Proceedings of the APAN – Research Workshop 2017
ISBN 978-4-9905448-7-4

Comparison of Service Description and Composition for Complex 3-Tier Cloud-based Services

Moonjoong Kang and JongWon Kim

***Abstract* — This paper focuses on service description and composition for complex 3-tier datacenter application services, tied with firewalls and load balancing. By adopting cloud-native container-based microservices architecture (MSA) for a small-sized datacenter situation, we attempt to compare several approaches for service description and composition, especially from the viewpoint of service function chaining (SFC). Also we prototype them with OpenStack-based cloud virtual machines (VMs) by comparing their performances with resource usages.**

***Index Terms* — Cloud-native computing, microservices architecture, container orchestration, service description and composition, and service function chaining.**

I. Introduction

With the advent of cloud-first computing era, the value chain around cloud industry has been rapidly growing. This led to gradual migration of the specialized application services over dedicated clusters to cloud-based shared infrastructures [1, 2]. Following this trend, the service oriented computing paradigm for diversified application services is transforming to so-called microservices architecture (MSA) that stitches together multiple openAPI-based component services (i.e., functions) to compose a whole composite service. This MSA is known for several benefits such as dynamic agility, easy and flexible maintenance, and cost effectiveness due to shared resource pooling [3].

Typically, the service composition for MSA-based application services is done by service function chaining (SFC). With SFC, we start with allocating the necessary resource slices from shared cloud infrastructure to accommodate all component services (i.e., functions). These functions and their stitching requirements are to be satisfied by enabling diverse inter-connections among them.

However, the effectiveness of SFC-based service description and composition is not an easy target for the complicated form of datacenter Web-App-DB 3-tier application services, which may include additional firewalls and load balancing [4]. There are several approaches to handle this kind of complex SFC-based service description and composition in general. Thus, in this paper, we are attempting to explain and compare them by choosing example complex Web-App-DB 3-tier application services. Also we attempt to understand the whole procedure behind service description and composition by prototyping the realization of SFC-based service description and composition and by evaluating their performance/cost in orchestrating SFC-based service description and composition.

This work was supported by Institute for Information & communications Technology Promotion (IITP) grants funded by the Korea government (MSIT): No. R7117-16-0218 (Development of automated SaaS compatibility techniques over hybrid/multisite clouds) and No. 2015-0-00575 (Global SDN/NFV open-source software core module/function development).

Moonjoong Kang and JongWon Kim are with the school of Electrical Engineering and Computer Science, Gwangju Institute of Science and Technology (GIST), 123 Cheomdangwagi-ro, Buk-gu, Gwangju, 61005, Republic of Korea (e-mail: {mjkang, jongwon}@nm.gist.ac.kr).

II. SFC-based Approaches for Service Description and Composition

We compare three SFC-based service description and composition in this paper: container-based MSA-SFC for web-based SaaS (Software as a Service) applications, HOT(Heat Orchestration Template)-SFC for OpenStack Heat-based (cloud-integrated or Web-based) SaaS applications, and IETF (Internet Engineering Task Force) NSH(Network Service Header)-SFC for network infrastructure-focused SaaS applications.

Regarding service description aspects, all 3 SFC approaches commonly describe application services as a set of abstracted functions with ordering constraints. They include identifiers for all component functions, access interfaces for function binaries (e.g., embedded scripts or URI), and the directional dependency relation among functions. IETF NSH-SFC, however, may include non-abstracted functions, which are 1-to-1 matched to specific physical machines and do not require function binaries. Also IETF NSH-SFC is unique in handling overlay networking by relying on NFV(Network Function Virtualization)-enabled network infrastructure, while other SFCs rely on service-transparent encapsulation-based

TABLE I
SERVICE DESCRIPTION AND COMPOSITION APPROACHES: DIFFERENCES.

Type	Container-based MSA-SFC	OpenStack Heat-based HOT-SFC	IETF NSH-SFC
Application services	Web-based SaaS	Web-based and cloud infrastructure-integrated SaaS	Network infrastructure-focused SaaS
Encapsulation	Transparent to application services		All functions need explicit NSH encapsulation or service function proxy
Flow tracking	Not available		NSH includes whole path info for packets
Service function stitching	No implementation required for high availability	Needs its own implementation for high availability	No implementation required for high availability

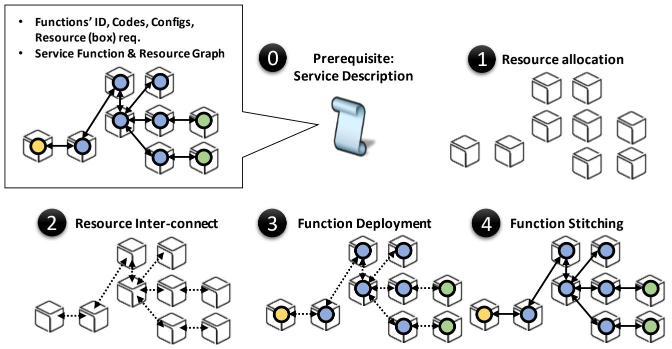
Functions' ID, Codes, Configs, Resource (box) req.
Service Function & Resource Graph
0 Prerequisite: Service Description
1 Resource allocation
2 Resource Inter-connect
3 Function Deployment
4 Function Stitching

Fig. 1. Step-by-step procedure for SFC-based service description and composition.

overlay networking [5-8]. Because of this, IETF NSH-SFC introduces NSH concept [9], where any involved function must handle NSH directly or be wrapped with service function proxy[1]. Also, due to the unique encapsulation for overlay networking, IETF NSH-SFC is more effective in monitoring and adjusting the complete routes of packets than other approaches [9]. HOT-SFC covers OpenStack cloud infrastructure [8], but its encapsulation is completely delegated to OpenStack Neutron and transparent to its service.

Next, regarding the service composition aspects, 3 SFC approaches mostly share similar step-by-step procedure depicted in Fig. 1. The SFC orchestration tool first parses the description for service composition and checks the resource requirements of involved functions. If the resource requirement is met, the orchestration tool proceeds to identify and allocate demanded resources for involved functions by considering their adjacency for stitching efficiency. Once resources are allocated, the orchestration tool performs the necessary interconnections to establish flexible networking of resources. Then, all involved functions are appropriately deployed (i.e., located, placed, and instantiated). Finally, these deployed functions are activated and stitched together to establish end-to-end composite service composition.

Also, during the function stitching, all 3 SFCs can manage the interconnections among functions via KV (key-value) storage. With function identifier as its key, we can orchestrate the scaling and load balancing of service composition to mitigate the service down time. However, OpenStack Heat-based HOT-SFC provides function stitching only to OpenStack-integrated functions while leaving the handling of web-based SaaS application services to own implementation.

[1] Service function proxy translates NSH for non-NSH-aware functions. Every inbound traffic must first go through a classifier that attaches NSH while NSH is detached for outbound traffic.

III. CONTAINER-BASED MSA-SFC SERVICE DESCRIPTION & COMPOSITION

Now, by choosing container-based MSA-SFC service description and composition, we explain the whole procedure behind service description and composition by prototyping the realization of SFC-based service description and composition and by evaluating their performance/cost in orchestrating SFC-based service description and composition. Note that MSA-SFC does not need any change of the application service to be composed as it does not need explicit encapsulation at the service level. Also it may provide high availability for Web-App-DB 3-tier application services without additional own implementation.

Now for container-based MSA-SFC service description and composition of complex Web-App-DB 3-tier cloud-based services with more than 10 functions, we compare two popular container orchestration tools: Docker Swarm and Kubernetes. As shown in Fig. 2, both Docker Swarm and Kubernetes follow almost identical workflows in terms of service description and composition as follows. The description for service composition uses string-based function identifier. Each function can be given with required (and optional) resource amount and identifier-based dependency configuration. Also,

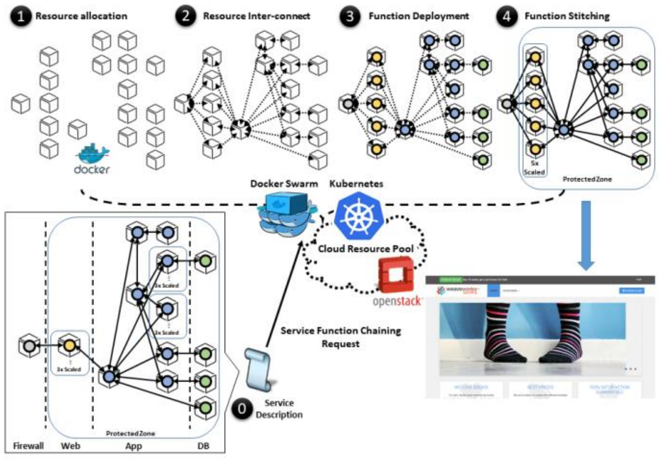

Fig. 2. Example SFC for socks shop applications with additional firewall and 3x scaling to selected functions via Docker Swarm or Kubernetes.

TABLE II
AVERAGE AND STANDARD DEVIATION OF COLLECTED METRICS DURING LOAD TEST ON COMPOSED SERVICES

Tool	Node Type	Value	CPU Active %	RAM Util %	Ethernet Bytes Received	Ethernet Bytes Sent	Disk Bytes Read	Disk Bytes Written
Docker Swarm	Manager	μ	0.86 %	44.31 %	1009086.44	966419.12	0.00	4741.95
		σ	0.76	0.02	361933.12	345394.14	0.00	10297.25
	Worker	μ	266.52 %	146.04 %	1594812.37	1594812.37	0.00	1045358.23
		σ	67.75	7.34	494473.55	494473.55	0.00	798980.66
Kubernetes	Manager	μ	4.67 %	29.62 %	1013737.24	974305.11	0.00	60993.94
		σ	2.30	0.04	363253.35	346119.57	0.00	40172.65
	Worker	μ	375.11 %	154.86 %	2214655.04	2214655.04	394567.97	32735233.68
		σ	81.74	11.87	669057.69	669057.69	3288517.74	9230700.90

μ = Average, σ = Standard Deviation

after parsing the description for service composition, the orchestration tool allocates resource boxes matching the requirements of lightweight Docker container functions. Then the orchestration tool makes a dedicated and isolated overlay networking by interconnecting all resource boxes. Binaries for involved functions are downloaded from Docker Hub and deployed (i.e., located, placed, and instantiated) to the resource boxes. Remember that the interconnections for the resource boxes are managed via KV-based storage.

However, while both orchestration tools allocate box-style resources for the service composition in the form of Docker containers, Docker Swarm orchestration tool is tightly integrated with extended Docker engine. In comparison, Kubernetes orchestration tool utilizes only Docker containers with Docker APIs and wraps it with other open-source tools for the required orchestration. Kubernetes deploys its own functions as additional Docker containers running inside resource boxes while Docker Swarm does not. For example, when interconnecting allocated resource boxes, Docker Swarm can provide its own native network driver to form overlay networking for the application services to be composed. On the contrary, Kubernetes does not provide any native counterparts and the operator must choose a network addon from 3rd parties.

IV. EXPERIMENT ENVIRONMENT AND RESULTS

While Docker Swarm and Kubernetes can orchestrate container-based MSA-SFC service description and composition, they have quite different architecture that may lead to performance differences. Thus, in order to evaluate the service description and composition for complex Web-App-DB 3-tier datacenter application services, an experiment environment is built based on two types of distributed OpenStack clouds to simulate remote users accessing the cloud-leveraged services. The primary OpenStack cloud consists of 3 boxes with Intel Xeon E5-2640v3, 24GB DDR4 ECC Register RAM, 400GB Intel 750 NVMe SSD and each box is distributed to 3 different sites. The secondary OpenStack cloud is based on one box (with the same specification with the former cloud boxes) as OpenStack control node, another 4 Supermicro SYS-E200-8D boxes with Intel Xeon D-1528, 32GB DDR4 ECC RAM, 500GB Samsung SSD. Each experiment is performed mostly on the VMs with the same configuration of 4 vCPU cores, 8GB RAM, and 40GB storage.

To eliminate any possible interference with each other, the service composition by Docker Swarm and Kubernetes are respectively realized at the different sites of the primary cloud. The cluster configurations of both orchestration tools are almost identical with one VM as a manager node, other two VMs as worker nodes. Also service function is not scheduled to the manager node. For container networking, Docker Swarm is configured use its native network driver and Kubernetes is configured to use Weave Net network addon from Weaveworks. The application services are using overlay networking to interconnect all involved functions. Also, both orchestration tools are configured to interconnect resource boxes using the same network interface listening for inbound connections from outside.

Also, for the example application services for the composition of both clusters, we choose socks shop application services from Weaveworks by considering its similarity with the complex application services used for production and its load-testing function that simulates users to test the composed service. We also use Linux kernel's Netfilter firewall function via Docker Swarm and Kubernetes to block any unintended access to the application services and modify its service description to scale socks shop's functions (i.e., 1 Web and 2 App functions with 3 replications for load-balancing). To perform load-testing on the composed application services, the load-testing function is placed at the last unoccupied sites of the primary and secondary clouds. Each load-testing function generates 3 clients and 40000 requests to the orchestration tools, respectively. Therefore it generates 6 clients with 80000 requests to the service composed by each orchestration tool from 2 remote places.

Our evaluation of service description and composition tool is focusing on resource usage of the composed services for the same 3-tier application. For the measurement of evaluations, we use Intel Snap telemetry framework and place its agents to each VM and collect CPU utilization percentages, RAM usage percentages, disk read/write bytes per second, and network interface sent/received bytes per second[2]. Collected metrics are stored into InfluxDB time-series database on another VM located at the secondary cloud.

Thus, as shown in Table II, when comparing CPU active percentages for manager node, Kubernetes shows slightly higher usage than Docker Swarm. For worker nodes, this slight

[2] These resource metrics represents the usage of computing, storage, networking resources.

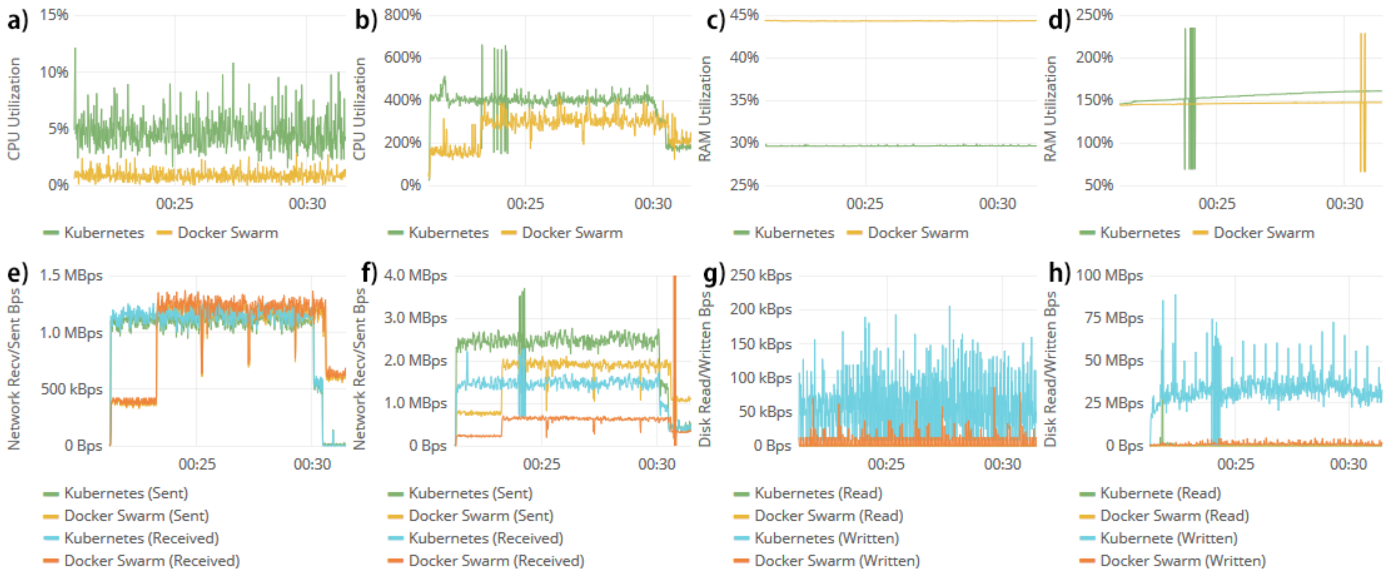

Fig. 3. Collected metrics during load testing on the composed services. From left to right, top to bottom: a) CPU utilization on manager nodes, b) CPU utilization on worker nodes, c) RAM utilization on manager nodes, d) RAM utilization on worker nodes, e) Received/sent bytes via network on manager nodes, f) Received/sent bytes via network on worker nodes, g) Read/written bytes from/to disk on manager nodes, and h) Read/written bytes from/to disk on worker nodes.

gap grows and reaches the average difference of 108.53%[3]. Memory utilization percentages show that Docker Swarm is using larger RAM from its manager node. More specifically it shows about 14.68% more than Kubernetes. Also, both environments show almost constant usages during the whole load testing. For the sum of the metrics from worker nodes, however, Kubernetes is 8.82% higher performance than Docker Swarm. Network interface sent/received bytes per second metric shows that Kubernetes generates more traffic than Docker Swarm with bytes received 4.54KB/s higher at manager node and 605.32KB/s higher overall at worker nodes and bytes received 7.70KB/s higher at manager node and 605.32KB/s higher overall at worker nodes. While both Kubernetes and Docker Swarm never read their manager node's disk as the metric always stays at 0, Kubernetes shows that bytes written only 54.43KB/s more than Docker Swarm. But Docker Swarm shows more constant rates than Kubernetes', with the difference of standard deviation by about 29,875. However, the most contrasting result is the metric of written bytes to disks at worker nodes, as Kubernetes is 30.22MB higher than Docker Swarm and shows 8431720.24 as higher standard deviation. As seen in Fig. 3, Docker Swarm shows almost no bytes written compared to Kubernetes.

Thus, by considering all the collected metrics, we believe that Docker Swarm is more effective than Kubernetes in terms of resource usage for the composition of complex Web-App-DB 3-tier application services.

V. Conclusion

This paper performed the comparison of service description and composition for complex Web-App-DB 3-tier datacenter application services by comparing Docker Swarm and Kubernetes on OpenStack-based cloud VMs. Also for Socks shop application service described and composed, VM's resource usage was collected during the load testing and evaluated.

[3] The scale of one core's full utilization is set to 100%.

References

[1] N. Kratzke and Q. Peter-Christian, "Understanding cloud-native applications after 10 years of cloud computing - A systematic mapping study," *Journal of Systems and Software*, vol. 126, pp. 1-16, Apr. 2017.
[2] N. Dragoni et al, "Microservices: Yesterday, today, and tomorrow," *in e-print arXiv:1606.04036*. June 2016.
[3] K. Karanasos et al, "Mercury: Hybrid centralized and distributed scheduling in large shared clusters." *in Proc. USENIX ATC*, 2015.
[4] J. Stubbs, M. Walter, and R. Dooley, "Distributed systems of microservices using docker and serfnode," *IEEE International Workshop on* IEEE *Science Gateways (IWSG)*, 2015.
[5] V. Marmol, R. Jnagal, and T. Hockin, "Networking in containers and container clusters," *Proc. of NetDev* 0.1, Feb. 2015.
[6] B. U. I. Tuan-Anh et al, "Cloud network performance analysis: An OpenStack case study," 2016.
[7] A. L. Kavanagh, "OpenStack as the API framework for NFV: The benefits, and the extensions needed," *Ericsson Review* 2, 2015.
[8] Y. Yamato et al, "Development of template management technology for easy deployment of virtual resources on OpenStack." *Journal of Cloud Computing*, vol. 3. No. 1, July 2014.
[9] J. Halpern and C. Pignataro, *Service function chaining (sfc) architecture*, IETF RFC 7665. 2015.