Vol.2022-OS-157 No.9
Vol.2022-IOT-59 No.9
2022/9/14

eBPF-based Packet Tracing for Service Mesh

Chunghan Lee[1,a)] Reina Yoshitani[2] Toshio Hirotsu[2]

Abstract: Large scale microservice applications have been arisen and the application's architecture and container overlay networks on servers have also been complex. Although the distributed tracing for the service mesh is widely adopted in the microservice applications, it only focuses on latency-based monitoring and service discovery at an application layer. It is still challenging to monitor the container overlay networks to collaborate with the service mesh. In this paper, we present a packet tracing method using eBPF for the latency measurement based on the service mesh on the container overlay network. To detect the distributed tracing context on a HTTP header efficiently, we move the location of the tracing context at the head of the HTTP on sidecar proxy. Our tracing method gathers the HTTP packets that have the tracing context and measures the latency using eBPF. Our evaluation using an open-source benchmark on Kubernetes shows that there is no significant change of end-to-end latency using the proposed tracing header format. Moreover, our eBPF tracing method presents the similar latency characteristics on the container overlay network in comparison with *tcpdump*.

Keywords: eBPF, Latency, Sidecar, Kubernetes(K8s), ServiceMesh, Microservices, Distributed tracing

1. Introduction

Applications based on microservice architecture are widely adopted in production, such as Uber [26], Lyft [19], Netflix [21], and Amazon [1]. In particular, Kubernetes (K8s) [15] and container technologies [4], [6] have been also used as main components for the microservices. Moreover, the scale of microservice applications is large and their architecture is also complex. Uber [18] has 35 K8s clusters that each cluster consists of 100,000 containers and Lyft [20] runs 300,000 containers on multiple K8s clusters. To support connectivity between the microservices, the container overlay networks [9], [23] in Linux kernel are popularly used. Due to such hyperscale microservices, the container overlay network is also complex and it would be a performance bottleneck point.

For microservices monitoring and service discovery, the distributed tracing [14], [24], [29] that inserts extra tracing context [5] to a HTTP header and measures latency between the microservices for the service mesh [13] is widely adopted. However, the distributed tracing focused on the latency measurement on microservices' application layer. Thus, there is no consideration of infra. layer, such as the container overlay network. To enhance the network monitoring and debugging in the kernel, extended Berkeley Packet Filter (eBPF) [7], [28] has been used. eBPF handles not only packet processing but also generic events in Linux kernel.

eBPF is promising for the latency-based monitoring on the container overlay network, however, existing monitoring methods [11], [25] using eBPF have the following limitations to collaborate with the service mesh. First, there is no consideration of microservice's service mesh and it is hard to discover the entire microservices topology on the container overlay network. Second, the previous works only focused on the latency measurement on IP/TCP/UDP protocols. Thus, there is no support of L7 protocols, such as HTTP and GRPC. Third, the detection of the distributed tracing context is also hard because the context is at the end of HTTP header.

To overcome the limitations of the existing methods, we present an eBPF-based packet tracing method for the service mesh. First, we move the distributed tracing context to the head of HTTP header by modifying the sidecar proxy [8] that attaches and detaches the distributed

[1] Toyota Motor Corporation
[2] Hosei Uniersity
[a)] lch@toyota-tokyo.tech

© 2022 Information Processing Society of Japan

tracing context. Next, we gather the K8s cluster information for automatically deploying eBPF programs that detect the distributed tracing header and mark timestamps on the container overlay network. Finally, we deploy the eBPF programs to dedicated tracing points and measure the latency using the timestamps on the container overlay network

In this paper, we will show 1) a comprehensive view of our eBPF-based tracing method to measure the latency on the container overlay network (§3), 2) the performance and the detailed analysis of our eBPF-based tracing (§4). Furthermore, in the evaluation of our eBPF-based tracing, we will present i) the side impact of proposed tracing header format through end-to-end latency measurement (§4.1), ii) the efficiency of proposed tracing header (§4.2), and iii) the latency characteristics using eBPF on the overlay network (§4.3). All of these results will help in monitoring the container overlay network and diagnosing performance problems from the aspect of latency.

2. Related work

Tracing-based monitoring methods. There have been a number of tracing-based monitoring methods [2], [10], [22], [24]. To the best of our knowledge, none of the existing approaches could not consider the distributed tracing for the service mesh to measure the latency on the container overlay network. X-Trace [10] provides the telemetry-based tracing on multiple layers (HTTP/TCP/IP). X-Trace inserts and removes extra metadata like the distributed tracing context. Moreover, it provides a comprehensive view of tracing results. X-Trace does not handle traffic on the container overlay network. Dapper [24] focuses on diagnosing and debugging of applications using the distributed tracing. It provides transparency between applications. This tracing scheme is widely used for other distributed tracing methods, such as Zipkin and Jaeger. Niu et al., [22] expanded INT [12] for IP-Over-Optical Network. It provides a view of multiple layers: IP and optical networks. Their INT-based approach does not handle packets related to the distributed tracing. Ashok et al., [2] shown that the service mesh can be possible to be expanded as a network layer for network routing and monitoring. However, their approach only focused on the routing and finding network bottleneck.

Latency Measurement using eBPF. eBPF is widely used for kernel tracing and debugging. Suo et al., [25] developed vNetTracer for packet tracing in the software-based virtual network. In vNetTracer, the extra unique headers are added to both TCP and UDP packet headers and the extra headers are used for the packet tracing in Linux kernel. vNetTracer can detect overhead points using the eBPF-based latency measurement in the kernel. However, there is a lack of consideration of microservice’s service mesh. HostINT [11] using eBPF is similar to our tracing method. It collaborates with INT [12] and provides the latency measurement of ICMP/TCP/UDP traffic. But, there is no consideration of HTTP and distributed tracing context parsing.

3. Design and Implementation

We show the design overview of our eBPF tracing method in Figure 1. There are five components to measure the latency on the container overlay network.

1. Sidecar proxy (§3.1) attaches and detaches the distributed tracing context [5] (TraceID, ParentSpanID, SpanID) on the HTTP header. To reduce the search space of the tracing context using eBPF, we move the tracing context at the head of HTTP and fix the location of the tracing context.

2. Trace info. collector (§3.2) consists of four subcomponents. It collects the information of K8s cluster from docker [6], istio [13], *iptables*, and the K8s master node.

3. eBPF dispatcher (§3.3) deploys eBPF programs to measure the latency with the filtering information and the tracing point on the container overlay network.

4. Agent (§3.4) collects timestamps from the eBPF program and calculates the latency.

5. eBPF program (§3.5) is in charge of filtering the packets that have both the HTTP header and the distributed tracing context and timestamping the filtered packets on the tracing points.

Our control plane (§3.2, §3.3) periodically gathers the K8s information from the K8s cluster. Then, the eBPF programs are deployed to filter the HTTP packets with the tracing context and to mark the timestamps to the HTTP packets. Finally, the agent gathers the timestamps and calculates the latency.

3.1 Sidecar proxy

We re-design the distributed tracing context location efficiently to detect the tracing context using eBPF on the container overlay network. In the original header format (Figure 2(a)), the distributed tracing context is located at the end of HTTP header. Due to the large search space on the HTTP payload, it is not easy to detect the trac-

© 2022 Information Processing Society of Japan

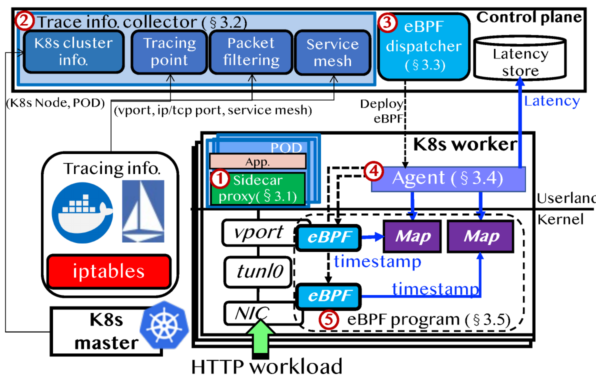
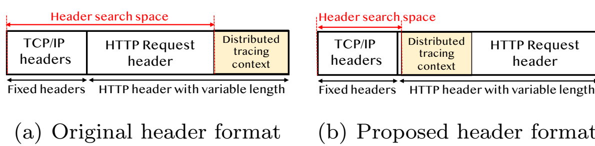


図 1　Design overview of eBPF-based tracing method.



(a) Original header format　(b) Proposed header format

図 2　Distributed tracing context location.

ing context using the eBPF. As the worst case, the eBPF cannot detect the distributed tracing context. To detect the tracing context using the eBPF on the container overlay network, we move the tracing context at the head of HTTP and fix the location of the tracing context (Figure 2(b)).

3.2 Trace info. collector

The trace info. collector monitors the K8s cluster and collects the information to configure the eBPF program for the latency measurement on the container overlay network. The K8s cluster info. component gathers the information of pods, nodes, and K8s namespaces via *kubectl*. It also collects the IP addresses at the pods.

The tracing point component collects a mapping information between a virtual interface (vport) and container's namespace on docker. To gather the mapping information, we use the vport and container's process IDs. Next, the packet filtering information for the filtering component is gathered from *iptables*. We estimate the packet filtering information on NAT tables on *iptables* using the collected K8s pod names. Finally, the service mesh is collected from istio. In detail, Jaeger [14] is used to gather the microservice's topology from HTTP requests through the microservices. If the service mesh is dynamically changed by users or auto-scaling, our collector also collects the additional information. The influxDB is used for the latency store.

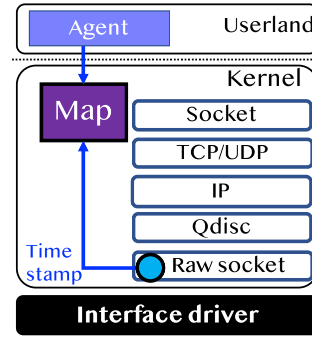


図 3　eBPF program location at network stack.

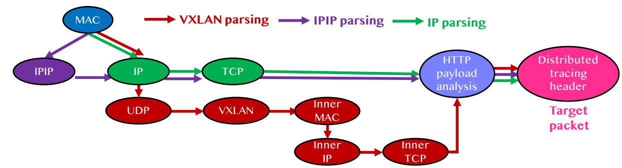


図 4　eBPF packet parser.

3.3 eBPF dispatcher

The eBPF dispatcher checks the names of NIC and vport. In our K8s worker node, the NIC indicates a hardware network interface to connect to network or other servers and the vport indicates a software virtual interface to connect to the K8s pods. The packet filtering information (inner/outer ip addresses and tcp port numbers) from the trace info. collector and confirms the existence of such resources on the K8s cluster. The dispatcher creates the eBPF program with the above configuration and sends the eBPF program to the K8s worker nodes.

3.4 Agent

The agent is located at the K8s worker nodes and deploys the eBPF programs to the dedicated tracing points (e.g., the NIC and the vport). We carefully select the two tracing points to consider the overhead of eBPF deployment. To measure the latency on the container overlay network, the two eBPF programs are simultaneously deployed to the tracing points. The agent periodically gathers the timestamps from the maps (eBPF Maps), calculates the latency between the timestamps, and forwards the latency to the store on the control plane.

3.5 eBPF program

The eBPF program is deployed to the dedicated trace points on the container overlay network. This program is located to raw socket layer at network stack to gather all of packet headers (Figure 3). Thus, we can access to L2/L3 layer packet information fully. The reason why

© 2022 Information Processing Society of Japan

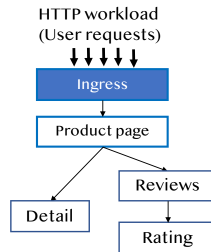


図 5 Microservices application (Bookinfo).

we use eBPF for the latency measurement is that eBPF can process both RX/TX direction packets. Although XDP provides the high performance of packet processing in comparison with eBPF, XDP can handle RX direction packet only. It is a drawback to measure the latency on the container network.

The eBPF program consists of packet parser with filtering, HTTP payload analysis, and timestamp generation/insertion. In the packet parser (Figure 4), VXLAN/IPIP protocol parsing is supported because these protocols and CNI plugins [9], [23] are widely adopted for the container overlay network. Next, the packet headers are parsed with the designated packet headers like P4 programmable packet parser and the five tuples (protocol, source ip/port and destination ip/port) at the matched packets are extracted during the packet parsing.

After the TCP/IP header parsing, the designated number of bytes is read and the HTTP payload analysis checks whether the HTTP header is included or not. If there is no HTTP header at the packet, the packet is just forwarded to destination (e.g., a container or network). If there is HTTP header, the fixed size of HTTP payload is read and the distributed tracing context are searched. If the distributed context is observed, the nano scale timestamp is marked. Next, the timestamp, the five tuples, the TCP sequence number, and the tracing context are stored to the map (eBPF map). The sequence number is important to identify the same packet at the trace points.

3.6 Implementation

We implement a prototype that measures the latency on the container overlay network. First, 0.1K lines of C++ code are added for the modification of sidecar proxy. Next, the control plane module that collects the information of K8s cluster is implemented in 1.0K lines of Python code. Then, the agent that calculates the latency and sends the latency is implemented in 0.5K lines of Python code. Finally, the eBPF program for the latency measurement is implemented in 1.0K lines of both C and Python codes.

4. Evaluation

We evaluate our eBPF-based tracing method to answer the following questions:

- Is there any side impact of the modification of sidecar proxy?
- How many search space can be reduced using the proposed header format?
- Is it possible to measure the latency using the eBPF on the container overlay network?

Latency Measurement Setup. We evaluate our tracing method using Bookinfo microservice application [3] in Figure 5. Kubernetes is used as container orchestration and Calico [23] (IPIP mode) is used for the container overlay network. We run the K8s cluster (v.1.22) and the agent on five machines with two Intel Xeon (E5-2650) CPUs, 128GB of memory, and Mellanox ConnectX-5 (100 Gbps). Istio (v.1.12.1) is used for the service mesh. Ubuntu 20.04.3 LTS (kernel version: 5.15.0-33) is used for the K8s cluster. We use one machine for the K8s master node, four machines for the K8s nodes, and the last one machine for the control plane module. We deploy the bookinfo application that consists of four microservices on the different K8s nodes (Figure 6). The ingress gateway is used as TLS termination proxy, handles the traffic from users, and generates the distributed tracing context for the service mesh. For the ingress gateway, the same type of sidecar proxy in the K8s pod is adopted.

To generate the dedicated HTTP workload, we use vegeta [27] as HTTP load testing tool. We prepared one machine that has the same spec. of K8s node and generated 1 Queries Per Second (QPS) and 80 QPS. If the number of QPS is larger than 80 QPS, the end-to-end latency is increased in seconds. To prevent the unexpected end-to-end latency, we fix the maximum workload as 80 QPS.

To reduce the timestamp fluctuation and the impact of unexpected interrupts that affect the latency measurement, we separate the CPUs for the eBPF program on the K8s node and the CPUs are only used for the eBPF program. We refer to the server configurations from [16], [17] for the latency measurement.

4.1 End-to-end latency

In this section, we measure the end-to-end latency to observe the impact of sidecar modification. If there is the side impact of sidecar modification, it would be observed

© 2022 Information Processing Society of Japan

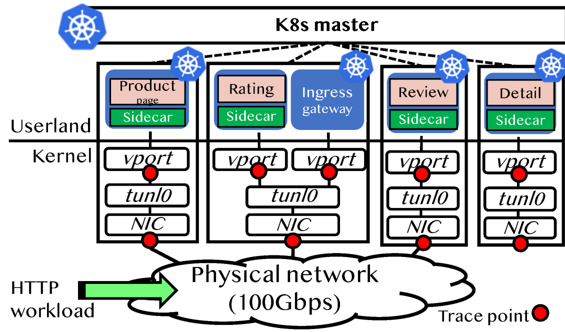
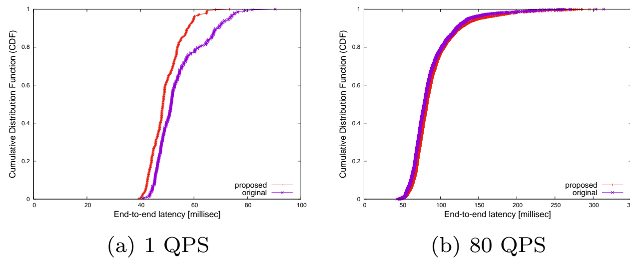


図 6　K8s cluster with Bookinfo application.

表 1　Statistics of end-to-end latency measurement results.

HTTP workload	Header format	Min [ms]	Median [ms]	Mean [ms]	Max [ms]
1 QPS	Original	40.7	51.5	54.2	90.4
	Proposed	39.4	48.4	49.4	73.3
80 QPS	Original	43.2	79.1	86.7	313.8
	Proposed	44.8	82.3	91.2	304.9



(a) 1 QPS　　(b) 80 QPS

図 7　CDF of end-to-end latency.

to the end-to-end latency. We prepare the two types of the sidecar: the original and the proposed header format and generate the two types of workload: 1 QPS and 80 QPS.

The statistics of end-to-end latency measurement are shown in Table 1. The similar statistics of latency are shown. While the proposed header format is slightly faster than the original header format with 1 QPS, the mean and median latency of the original header is smaller than those of the proposed header format with 80 QPS. To see the entire latency characteristics, the CDF curves are shown in Figure 7. There is a difference between the proposed and the original headers with the 1 QPS. Although the statistics of the proposed header are slightly higher than those of the original header, we cannot find any significant characteristics of the CDF curves with 80 QPS. From the results, the similar latency characteristics are observed by the two different header formats and their performance gap is negligible.

4.2 Distributed tracing context detection

To answer the second question, we control the HTTP payload search space from 100 bytes to 800 bytes to detect the distributed tracing context. When the number of search spaces is increased to 900 bytes, the number of instructions is larger than 4096 and we cannot execute our eBPF program. It would be possible to use tail call to overcome the instruction limitation. But, the tail call is out of our scope.

In the product page (Figure 5), there are three types of distributed context. The first one is from the ingress gateway, the second one is to the detail, and the last one is to the reviews. Thus, the number of SPANs in the product page is higher than the other microservices applications. Because of the above reason, we choose the product page as our evaluation target and generate the same HTTP workloads (1 QPS and 80 QPS) to the bookinfo application.

We show the tracing context detection rate in Figure 8. With the small search space (100 bytes), the eBPF program cannot detect the tracing context fully. However, the eBPF program can parse the tracing context provided by the proposed header format after the search space is increased in 200 bytes. Meanwhile, the original tracing header format cannot be detected fully. The major cause of decreased detection rate of original format is that there are many extra distributed tracing contexts in the particular HTTP header and the HTTP header length is changed by both HTTP types and distributed tracing decorators. In detail, the length of distributed tracing metadata from the ingress gateway is approximately 1000 bytes and it is too long to parse it using the eBPF program.

From the results, the search space of the proposed tracing is smaller than that of the original format and it is enough to show the efficiency of the proposed header format. The proposed header format would be more friendly to another network infra. layers, such as physical network (underlay network) to detect the distributed tracing context.

4.3 Latency on container overlay network

In this section, we show the latency measurement results on the overlay network. First, we apply the sidecar proxies to the bookinfo to generate the proposed header format and deploy the eBPF programs to the tracing points (the NIC and the vport) in Figure 6. In the eBPF program, the payload search space is fixed as 200 bytes. Next, we simultaneously run tcpdump on the tracing points in comparison with the results from the eBPF program and use ramdisk to reduce the packet capture

© 2022 Information Processing Society of Japan

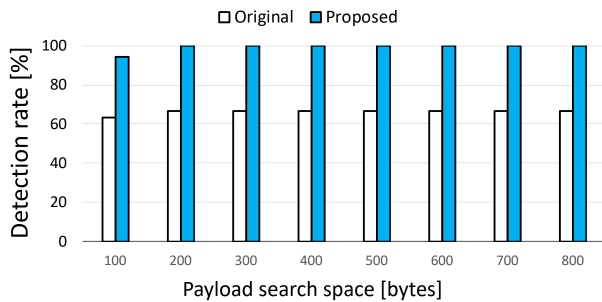
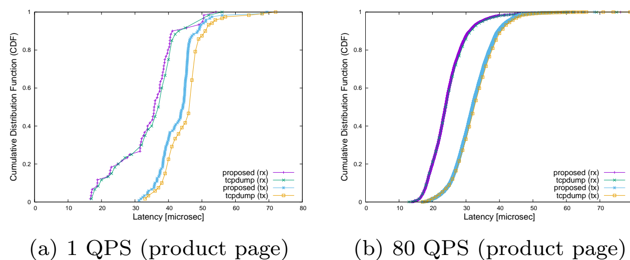


図 **8** Distributed tracing context detection rate (Due to the limitation of instructions (4096), we cannot use the large search space (e.g., 900 bytes)).



(a) 1 QPS (product page) (b) 80 QPS (product page)

図 **9** CDF of latency on container overlay network.

loss. The tcpdump and server configurations are referred from [17]. Although the timestamping point between tcpdump and the eBPF program is different in Linux kernel, tcpdump is useful to identify the five tuples and the tracing context. Moreover, we will show the CDF curves of both tcpdump and the eBPF. Finally, we gather the HTTP headers with the proposed tracing header format and packet-level traces from tcpdump. The HTTP workloads and the tracing points are same to the tracing context detection experiments.

Through off-line analysis, we extract the same packets from the packet traces using the tcp seq. of the eBPF results and classify the packets into two type directions: rx (NIC→vport) and tx (vport→NIC). In here, we show the CDF curves of the product page in Figure 9. The other CDF curves are also similar to Figure 9, but they are omitted for the sake of brevity. Although the timestamping point is different between tcpdump and our eBPF-based tracing method, the similar CDF curves are observed. With 1 QPS, the mean latency of the eBPF for the rx (NIC→vport) is 34.2 μs and that for tx (vport→NIC) is 43.1 μs. With 80 QPS, the mean latency for the rx (NIC→vport) is 25.0 μs and that for tx (vport→NIC) is 32.2 μs.

5. Conclusion

In this paper, we focused on the latency measurement for the service mesh on the container overlay network by using eBPF. Although the distributed tracing is widely used for the service mesh, existing distributed tracing methods do not consider the container overlay network. Our eBPF-based tracing method utilizes the sidecar proxy that has the proposed header format and the eBPF program that detects the proposed header on the overlay network. We show the effectiveness of our eBPF-based approach through the evaluation using the open-source microservice application.

Our future work involves evaluating our eBPF tracing method to other microservice applications and enhancing the distributed context detection on other protocols. We plan to expand our tracing method to measure the latency on multiple sites and container underlay network (physical network).

参考文献

[1] Amazon: `https://www.amazon.com/` [Online].
[2] Ashok, S., Godfrey, P. B. and Mittal, R.: Leveraging Service Meshes as a New Network Layer, *Proceedings of the Twentieth ACM Workshop on Hot Topics in Networks (HotNets '21)*, HotNets '21, p. 229–236 (2021).
[3] Bookinfo Application: `https://istio.io/latest/docs/examples/bookinfo/` [Online].
[4] containerd: `https://containerd.io/` [Online].
[5] Distributed Trace Context: `https://www.w3.org/TR/trace-context` [Online].
[6] Docker: `https://www.docker.com/` [Online].
[7] eBPF: `https://ebpf.io/` [Online].
[8] Envoy: `https://www.envoyproxy.io/` [Online].
[9] flannel: `https://github.com/flannel-io/flannel` [Online].
[10] Fonseca, R., Porter, G., Katz, R. H., Shenker, S. and Stoica, I.: X-Trace: A Pervasive Network Tracing Framework, *4th USENIX Symposium on Networked Systems Design and Implementation (NSDI 07)*, pp. 271–284 (2007).
[11] Host-INT: `https://github.com/opennetworkinglab/int-host-reporter` [Online].
[12] In-band Network Telemetry (INT): `https://p4.org/assets/INT-current-spec.pdf` [Online].
[13] Istio: `https://istio.io/` [Online].
[14] Jaeger: `https://www.jaegertracing.io/` [Online].
[15] Kubernetes (K8s): `https://kubernetes.io/` [Online].
[16] Lee, C., Mori, N., Ohara, Y., Murakami, T., Asaba, S. and Matsushima, S.: The Latency Characteristics of GTP-U and SRv6 Stateless Translation on VPP Software Router, *2021 IEEE 45th Annual Computers, Software, and Applications Conference (COMPSAC)*, pp. 1429–1436 (2021).
[17] Lee, C., Asano, K. and Ishihara, T.: The Impact of Software-based Virtual Network in the Public Cloud,

© 2022 Information Processing Society of Japan

2018 4th IEEE Conference on Network Softwarization and Workshops (NetSoft), pp. 494–499 (2018).

[18] Liu, Y.: Only Slightly Bent: Uber's Kubernetes Migration Journey for Microservices, https://www.youtube.com/watch?v=91c3iUI2K7M [Online].

[19] Lyft: https://www.lyft.com/ [Online].

[20] Lyft Runs 300,000+ Containers in a Multicluster Kubernetes Environment: https://www.altoros.com/blog/lyft-runs-300000-containers-in-a-multicluster/ [Online].

[21] Netflix: https://www.netflix.com/ [Online].

[22] Niu, B., Kong, J., Tang, S., Li, Y. and Zhu, Z.: Visualize Your IP-Over-Optical Network in Realtime: A P4-Based Flexible Multilayer In-Band Network Telemetry (ML-INT) System, *IEEE Access*, Vol. 7, pp. 82413–82423 (2019).

[23] Project Calico: https://www.tigera.io/project-calico/ [Online].

[24] Sigelman, B. H., Barroso, L. A., Burrows, M., Stephenson, P., Plakal, M., Beaver, D., Jaspan, S. and Shanbhag, C.: Dapper, a Large-Scale Distributed Systems Tracing Infrastructure, Technical report (2010).

[25] Suo, K., Zhao, Y., Chen, W. and Rao, J.: vNetTracer: Efficient and Programmable Packet Tracing in Virtualized Networks, *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*, pp. 165–175 (2018).

[26] Uber: https://www.uber.com/ [Online].

[27] Vegeta: https://github.com/tsenart/vegeta [Online].

[28] Vieira, M. A. M., Castanho, M. S., Pacífico, R. D. G., Santos, E. R. S., Júnior, E. P. M. C. and Vieira, L. F. M.: Fast Packet Processing with EBPF and XDP: Concepts, Code, Challenges, and Applications, *ACM Comput. Surv.*, Vol. 53, No. 1, pp. 1–36 (2020).

[29] Zipkin: https://zipkin.io/ [Online].

© 2022 Information Processing Society of Japan