Real-time Monitoring System for Container Networks in the Era of Microservices

Takashi Shiraishi Masaaki Noro Reiko Kondo Yosuke Takano and Naoki Oguchi

DevOps Innovation Unit, FUJITSU LABORATORIES LTD.

4-1-1 Kamikodanaka, Nakahara-ku, Kawasaki, 211-8588, Japan

E-mail: shiraishi-ten@fujitsu.com

***Abstract*—Large-scale web services are increasingly adopting the microservice architecture that mainly utilizes container technologies. Microservices are operated on complex configured infrastructures, such as containers, virtual machines, and physical machines. To ensure service quality of microservices, it is important to monitor not only the quality of services but also the quality of the infrastructures utilized by the services. Therefore, the metrics of the infrastructure related with the services should be traced. An extended Berkeley Packet Filter (eBPF) is a relatively new Linux's function, which is effectively used as a sensor of container-network metrics. There are two key challenges in realizing the service-linked monitoring system. One challenge is making the full-stack topology between microservices, containers, and machines visible to set the sensor related with the services. Another challenge is dynamic sensor management that can relocate the sensor quickly after the topology's change. In this paper, we propose a real-time monitoring system that creates a full-stack topology and relocates the sensor in conjunction with events from a container orchestrator. The system enables a dynamic deployment of the sensors related with the monitored services.**

Keywords—Microservice, Container, Kubernetes, eBPF, Network latency

I. Introduction

Today, agility in system development and operation is increasingly required to flexibly respond to customer needs. Conventional monolithic service applications have difficulties in terms of agility, so the adoption of a microservice architecture is accelerated [1]. In a microservice architecture, a service is built using multiple, weakly coupled microapplications. Traditionally, services are deployed and operated on virtual machines (VMs). The microservice architecture is likely to combine with recent containerization technologies, such as Docker. Compared with VMs, containers have advantages of a cloud-native model, which includes greater agility in software development and portability across cloud environments. A container orchestrator enables automatic management of containers. Today, it is becoming a standard that microservices deployed on multiple containers should be operated by utilizing container orchestrators, such as open-source Kubernetes (K8s). Tens of thousands of microservices are actually operated in hundreds of thousands of containers within a single data center [2]. A service consists of several microservices which propagate requests each other. Each microservice is executed on a container, and the container is stored in a pod that is a management unit of resources in K8s. A pod can contain many containers that share a Linux namespace.

Maintaining the service quality is an important work for service operators. For example, an end-to-end latency of requests through the microservice is one of the key qualities [3]. Once the service quality is degraded by some failures, the operator must quickly identify the root causes of the failures and recover the quality of the service. However, maintaining the service quality or quickly analyzing the failure cause of the microservice is challenging because granular service applications are distributed on various infrastructures in the cloud or on-premise environment and many degradation factors (application side or infrastructure side) affect the service quality.

Therefore, the service quality of the microservices and the metrics of the infrastructure utilized by the microservices should be monitored. In recent years, technologies, such as application performance management (APM), has been utilized for service quality monitoring. By using APM, the end-to-end latency of microservices can be monitored at the service level. However, there is a difficulty in monitoring infrastructure metrics in which microservices are executed. The topology of the infrastructure becomes complicated due to an introduction of a container and frequent changes in the topology. For example, K8s has various automatic management functions for containers. A self-healing function for containers automatically move pods for the container from an abnormal VM to another VM. To quickly analyze the cause of a system failure and maintain the service quality, the infrastructure metrics related to the service should be consistently monitored in real time and past metrics related with the service should be referred to even though the infrastructure topology changes during its monitoring period.

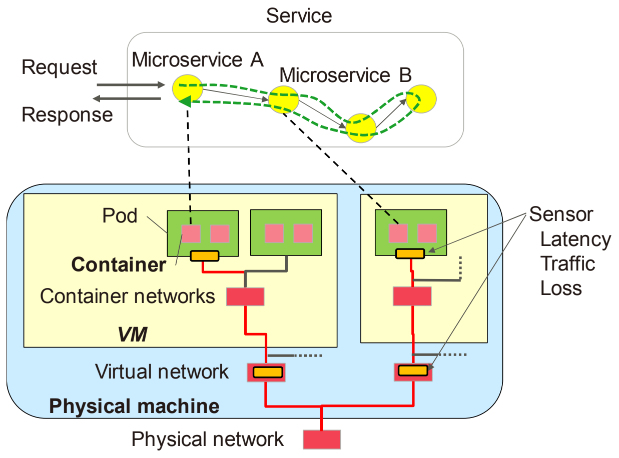


Fig. 1. Concept of our proposed monitoring system for a microservice-deployed infrastructure.

In this work, we propose a real-time monitoring system that can collect the service-linked network metrics of infrastructures using eBPF sensors for containerized microservices. Fig. 1 shows an conceptual view of our proposed monitoring system. Our system enables deploying sensors selectively to the infrastructures where monitored services are utilized. The sensor can be flexibly set to the virtual network devices of the service communication path based on the eBPF technology. This feature helps to analyze the location of network troubles. Moreover, the monitoring system enables quickly relocating the sensor depending on the linked service when the system topology changes.

The rest of the paper is organized as follows: In Section II, we first explain the related works. In Section III, we introduce technical issues that we tackle. In Section IV, we explain the real-time monitoring system we developed. In Section V, we evaluate the proposed monitoring system. In Section VI, we conclude this paper.

II. Related Works

A distributed tracing technology is a kind of APM that has been developed for monitoring the service quality of distributed applications. Dapper is the first reported article on a large-production distributed tracing framework [4]. Unique span IDs were assigned to each service request and transmitted through the distributed services. Application logs, such as timestamps, were analyzed with the span IDs to identify the relationship between the service and application logs. Service latency between distributed service applications can be monitored without the need for any application-level modif cations. Jaeger [5] and Zipkin [6] are open-source frameworks that evolved based on Dapper's technology to monitor the service quality of microservices. However, the latency measured by these tools contain processing times in containers and network latencies in virtual and physical networks. Therefore, it is difficult to determine whether the cause of a degradation is in the application or infrastructure side when the service latency increases.

cAdvisor was recently developed by Google Inc. and has since been a widely used tool that can collect infrastructure metrics linked by containers [7]. This software can measure various server metrics, such as CPU load and memory usage rates utilized by the container. However, when the containers are on VMs, cAdvisor can only collect the metrics in the VMs linked by the containers. The metrics in the physical machine linked by the containers cannot be collected. Moreover, the network metrics is not sufficient for failure analyses. It is not possible to obtain data, such as packet information or network latency, at the container level.

The conventional packet-capturing technique on the Linux OS has been used for dumping raw packets. The network latency monitoring of containers is reported using the packet-capturing technique of a tcpdump tool [8]. The packets of the application container are copied to the adjacent dummy container for monitoring. The tcpdump tool is quite useful to analyze network troubles, but it is not suitable as the sensor of the real-time monitoring system due to the relatively high consumption of system resources and the small packet receive ratio [9].

eBPF is an extended form of the Berkley Packet Filtering (BPF) designed to be a general-purpose monitoring and f ltering tool in the Linux system. User-defined eBPF program can be run in the system space of the kernel. The eBPF programs can be called from different layers of the network, e.g., socket, qdisc, and drivers. This feature enables eBPF to process the captured packets before forwarding them to the subsequent layer [10]. It is an attractive candidate for the system monitoring tool due to the low resource usage of the system.

III. Technical Issues

In this work, we developed a service-linked real-time monitoring system using the eBPF as a packet monitoring sensor. There are two technical challenges to realize our proposed system.

A. Full-stack topology between different layers

Using the eBPF as a sensor, the sensor can capture packets passing through a virtual network interface card (VNIC) of a pod with a container. A linkage system between the service information and the metrics of the packet should be established because the eBPF is essentially executed in kernel space of a VM and there are no information of the container and service from which the packets come. The pod of the container is managed by a container orchestrator, and the container itself is executed by a container runtime software, such as Docker. To associate the eBPF metrics to the service identifier, the acquired metrics with the related service-identification information should be recorded to the database, and the metrics should be retrieved using the service identifier as a key from the database, as shown in Fig. 2. For this purpose, a mechanism is necessary to selectively deploy sensors to the monitored service. However, creating a full-stack topology including the services, containers, and VMs is challenging. Because these configuration elements are managed by the different management software (Docker, K8s and Linux OS of VMs), they are not expressed by a unified format. There is a difficulty in combining these elements as the unified full-stack topology.

It is also possible to capture all packets in the VMs with a sensor. For instance, there is a technique that can be used to analyze the topology of networks in a data center using the captured packets [11]. However, it drastically consumes resources to send many extraneous packet data.

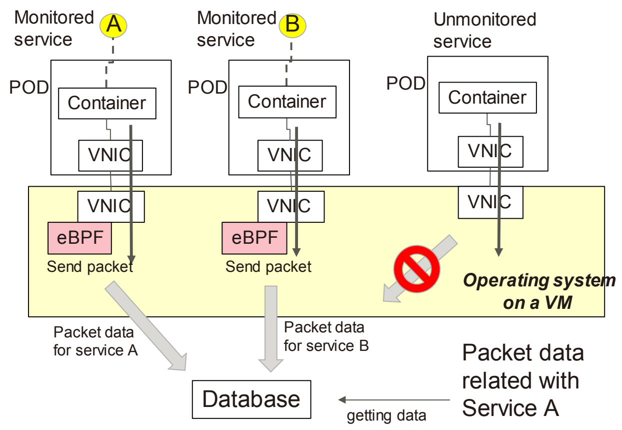


Fig. 2. Overview of sensor deployment combined with the service.

B. Dynamic sensor deployment driven by a topology change

As shown in Fig. 3, there are many cases where containers move from a VM to another VM due to a failure or setting change by an operator. During this time, the eBPF sensor installed in the VM cannot collect metrics. To collect the metrics again, it is necessary to recreate the full-stack topology and relocate the sensor in accordance with the new topology. It is possible to periodically acquire the topology and check the differences, but many failures often occur at the timing of the topology changes in the actual operation of the system [12]. The dead period during which the metrics cannot be collected should be as short as possible. Therefore, a mechanism that can detect the movement of the container and quickly recreate the full-stack topology and relocate the sensor is necessary.

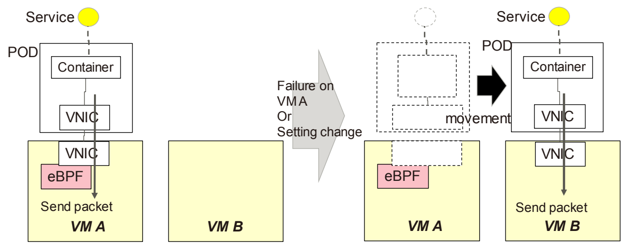


Fig. 3. Issue in metrics monitoring by the eBPF in the case of container movement.

IV. Real-time monitoring system

We examined two technical issues mentioned in the previous chapter and developed a real-time monitoring system for a container network. Fig. 4 shows an overview of the developed monitoring system. We assumed a recent typical configuration of system where multiple VMs are connected through Open vSwitch (OVS) in multiple physical machines. Multiple containers are executed in the VMs, and microservices are operated on the containers. We used OpenStack for the management of the VMs and Docker and K8s for the management of the containers.

Our monitoring system consists of distributed software units connected to one another through representational state transfer application program interface (REST API). First, a pair of services to be monitored is set in a sensor management unit. The sensor management unit calls a topology analysis unit. The topology analysis unit analyzes the setting information acquired from K8s, Docker, and Linux OS of the VM to create a full-stack topology across the layers. Based on the acquired topology, the sensor management unit instructs agents of sensors in the VM in which the monitored service is running. Each agent deploys and deletes the eBPF sensors program based on the instruction. The metrics acquired by the sensor are associated with the monitored service information and registered to a database in a metrics collector unit.

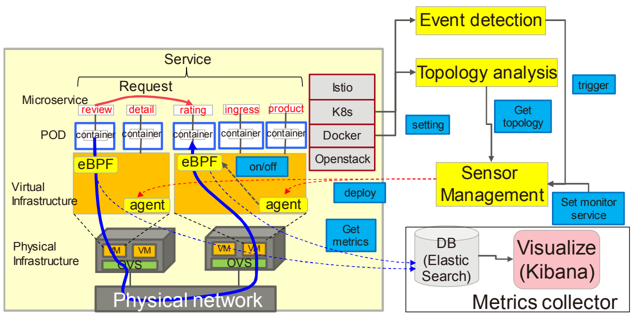


Fig. 4. Our developed real-time monitoring system

We demonstrate the monitoring system to the monitoring of latency between containers. This concept with the topology enables a real-time service-linked monitoring of various network qualities of infrastructures that those services utilize, including containers, VMs, OVS, and physical machines.

An event detection unit constantly observes kube-apiserver of K8s [13] and triggers the creation of the full-stack topology by the topology analysis unit when pods move. This process enables the dynamic relocation of the sensors when the pod configuration has changed.

A. Topology analysis unit

The eBPF can acquire the packet that passes through a VNIC by specifying its device name in the VM. Therefore, it is necessary to create the full-stack topology that includes the relationship and the elements of the service container and container network. The topology analysis unit automatically analyzes the topology by extracting the elements from the management software and creating connection between the elements, through the following procedure. It takes a great deal of cost for the operator to manually analyze it.

1) Extract relationships between services and pods and the container's name inside the pods from the kube-apiserver.

2) Get a process IDs of the containers from Docker apiserver in each VM and match the process IDs and the services.

3) By utilizing the process ID, connect to the namespace of the pods and investigate the VNICs (veth) to which the containers are connected.

4) Login to the VMs and investigate container networks to which the VNICs are connected.

Fig. 5 shows an example of the created full-stack topology and a block diagram of the topology analysis unit. Containers stored in the pod use separated namespace from other pods by the cgroup function, which enables the isolation of resource usage for group of processes in a Linux system. The network of the container inside the pod is connected to the OS space of the VM with a VNIC pair (called a veth pair). A container network in the VM consists of a bridge, a container network interface (CNI), and external network interface. The bridge (cni0) is used for pods to communicate with other pods inside the same VM. To communicate other pods of other VMs, the CNI is used to resolve IP address conflicts [14]. Flannel is utilized as a CNI. Flannel provides a mechanism that encapsulates the packet that came to an interface of flannel.1 into VXLAN packets and sends it from the external network interface (eth) of the VM. Each configuration element (such as service, pod, container, eth0, veth, cni0, flannel.1 and eth) is expressed as model by unified format in the topology.

Based on the request from the sensor management unit, the topology analysis unit automatically collects the setting information through the REST API of K8s and Docker or through command line interface (CLI) of VM. The service information is associated with a container using the collected information from K8s and Docker. The container and container network in the VM are matched with the information from Docker and Linux OS. In addition to the topology, information, such as network interface name necessary for sensor deployment and the IP address of the pod necessary for packet filtering, are also acquired.

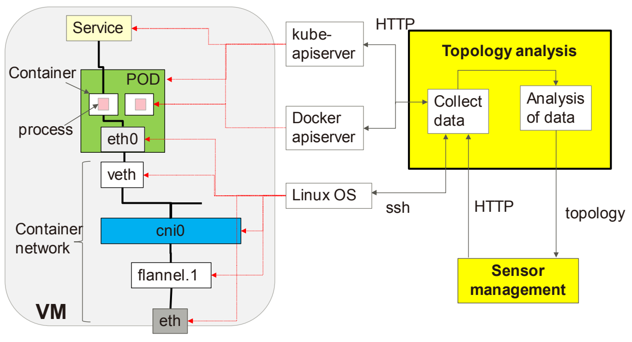


Fig. 5. Block diagram of a full-stack topology analysis unit

B. *Event detection unit*

K8s has various built-in controllers. These controllers observe the status of pods and other resources. When the state of the resources changes, the controller recovers the state as it is described in the setting. The controller is written by Go language with client-go library. By leveraging the library, various custom controllers can be created and used with the K8s system. The event detection unit is a custom controller that observes K8s events. In this research, we revised an open-source controller called kubewatch and used it as an event detector [15].

Fig. 6 shows a block diagram of the event detection unit. The config block determines the K8s resource to be monitored. In this work, pod events are monitored. A detection block observes the kube-apiserver for the events. Events, such as "create," "delete," and "update," for any pods can be monitored. When one of the specified events of the pod occurs, the event handler sends an HTTP request to the sensor management unit and the topology analysis units to create the topology and then relocate the sensor to the infrastructure. For example, when a pod moves due to the failure of the host VM, a copy of the pod is first created ("create event"). After the new pod has normally started, the original pod is deleted ("delete event"). In this way, the movement of the pod can be detected by watching the "delete event" without burden.

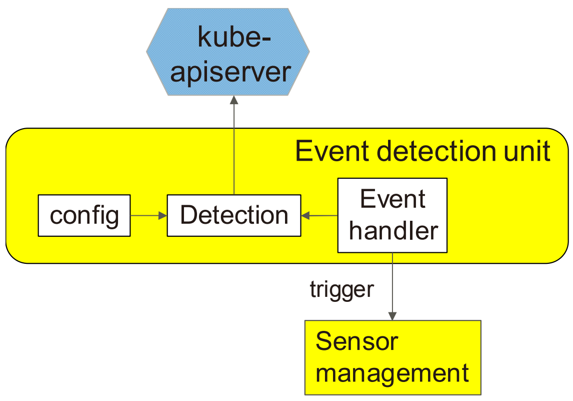


Fig. 6. Block diagram of an event detection unit

C. *Sensor management unit, agent unit, and metrics collector unit*

The sensor deployment and metric collection were controlled by a sensor management unit, an agent unit deployed in the VMs, and a metrics collector unit. Fig. 7 shows a block diagram of these units. The sensor management unit manages the start and stop of the sensor. When the service initially deployed on the system or the event detection unit detects changes in the topology, a controller in the sensor management unit calls the topology analysis unit and acquires the current topology of the system. From the acquired topology and a monitored service information to be set in a config, a sensor list is created. The sensor list includes VNIC names that the sensor is attached and IP addresses of VMs that the VNIC is located. The sensor list is compared with the previous sensor list to determine the state of the sensor to be deployed. That is, the sensors should be added, stayed, or deleted. Then, a sensor deployment block sends instructions to a sensor agent unit, which is installed in each VM. A sensor controller of the agent unit starts and stops the eBPF sensor program based on the instruction from the sensor management unit. The measured metrics is sent to a metrics collector unit with the service information. The collector pushes the data to the database periodically. Elasticsearch [16] is used as the database.

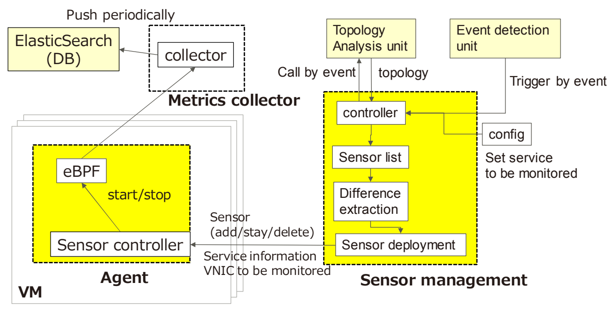


Fig. 7. Block diagram of a sensor management unit, agent unit, and metrics collector unit

We demonstrate the monitoring of packet latency between containers as a network quality. Fig. 8 shows the principle of the latency measurement when two service pods are communicated using the Transmission Control Protocol (TCP). We adopted a passive monitoring method that monitors the actual packets of applications. Generally, in the case of microservice applications, there are a proxy container inside the pods. The proxy container monitor the service-level latency. Envoy is a standard proxy technology for this purpose [17]. A service container shares the network namespace with the proxy container inside the same pod, so the service container sends messages to the proxy container by sending the messages at the localhost.

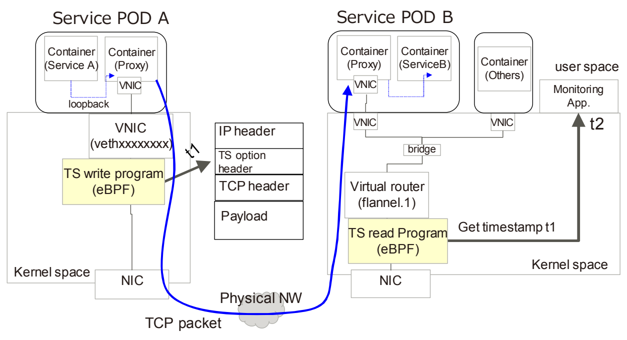


Fig. 8. One-way latency measurement between two containers

An eBPF sensor is attached to a veth device of a pod for the transmitter side to acquire TCP packets through the devices. Because the current eBPF can only capture the packets received in the kernel, the eBPF sensor was attached to a flannel.1 device for the receiver side. An eBPF program of the transmitter writes the timestamp (t1) in the TS option headers of the acquired packet. An eBPF program of the receiver reads the timestamp and is compared with a current timestamp (t2) by a monitoring program to measure the one-way latency. Communication sessions to multiple pods of the receiver VM comes to the flannel.1. IP address, and the port number was utilized for filtering the packets to acquire only the packets related to the monitoring service. A pod IP address was used for the transmission side. A pod IP address and a listening port number were used for the receiver side.

V. EVALUATION

We evaluate the developed monitoring system with one-way-latency measurement and verified the dynamic sensor deployment. A sample service composed of six microservices is containerized. The containers are executed on VMs. The VMs are located on physical machines. Table I shows versions of the management software used in this experiment.

We use a simple book review service named "Bookinfo", as the microservice to be monitored. BookInfo consists of six small microservices (Productpage, Reviews-v1, Reviews-v2, Reviews-v3, Ratings, and Details) which communicate with one another. Fig. 9 shows a service mesh architecture of the "BookInfo." Each microservice is executed on each container. Users who access the web application send requests to the Productpage service. The Productpage calls one of the Reviews services and Details service. Requests to the Reviews services are distributed by an application-level load balancer of Istio [18]. Finally, the selected Reviews service calls the Ratings service.

Table I. Software version

Type	Management software
Service mesh	Istio-1.2.2
Container orchestrator	Kubernetes-v1.14.3
Container runtime	Docker://18.9.7
VM management	Openstack 3.16.2
Database of metrics	ElasticSearch 5.6.0
Visualization of metrics	Kibana 5.6.3
Operating system of VM	Ubuntu 19.04/Kernel:5.0.0-37-generic

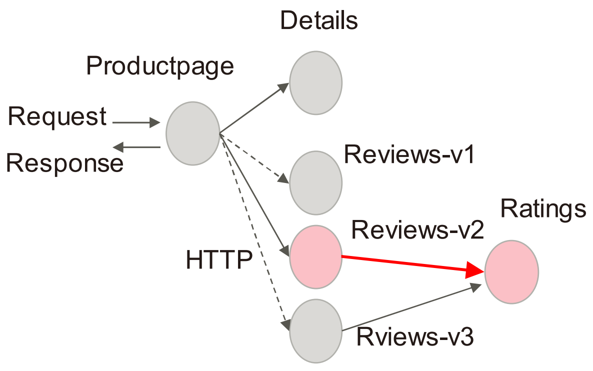


Fig. 9. Microservice used for an evaluation.

The evaluation setup is shown in Fig. 10. We measure the one-way latency of the service path from the Reviews-v2 to the Ratings. For comparison, a service-level request latency is measured by using Jaeger, a conventional APM software. The service latency is the time from the arrival of a service request to the Reviews-v2 until the Reviews-v2 returns a response to the Productpage. This function includes the local computation time of the Reviews-v2 and Ratings in addition to the round-trip communication time. The service latency is actually logged in an envoy container of the Reviews-v2 pod. Our monitoring system monitors the one-way latency between the Reviews-v2 container and Ratings container. One superior aspect of our monitoring system is that it is possible to extract the packet delay of the network infrastructure from the conventional service-level latency. This process helps to isolate the cause of the service's latency increase, whether it is in the application or infrastructure side.

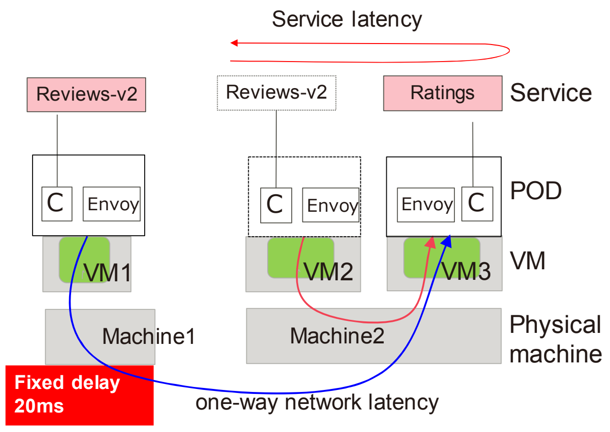


Fig. 10. Evaluation setup of one-way latency.

An experiment is conducted on the assumption that the pod accidentally moves from physical machine 2 to physical machine 1. A delay occurs in the physical network between machine 1 and machine 2. Such a case happens when networks of machine 1 have troubles or machine 1 is distant from machine 2. That is easily happen when multiple microservices are deployed on multi-cloud environment. Recently, a federation technology of container orchestrators enables a flexible utilization of the infrastructure geographically separated for the container [19]. The one-way latency between the Review-v2 and the Ratings is tested by inserting a 20-ms delay in a transmitter qdisk of physical machine 1 by using the tc command.

Fig. 11 shows the measured results. The pod was moved by applying a new setting to K8s, and an operation of a dynamic sensor deployment was evaluated. A service latency was monitored at microsecond time resolutions. A one-way latency was monitored at nanosecond time resolution. The service latency was increased by 20 ms when the pod moved from the VM2 of machine 2 to the VM1 of machine 1. On the other hand, the one-way latency between the containers was also increased by 20 ms. The measured data show that the service latency is increased by the delay of the network infrastructure.

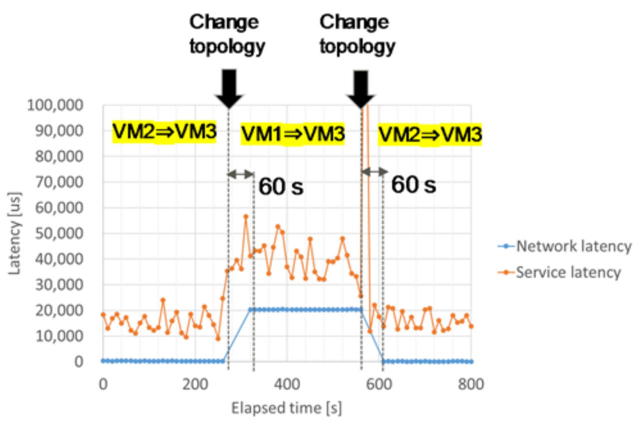
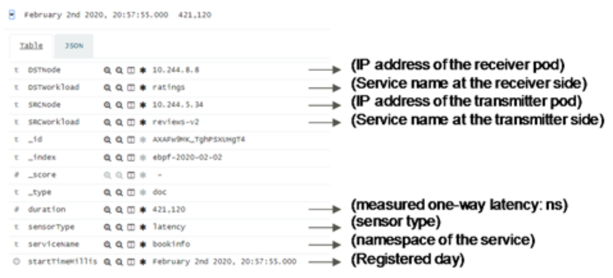


Fig. 11. Measurement results of the one-way latency between two service containers.

It took approximately 60 seconds to relocate eBPF sensors. This is a period when the sensor relocation was completed and the metrics acquisition was restarted, including the pod movement. The breakdown of the period was as follows. It took 34 seconds to complete the movement of the pod itself. It took 21.5 seconds from the event detection of the pod to metrics recollection in the database after the sensor relocation. The dead period of the monitoring system leads to missing important metrics for a system operator right after a system topology change. This time will be improved in our future work because the system failure often caused by the topology change.


February 2nd 2020, 20:57:55.000 421,120

t	DSTNode	10.244.8.8	(IP address of the receiver pod)
t	DSTWorkload	ratings	(Service name at the receiver side)
t	SRCNode	10.244.5.34	(IP address of the transmitter pod)
t	SRCWorkload	reviews-v2	(Service name at the transmitter side)
t	_id	AXAFw9MK_TghPSXUHgT4	
t	_index	ebpf-2020-02-02	
#	_score	-	
t	_type	doc	
#	duration	421,120	(measured one-way latency: ns)
t	sensorType	latency	(sensor type)
t	serviceName	bookinfo	(namespace of the service)
◷	startTimeMillis	February 2nd 2020, 20:57:55.000	(Registered day)

Fig. 12. Data format example of an eBPF metrics into the database.

Fig. 12 shows the data format of the measured metrics in the ElasticSearch database. We defined the original format. The measured metrics of the eBPF sensor was registered with the service names at the transmitter side and the receiver side. By using these service information as key, the past metrics of the infrastructure can be retrieved as consistent time-series metrics linked with the service even though the topology of the infrastructure was changed.

VI. Conclusion

We developed an eBPF-based real-time monitoring system that can monitor the latency between containers at a packet level. The real-time monitoring system can be utilized for maintaining the service quality of microservices by comparing the packet latency with service latency by APM. A real-time monitoring of the consistent service-linked metrics of infrastructure is achieved by creating a full-stack topology with services, containers, and VMs. In addition, the sensor can be dynamically relocated even though the container moves to another VM by detecting the event of K8s.

References

[1] P.Jamshidi, C. Pahl, N. C. Mendonca, J. Lewis and S. Tilkov, "Microservices: The Journey So Far and Challenges Ahead," IEEE SOFTWARE, pp.24-35, May/JUNE 2018.

[2] H. Liu, S. Chen, Y. Chen and W. Ding, "A High Performance, Scalable DNS Service for Very Large Scale Container Cloud Platforms", Middleware '18 Industry, December, 2018.

[3] J. Rahman and P. Lama, "Predicting the End-to-End Tail Latency of Containerized Microservices in the Cloud," IEEE International Conference on Cloud Engineering (IC2E), pp.200-210, 2019.

[4] B. H. Sigelman, L. A. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan and C. Shanbhag, "Dapper, a Large-Scale Distributed Systems Tracing Infrastructure," Google Technical Report dapper-2010-1, April, 2010.

[5] Jaeger. (2020, 5) [Online]. https://www.jaegertracing.io/docs/1.18/architecture/

[6] Zipkin. (2020, 5) [Online]. https://zipkin.io/

[7] Cadvisor. (2020, 1) [Online]. Available: https://github.com/google/cadvisor/

[8] P. Goyaland A. Goyal, "Comparative Study of two Most Popular Packet Sniffing Tools- Tcpdump and Wireshark," 9th International Conference on Computational Intelligence and Communication Networks (CICN), pp.77-81, September, 2017.

[9] F. Moradi, C. Flinta, A. Johnsson and C. Meirosu, "On Time-Stamp Accuracy of Passive Monitoring in a Container Execution Environment," IFIP Networking 2016 conference, pp.117-125, May, 2016.

[10] D. Scholz, D. Raumer, P. Emmerich, A. Kurtz, K. Lesiak and G. Carle, "Performance Implications of Packet Filtering with Linux eBPF," 30th International Teletraffic Congress (ITC 30), September, 2018.

[11] Cisco Tetration. (2020, 1) [Online]. https://www.cisco.com/c/en/us/products/security/tetr ation/index.html

[12] X. Zhou, X. Peng, T. Xie, J. Sun, C. Ji, W. Li and D. Ding, "Fault Analysis and Debugging of Microservice Systems Industrial Survey, Benchmark System, and Empirical Study," IEEE Transction onf Sowtware Engineering, vol. 14, No. 8, August, 2018.

[13] Kube-apiserver. (2020, 1) [Online]. Available: http://kubernetes.io/docs/admin/kube-apiserver/

[14] H. Zeng, B. Wang, W. Deng and W. Zhang, "Measurement and Evaluation for Docker Container Networking," International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery (CyberC), pp.105-108, October, 2017.

[15] Kubewatch. (2020, 1) [Online]. Available: https://github.com/bitnami-labs/kubewatch

[16] Elasticsearch. (2020, 1) [Online]. Available: https://www.elastic.com/

[17] Envoy. (2020, 1) [Online]. Available: https://github.com/envoyproxy/envoy

[18] Istio. (2020, 1) [Online]. Available: https://istio.io/

[19] Kubefed. (2020, 1) [Online]. Available: https://github.com/kubernetes-sigs/kubefed