability testing for checking communication between VNFs. Previous works on VNF testing have been published[2-4]. In [2], the study provides the conformance test, but does not support performance test and interoperability test. In [3] [4], the studies focus on only performance test, but do not support conformance test and interoperability test. In this paper, we aim to provide an automatic testing solution for container-based network services framework. It consists of conformance testing, interoperation testing, and performance testing for container-based network services.

The rest of the study is organized as follows. Section II describes background different types of testing. Enabler technologies and related works are presented in Section III. Section IV depicts detail of the proposed system architecture and the workflows. Section V shows the experimental results to demonstrate the proposed solution. Finally, section VI concludes the paper and discusses future works.

## Ⅱ. Background

In this section, we describe three typical types of VNF testing defined by ETSI[5].

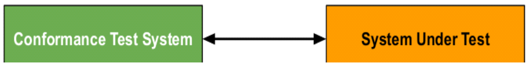


Fig. 1. Conformance Testing

Conformance testing shown in Figure 1 is to check whether a feature has been implemented correctly based on a particular standardized protocol[5]. It also shows whether or not the implementation in question meets the requirements specified for the protocol itself.

Figure 2 describes the interoperability testing presents whether the end-to-end feature of at least two functions is working as expected. Hence, it can demonstrate whether a product will work with other like products[5]. Additionally, it also assesses the ability of the implementation to support the required trans-network functionality between itself and another, similar implementation to which it is connected.

Performance Testing in the proposed system considers the number of requests per second and service latency as metrics. The purpose is to show how the service latency increases under high load and then identify the tipping point for the respective resource configuration. Figure 3 shows an example result of performance testing.

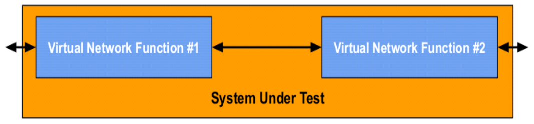


Fig. 2. Interoperability Testing

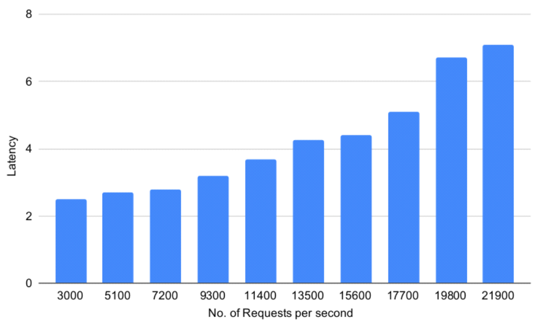


Fig. 3. A result example of performance testing

## Ⅲ. Related Works

OPNFV Yardstick aims to verify infrastructure compliance when running VNFs[3]. It helps to break down the VNF performance metrics into various attribute vectors. But it does not support functional testing of VNFs. Another tool is OPNFV Bottlenecks, as called name, its purpose is to find system bottlenecks by verifying OPNFV infrastructure in a VNF testing[4]. Its strategy is embraced with Bottlenecks to benchmark a deployment. But it is using limitation only on OPNFV infrastructure. One other effort is V&V 5GTango; it is a testing framework that targets the 5G network with many use cases[6]. However, it also misses the orchestration and does not cover the conformance, interoperability, and performance aspect of VNFs in a Network Server. Furthermore, all of the above efforts are missing considering VNFs implemented by containerized technology. It is important for the road aims to implement cloud-native network function.

A set of enabler technologies supports the proposed system becomes feasible.

### 3.1 Docker

Docker engine is an open-source containerization platform based on Linux containers for building and containerizing our application[7]. It extends existing container technology by providing a single lightweight container, API for managing Docker images and execution of container, and even a tool that allows you to create and spin multi-container applications. Docker can help us to deploy an application regardless of environment settings from one environment to another one. Besides, it can create any of our network functions in service function chains as a container. It not only saves resources but also provides easy deployment.

Docker container relies on the kernel of the host running the docker engine. It only consumes resources when they run an application. It is nothing but a user space of the OS. In the most basic level, a container is simply a set of processes that are separated from the rest of the system and run from a separate image that contains all of the files used to maintain the processes. The containers running in Docker share the host OS kernel. It uses Linux Kernel features such as storage, networking, and control groups to build containers on top of an operating system. Docker encloses an application's software into a package with everything it needs to run like OS, application code, Run-time, system tools, libraries, etc. Hence, the application can run on any other machine based on the Linux kernel without any customization.

### 3.2 Container-based Network Function

Nowadays, there is a tendency in the internet industry to roll their services out by using container technology and microservices architecture. Monolithic virtual network functions (VNFs) operating in virtual machines are among the current NFV's drawbacks (VMs)[8]. Users want consistency with their work in data center clouds, and that means cloud-native implementations of VNFs. Breaking up the monoliths into microservices architecture and implementing them as containers are a trend. It will make the transition from hardware appliances to virtualized solutions as smooth as possible. Implementing container-based network functions (CNF) is a first step that eventually would aim to become a cloud-native network function (CNF). A container-based network function (CNF) is a container-based application that implements or provides network functionality, which runs inside a Linux container. A container-based network service consists of one or more microservices container-based network functions. It includes an immutable infrastructure and declarative APIs. The network service behavior results from the combination of the individual network function behavior and the behaviors of the network infrastructure that the network service runs on.

### 3.3 Kubernetes

Kubernetes streamlines the process of implementing multi-container applications. Using Kubernetes, operators can fine-tune each container of services exactly how much resource is allocated for each container and combine different containers within a single Pod to their work together. Kubernetes then handles the process of rolling them out, maintaining them, and ensuring that all the components remain in sync[9].

At least one master and multiple worker nodes make up a Kubernetes cluster. In practice, it will have more masters to guarantee high availability and load balancing. A set of master nodes called a control plane is the management layer of Kubernetes. Basically, the control plane contains a Kubernetes API server serving the Kubernetes API. Every communication between Kubernetes components is through an API server. The API server communicates with Etcd, a distributed key-value database, via gRPC protocol. The API server stores metadata of the API objects in Etcd. All resources in a Kubernetes cluster are presented as API objects. The other parts of the control plane are controllers, which the Kube-controller-manager executes. The Kube-controller-manager consists of the different controllers. Each kind of workload resource has a respective controller providing the logic of this workload resource. The controller monitors resource objects through the APIs. In Kubernetes, controllers play a role as control loops that watch the cluster's state, then make or request changes where needed. Specifically, each controller tries to sync the current state closer to the desired state. Finally, the control plane contains a scheduler. It is responsible for scheduling the applications pod to the appreciated compute nodes. In the same fashion with controllers, the scheduler monitors the cluster's state via API for unscheduled Pods, runs scheduling algorithms, and updates the Pods with information about the node it was scheduled to.

### 3.4 Sidecar Container Pattern

A sidecar container is a special container that accompanies the main container in a given pod. It helps to extend and enhance the function of the main container without any complex fixing[10]. The idea of a sidecar is if something does one thing, it will do it very well. Containerized technology is to wrap everything regarding a given application to run anywhere. Each container of a pod will focus on only one thing.

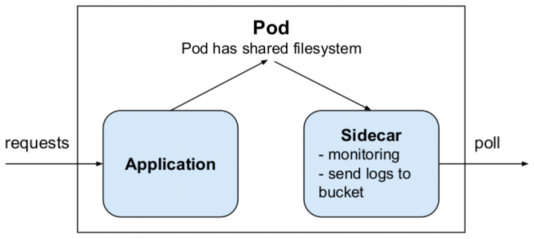


Fig. 4. Sidecar container pattern

We use sidecar containers to monitor the network service and send logs to the monitor manager. Figure 4 shown Sidecar container pattern.

### 3.5 Init Container Pattern

A init container is a specialized container run firstly in a given Pod. A Pod can have many containers that run together and one or more init containers that run before the rest launches[11]. The init containers always launch to completion, and each of them must complete successfully before the other one launches. When a init container fails, the Node's kubelet restarts it until it succeeds. Adding an initContainers field into a Pod description to define an init container. Init containers support nearly almost all of the fields of regular containers. Besides, it does not help livenessProbe, readinessProbe because it must launch to completion before the rest of the Pod's containers. Once all of the init containers launch to completion, kubelet will launch the rest of the Pod's containers and spins them as usual.

In the study, we have using init containers for two use-cases. One is to set up that is not present in a network service and a web app image. Another one is to offer a mechanism to block traffic

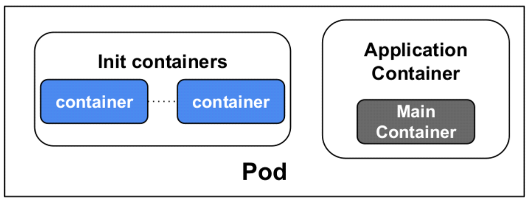


Fig. 5. Init container pattern

generator launch until the network services are ready to run the test cases. Figure 5 depicts Init container pattern.

### 3.6 Network Service Mesh

A service mesh is a dedicated infrastructure layer for handling service-to-service communication. A service mesh often sits on top of the Container Network Interface and builds on its capabilities. Network Service Mesh is a new manner to address L2/L3 use cases in the Kubernetes Network model[12]. It injects sidecar containers to each network service to communicate with the distributed network service container plane to provide functionality as traditional service meshes.

## Ⅳ. Proposed System of the Automated Testing Framework for Container-based Network Service

### 4.1 Scope of the proposed system

By filling in the gaps in current work[2-4], the proposed solution aims to have the comprehensive to automate test container-based network service. The scope of the proposed system covers multiple types of testing from the conformance to interoperability, as well performance testing of container-based network services. The procedure includes deploying the system under test a corresponding VNF that needs to be tested by vendors on the multi-cloud environment and analyzing data of the network packages.

### 4.2 Detail system architecture

The proposed system provides flexible interaction between components, as shown in Figure 6. Each component is described as following:

- Network Service is a composition of Network Functions and defined by its functional and behavioral specification. A network service can have multiple VNFs that are connected.
- Tenant provides a set of separate components for each user based on the Factory pattern. In the Factory pattern, a new set of objects is created using a common interface. The tenant plays as a common interface in the Factory pattern. It will create a set of new system components that it manages for each user.

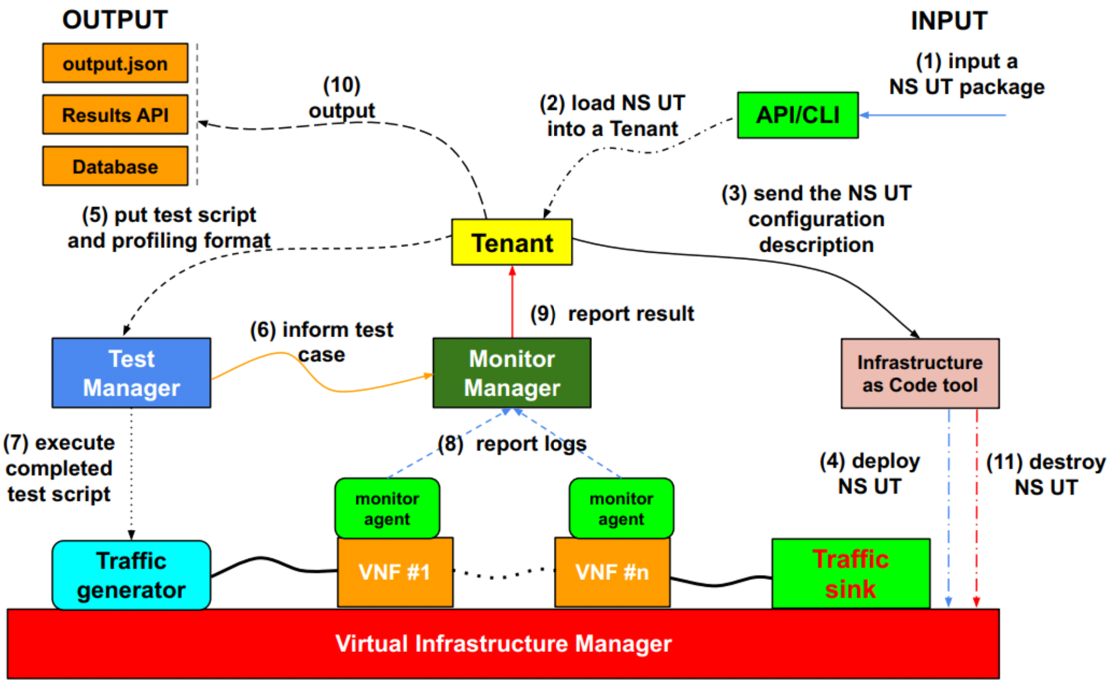


Fig. 6. The proposed framework architecture

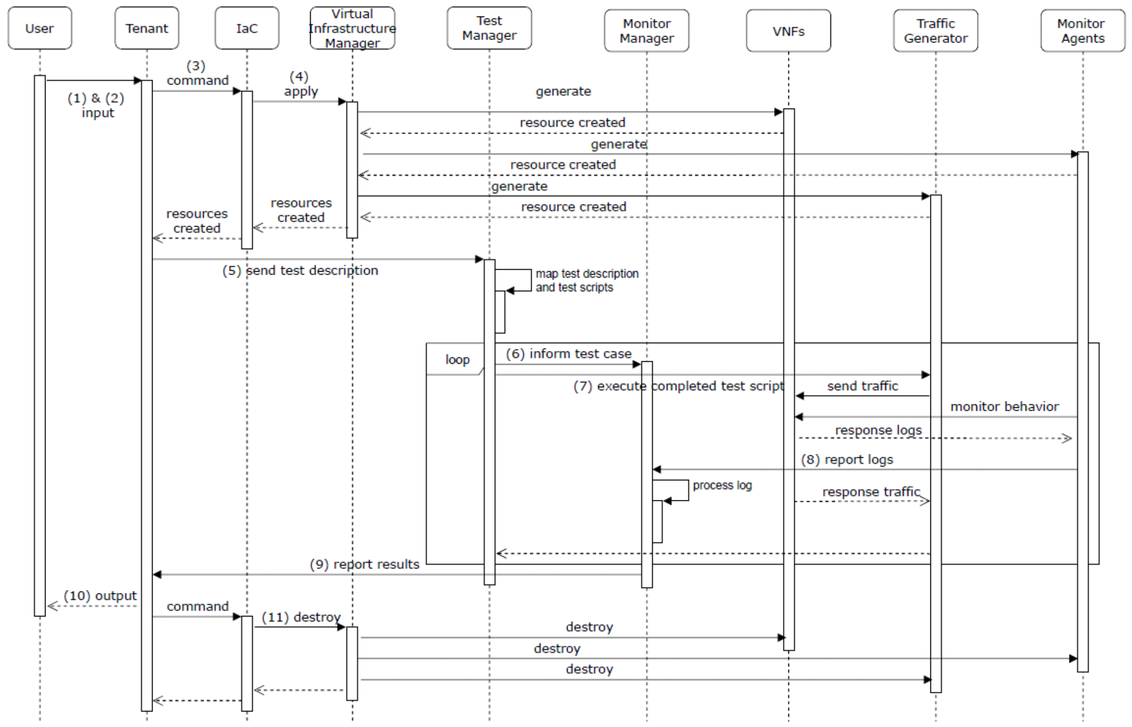


Fig. 7. Overview of the workflow

- An Infrastructure as code (IaC) tool is a consistent process of managing the IT infrastructure and provisioning virtualized resources. It helps the proposed framework automate deploy the System Under Test and independence of any particular virtualized infrastructure manager.
- Traffic generator is used to put traffic onto a network for a network service to consume. For example, the apache bench tool - an HTTP benchmarking tool.
- Traffic sink is an end-point that receives traffic from a traffic generator. For example, a web application.
- Test Manager maps test description and test script function to build completed commands that execute on the traffic generator. Besides, it provides a profiling format for each test case in the test description.
- Monitor Manager receives profiling format for each test case, and reports log from monitor agent to generate reports.
- Monitor Agents are integrated into each Virtual Network Function (VNF) of Network Service (NS) as Sidecar Container and report log of VNF to Monitor Manager

## 4.3 Workflow

The core components interaction follows the "Request-response" paradigms through a Representational State Transfer (REST) API[13] with custom JSON message formats. We present a generic workflow in Figure 7:

1. The user submits a Network Service package under test via API/CLI gateway (1). The Network Service package consists of a network function under test, test description written by YAML language, and test profiling format. A tenant that serves to isolate resources for each user will extract them (2).
2. The tenant will send the configuration under test description to infrastructure as code tool (3) to deploy the system under test on the Kubernetes cluster (4). It also sends the test description to the Test Manager (5).
3. The Test Manager waits until the system under

test has deployed successfully. Test functions built-in test manager use meta-data in the Test description as the parameters to generate completed test scripts. Then the test scripts execute on the system under test (7). At the same time, the test profiling format is also informed to the Monitor Manager to create report results (6).

4. The traffic generator follows the test script to generate traffic across the system under test. The monitor agent catches the network package and extracts needed information. After that, it sends the report to the Monitor Controller (8). Then the report result will be generated from the Monitor Controller and sends to the Tenant (9) to store in a centralized Database of the proposed system (10). When everything is finished, the system under test will be destroyed (11).

## Ⅴ. Implementaion and Evaluation

### 5.1 Configuration of the implementation

We implemented the proposed system using a set of open sources and technologies. They include Kubernetes as the target platform for managing containerized network services, Network Service Mesh as the virtual infrastructure to provide the ability to simplify connectivity between workloads (e.g., network functions) in Kubernetes. Because HAProxy includes network functions such as load balancing and caching, we use it as the VNF under test. And an HTTP Server based on the Node JS framework is a role of an endpoint server. The specification of the PC to deploy the experiment is shown in Table 1. All the connectivity is non-SSL-based.

Table 1. Specification of configuration

| Entity | Details |
|---|---|
| Compute | Intel(R) Xeon(R) CPU E3-1240 V3 @ 3.40GHz, 8 cores |
| Memory | 24GB DDR3, 1333 MT/s |
| OS | Ubuntu 18.04 LTS, 64 bit |
| Software | Kubernetes v1.20, Docker 20.10.6, Network Service Mesh v0.2.0, haproxy:2.2.11-alpine container image, NodeJS 10.0 |

The configuration details of the experiment:

- As a role for a traffic generator, we ended up using the Apache Bench tool because it is a well-known and stable tool for load testing HTTP endpoints and provides beautiful summarized results.
- The L7 load balancing feature of HAProxy is the VNF under test.
- A NodeJS server responds to the requests sending via URL paths.
- Our entire infrastructure supports HTTP protocol.

### 5.2 Results and Evaluation

The connections of the network function under test for the experiment supported by Network Service Mesh are shown in Figure 8.

Test network function interoperability in a network service requires each network function needs to pass respective conformance testing. In Figure 8, we depict three network functions are a load balancing and two web application instances. The result showing in Figure 9 validates whether the requests are distributed to two endpoints by the load balancer network function under test. If the load balancer has requests and two endpoints also have requests going through, the load balancer works correctly[14]. It means that the load balancer passes its conformance testing. Since two endpoints also receive the respective requests, the network service interoperability is also verified.

We run performance testing for the Layer 7 load balancing network function of HAProxy. In performance testing, we test the limits of the resource configurations (compute milicore and memory) of the network function under test regarding the number of requests per second and latency of response. Then, we profile the network function to find a tipping point of its respective resource configurations. We can consider the result to build a strategy to allocate the number of resources for the network function under test in production to achieve the best efficiency.

Apache Bench tool provides some of the parameters used for our load test as following:

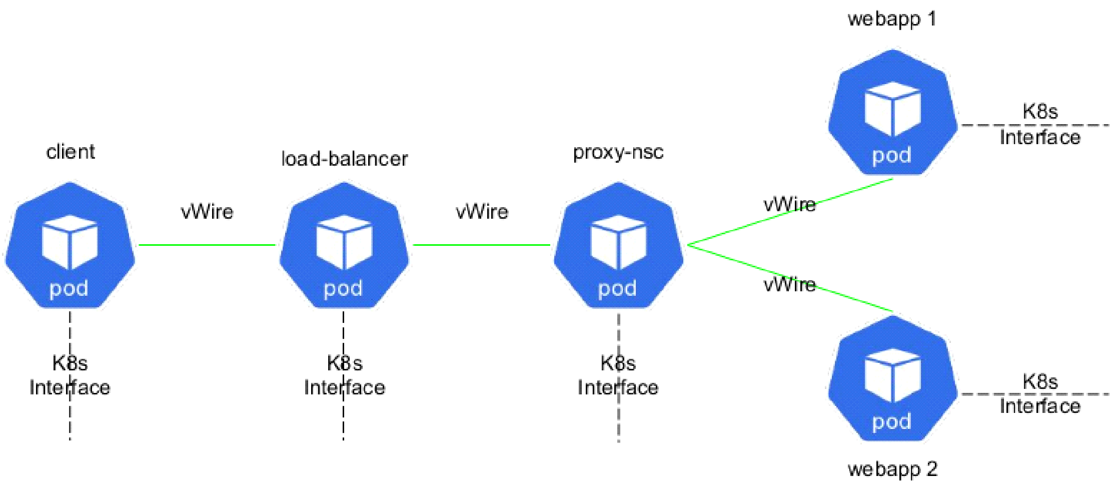
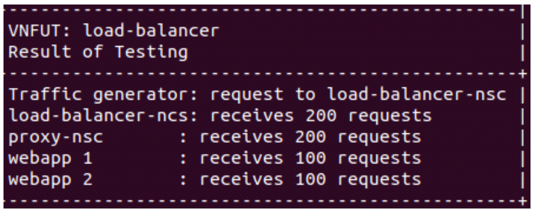


Fig. 8. Connections of component of System Under Test support by Network Service Mesh



```
-------------------------------------------------
VNFUT: load-balancer                            |
Result of Testing                               |
------------------------------------------------+
Traffic generator: request to load-balancer-nsc |
load-balancer-ncs: receives 200 requests        |
proxy-nsc        : receives 200 requests        |
webapp 1         : receives 100 requests        |
webapp 2         : receives 100 requests        |
------------------------------------------------+
```

Fig. 9. Result of testing load balancing network function

- Concurrency: the number of concurrent requests that hitting the endpoint

- Number of requests: the total number of requests of the current load run

We consider the two aspects: the number of requests per second and latency. The performance testing results, along with respective resource configurations, are shown in Figure 10. Load testing with the Apache bench tool, the goal in mind, is to find the tipping point. The above graph states that up until a certain point, if we keep increasing the

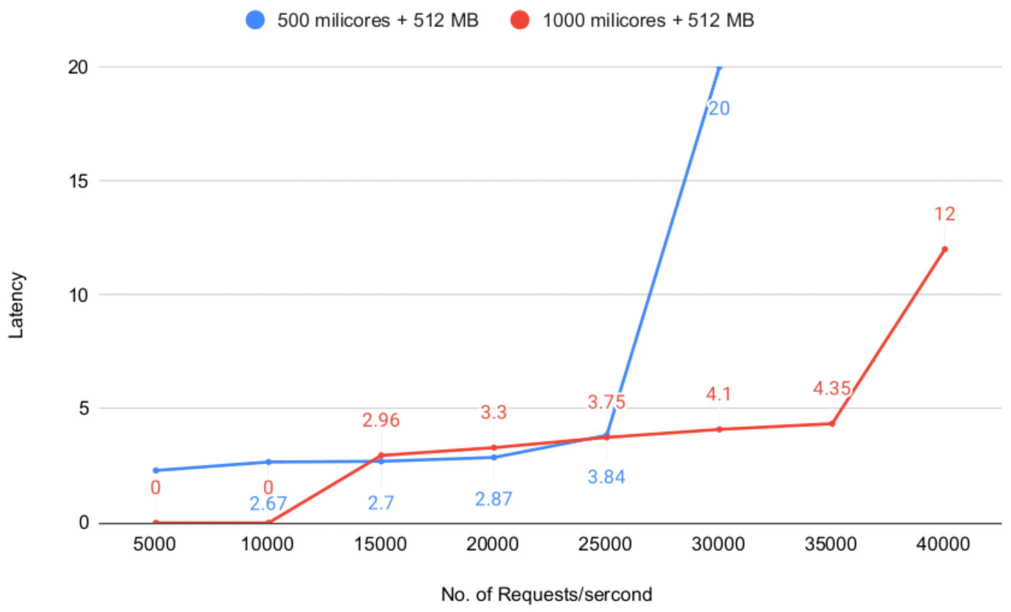


Fig. 10. The graph to identify tipping point from the result of the performance testing

number of requests, the latency will remain almost the same. However, after a certain tipping point, the latency will begin to increase exponentially. It is the tipping point for a container in a pod with the respective resource configuration (compute and memory).

## Ⅵ. Conclusion

In this study, we considered the existing efforts for VNF testing and proposed a framework to enhance fault prevention in the NFV environment. By adhering to ETSI standards to ensure future compatibility, the proposed framework provides a comprehensive solution covering conformance, interoperability, and performance testing in the NFV environment by taking advantage of enabler technologies. Besides, it offers a flexible approach to generate test scripts that we can reuse for multiple use cases. Finally, we implemented the proposed framework and verified it with a test case.

## References

[1] ETSI, "*Network function virtualisation (NFV); Architectural framework*," [Online] https://www.etsi.org/deliver/etsi\tr/103400\103499/103495/01.01.01\60/tr\103495v010101p.pdf.

[2] R. V. Rosa, C. Bertoldo, and C. E. Rothenberg, "Take your VNF to the gym: A testing framework for automated NFV performance benchmarking," *IEEE Commun. Mag.*, vol. 55, no. 9, pp. 110-117, Set. 2017.

[3] *OPNFV Yardstick*, [Online] Available: https://wiki.opnfv.org/display/yardstick.

[4] *OPNFV Bottlenecks*, [Online] Available: https://wiki.opnfv.org/display/bottlenecks.

[5] ETSI, "*Network function virtualization (NFV); Testing methodology; Report on NFV interoperability testing methodology*," [Online] https://www.etsi.org/deliver/etsi_gs/NFV-TST/001_099/002/01.01.01_60/gs_NFV-TST002v010101p.pdf

[6] M. Zhao, et al., "Verification and validation framework for 5G network services and apps," *NFV-SDN*, Berlin, Germany, Nov. 2017.

[7] *Docker*, [Online] Available: https://docs.docker.com.

[8] R. Cziva, S. Jouet, K. J. S. White, and D. P. Pezaros, "Container-based network function virtualization for software-defined networks," *ISCC 2015*, pp. 415-420, Larnaca, Cyprus, Jul. 2015.

[9] *Kubernetes*, [Online] Available: https://kubernetes.io/docs/home/

[10] *Sidecar container pattern*, [Online] Available: https://medium.com/bb-tutorials-and-thoughts/kubernetes-learn-sidecar-container-pattern-6d8c21f873d

[11] *Init container pattern*, [Online] Available: https://kubernetes.io/docs/concepts/workloads/pods/init-containers/

[12] *Network Service Mesh*, [Online] Available: https://networkservicemesh.io/docs/concepts/what-is-nsm/

[13] R. T. Fielding and R. N. Taylor, "Architectural styles and the design of network-based software architectures," Ph.D. dissertation, Dept. Info. and Comput. Sci. Univ. California, 2000.

[14] C. Le and M. Yoo, "An extended agent-based mechanism for testing service function chain," *2020 ICTC*, Jeju, Korea, Oct. 2020.

**레 반끄엉** (Van-Cuong Le)

Van-Cuong Le received the B.Eng. degree in software engineering from the University of Science & Technology, University of Da Nang, Da Nang City, Vietnam, in 2018. He is currently pursuing a master's degree with Soongsil University. His research interests include Software-defined Network/Network Function Virtualization.

**유 명 식** (Myungsik Yoo)

Myungsik Yoo received his B.S. and M.S. degrees in electrical engineering from Korea University, Seoul, Republic of Korea, in 1989 and 1991, and his Ph.D. in electrical engineering from State University of New York at Buffalo, New York, USA in 2000. He was a senior research engineer at Nokia Research Center, Burlington, Massachusetts. He is currently a professor in the school of electronic engineering, Soongsil University, Seoul, Republic of Korea. His research interests include visible light communications, sensor networks, Internet protocols, control, and management issues.
[ORCID:0000-0002-5578-6931]